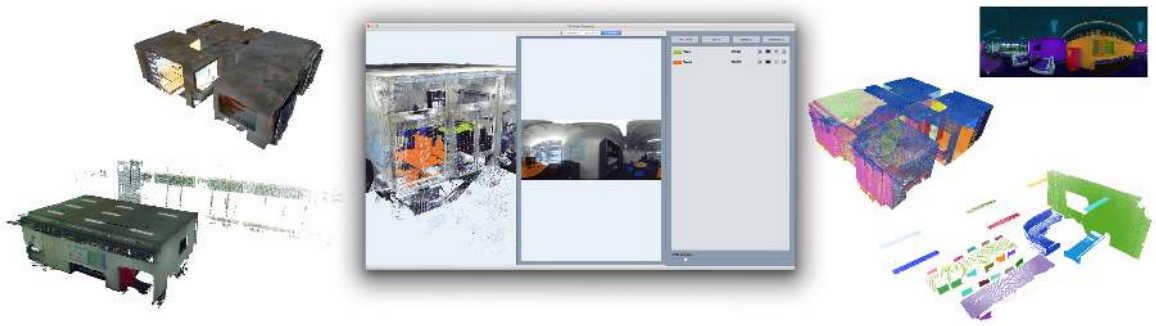
A Toolkit for the Geometric and Semantic Annotation of Point Cloud Data



Master Thesis

1st July 2020 - 31th December 2020

Francesca Monzeglio

Minusio, Ticino, Switzerland

14-732-853

Supervisors:
Prof. Dr. Renato Pajarola
Dr. Claudio Mura

Visualization and MultiMedia Lab
Department of Informatics
University of Zurich

University of Zurich[UZH]

VISUALIZATIONANDMULTIMEDIALAB

Cover image: 3D point cloud data is annotated with functionalities of the *PointCloudAnnotator*

Abstract

The number of applications requiring large point cloud data sets is increasing. Sensors, such as Light Detection and Ranging (LiDAR), are able to produce large point clouds as raw data. Therefore, this data must undergo a process of annotation in order to obtain semantic meaning. Many data science applications exploit deep-learning techniques which require large amounts of annotated data to train the underlying neuronal network.

Annotation must generally be performed manually. As data sets increase in complexity, time and effort needed for the annotation also increase. Therefore, general-purpose tools to simplify and accelerate the process of annotation can bring benefit to the entire process of data analysis. In this thesis such a toolkit is developed, the *PointCloudAnnotator*.

The *PointCloudAnnotator* is able to load point clouds as unstructured lists of points or as a collection of registered panoramic depth maps. The points can then be visualised as a 3D model. Additionally, if the data is loaded as a collection of maps, each map can be visualised as an image. The *PointCloudAnnotator* includes functionalities such as grouping points either by selecting individual points, selecting regions of points, or by boolean union of existing groups. Depending on the user's needs, the selection of points can be done on the 3D-model, or on the map images. In addition to manual selection, the user is provided with tools for global or local segmentation of the point cloud, featuring various degrees of automation. The resulting groups are associated with a geometric primitive best approximating the shape of each group. Additionally, groups can be labelled by the user according to their semantic meaning.

For a toolkit to fully accomplish the goals of improving the overall annotation process, the provided tools must not only be efficient but also easy to use, as the *PointCloudAnnotator* aims to achieve.

Contents

List of Figures

1. Introduction

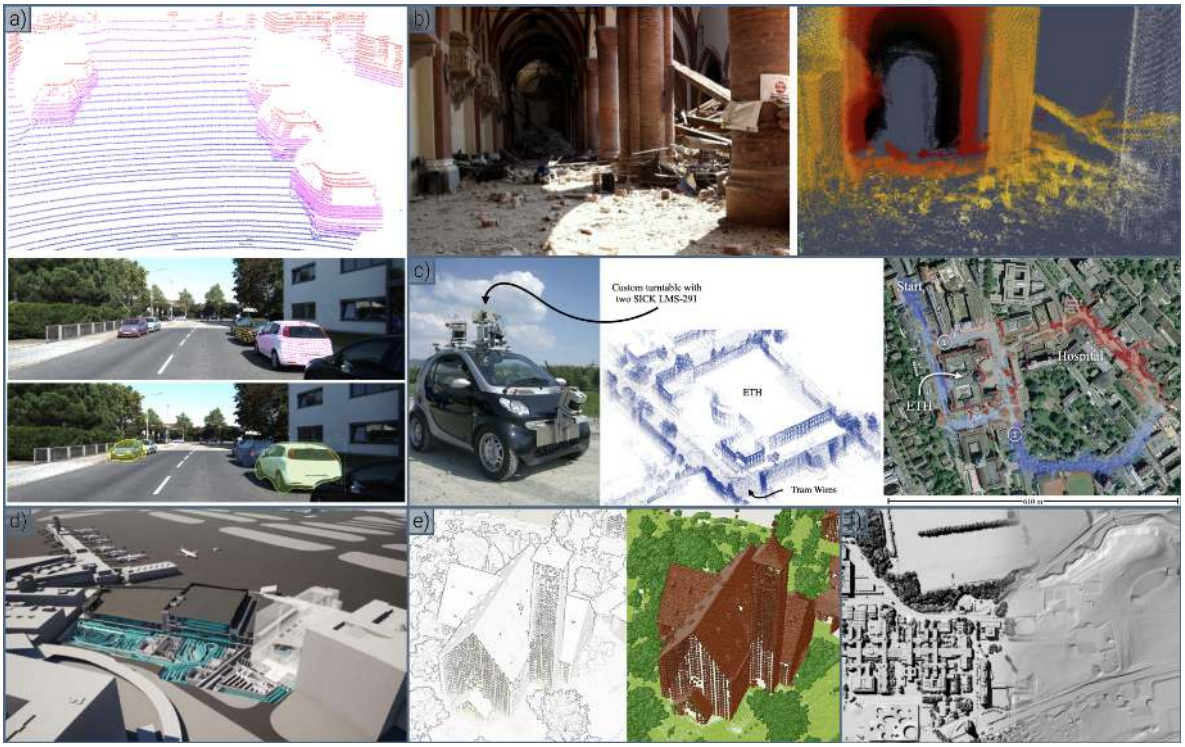


Figure 1.1.: Examples of application fields exploiting large point cloud data. a) Autonomous driving, o.a.o.[1] [38]. b) Reconstruction for damage assessment [32]. c) Reconstruction from autonomous vehicles within a urban environment, o.a.o. [32]. d) Visualisation of the project for the GSA extension at Zurich Airport[2]. e) Urban model in GIS [14]. f) From left to right, from swisstopo DSM to DTM [28].

[1] Own adaptation of.

[2] *ZONE A – Baggage sorting system (GSA)* at `https://www.zurich-airport.com/the-company/zurich-airport-ag/current-construction-projects/zone-a-baggage-sorting-system-gsa`. Accessed December 24th, 2020.

1.1. Motivation and Approach

An increasing number of application fields are exploiting the potential provided by large point cloud data capturing indoor and outdoor environments by means of 3D scanning technology such as Light Detection And Ranging (LiDAR) [31]. The constant progress of these devices enables the the production of large data sets in a relatively short amount of time.

Such devices, especially airborne scanners, are well established to gather geographical data mostly for the creation of urban areas' 3D models – see Figure 1.1.e) – or topographical models of the landscape. Examples in the last category include the swissSURFACE3D data set, the Digital Terrain Model (DTM) and the Digital Surface Model (DSM) produced by the Federal Office of Topography (swisstopo) [28]. A transition from DSM to DTM is visible in Figure 1.1.f).

In the field of architecture and building development, digital models in the format of Building Information Models (BIMs), are now widespread mediums to visualise building-related issues and convey information between architects, engineers, designers and other experts. See for example, the project for the renewal and expansion of the current baggage sorting system (GSA) at Zurich Airport, in Figure 1.1.d).

Mobile robotics platforms can be combined with 3D sensor devices to map the environment and thus support tasks with various applications. Some experiments with these platforms are analysed by Pomerleau et al. [32]. Robots can move in environments too dangerous or unreachable for human beings and thus greatly contribute in search and rescue missions, which include damage assessments or collaborative mapping. In the cover image, Figure 1.1.b), we see how a point cloud model of the Chiesa di San Francesco d'Assisi – in Mirandola, Italy – can be built from 3D photographic scans captured by a remotely controlled robot moving around the damaged site after a sequence of earthquakes hit. Figure 1.2.a) shows the maps of the Zurich firefighter training site resulting from tele-operated flying and ground devices as part of a collaborative mapping experiment.

In the context of environmental monitoring, Figure 1.2.b), shows the autonomous surface vessel used to reconstruct the shoreline of the Lake Zurich in support of the monitoring of freshwater bodies.

What can be considered a subfield of robotics and is increasingly gathering popularity is the industry of autonomous vehicles. These are usually equipped with LiDAR sensor to recognise the surrounding, constantly evolving environment and react accordingly. One application of such vehicles consists in serving as a robotic platform to reconstruct a 3D model of the travelled environment, as in the experiment reported by Pomerleau et al. [32] and illustrated in Figure 1.2.a).

As it will be briefly discussed in Chapter 4, new applications in the domain of indoor mapping are rising as well.

In order to be properly employed for the various above-mentioned applications, a learning process by means of manual annotation is generally required for the raw data that upon capturing still lacks any ground-truth semantic meaning. The recent trend towards this field of research is further demonstrated by the development of many deep-learning techniques for supervised 3D point cloud classification. Such systems are usually data-hungry as their underlying neural networks need to be trained with annotated data in considerable amounts. More or less general-purpose interactive tools for the annotation of 3D point clouds are thus necessary to solve such a demanding workflow, which usually consists of tedious and repetitive operations.

This thesis fits into the described context since it aims to develop a comprehensive set of effective tools for a quick annotation of point-based 3D scanned models. The tools will be part of a stand-alone application, the *PointCloudAnnotator*, which loads an input 3D scan and visualises it. Depending on the format, the input scan is provided either as a single unstructured list of points, PLY, or as a collection of registered panoramic depth maps, PTX files. In both cases the point cloud is displayed

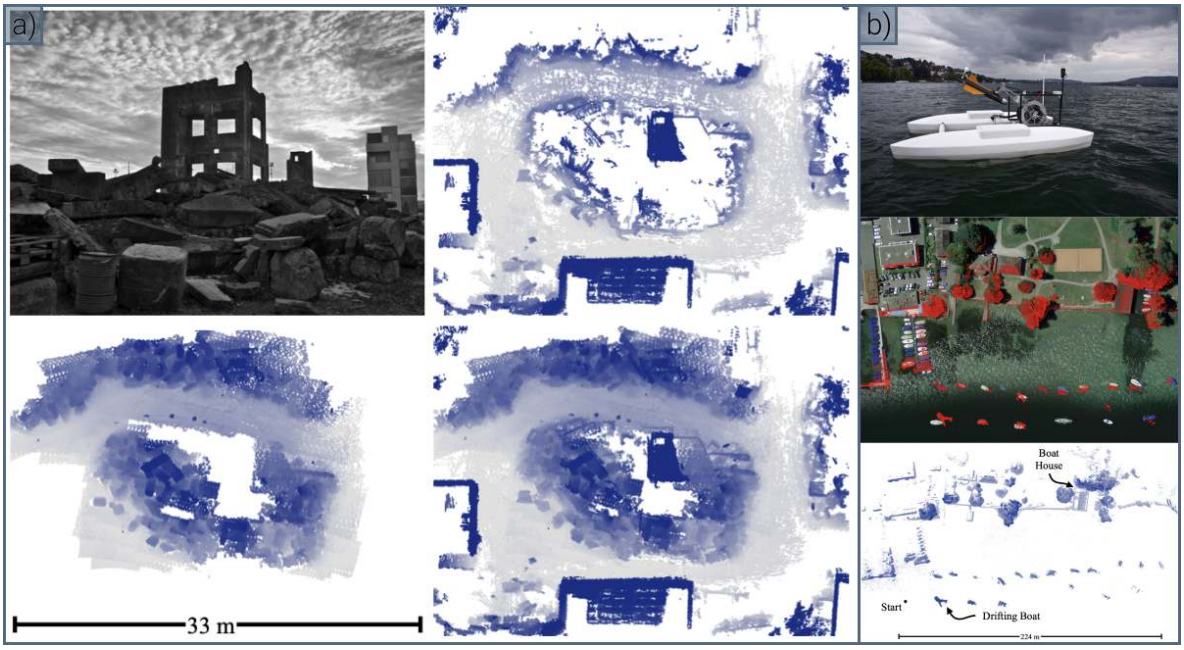


Figure 1.2.: Examples of mapping applications from 3D point cloud data. Reconstruction a) from flying and ground robotic platforms of the Zurich firefighter training site and b) from an autonomous surface vessel of the shoreline of the Lake Zurich. Image courtesy of [32].

as 3D model and in the second case, each map will be additionally visualised as separate image in the application.

Central to this project is the simplification of the manual annotation which consequently leads to an optimisation in terms of time demand. These goals are accomplished with the implementation of specific solutions within the *PointCloudAnnotator*. The user is allowed to annotate the given point cloud by interactively selecting regions of interest – either from the 3D model or, when available, from the images – in the form of groups of points and mark them with a meaningful label. This can either be newly typed or selected among the several ones provided as a separate input list. The result of the annotation process is then stored in a proper output format. Groups can also be combined with boolean union operation. It is worth stressing that the 3D selection of points must be computationally efficient, thus an appropriate spatial acceleration structure will embed the point cloud.

Besides these selection options, the system will additionally provide the automatic performance of a planar segmentation of the point cloud and the semi-automatic fit of non-planar primitives at a region selected by the user. The user is further allowed to calibrate input parameters for the global segmentation, which is then performed completely automatically by producing a set of individually labelled planar segments. The result of the local fit of non-planar shapes such as spheres and cylinders, among others, is also influenced by a set of possibly user-defined parameters, but is additionally user-supervised. The system indeed, recommends the best primitive based on the fitting error for each shape type and the user can then decide which one to definitively accept and label.

For a toolkit to deliver concrete improvements – in terms of time and effort – to the annotation process of complex data sets, the sole availability of powerful and efficient tools is not sufficient. These tools must in fact be additionally quick to understand and easy to use.

1.2. Outline

In order to fulfil the previously mentioned requirements, the technical part of this thesis can be subdivided into the following main tasks:

1. Input/Output management of scan data: implementation of the procedures to convert the input data into an appropriate format and of a simple API to manage data operations on disk, such as access, load and save.

2. Scan data visualisation: implementation of the main Graphical User Interface (GUI) of the application and of the visualisation logic, including both the 3D rendering of the point cloud and the plain 2D display of the panoramic depth maps.

3. Basic selection: implementation of simple selection operations, including the picking of individual points, a basic group selection mechanism for the 3D point cloud and a region-based selection mechanism for the panoramic maps.

4. Semi-automatic geometric segmentation: implementation of the automatic planar segmentation and of the user-supervised fitting of non-planar surfaces.

The detailed written description of the implementation based on C++ with the integration of Qt for the GUI and the support of external libraries for specific tasks, will be provided in Chapter 5. The chapter is preceded by three other chapters explaining necessary concepts and definitions. First, an overview on the context of large scan data annotation is given in Chapter 2 which introduces standing tools for the processing and annotation of point clouds, mainly categorised in three groups, general-purpose processing tools, specific-purpose annotation tools and general-purpose annotation tools. Chapter 3 provides a technical explanation of the concepts applied during the implementation, while the structures of input and output data are described in Chapter 4. The implementation chapter is followed by Chapter 6 which exposes the thesis' result as a visual presentation of the final state of the *PointCloudAnnotator* by means of a series of screenshots. Chapter 7 evaluates the toolkit by discussing valuable and critical aspects and proposes insights on possible directions for future work, while a wrap up of the achievements in Chapter 8 concludes the written part of the thesis.

2. Related Work

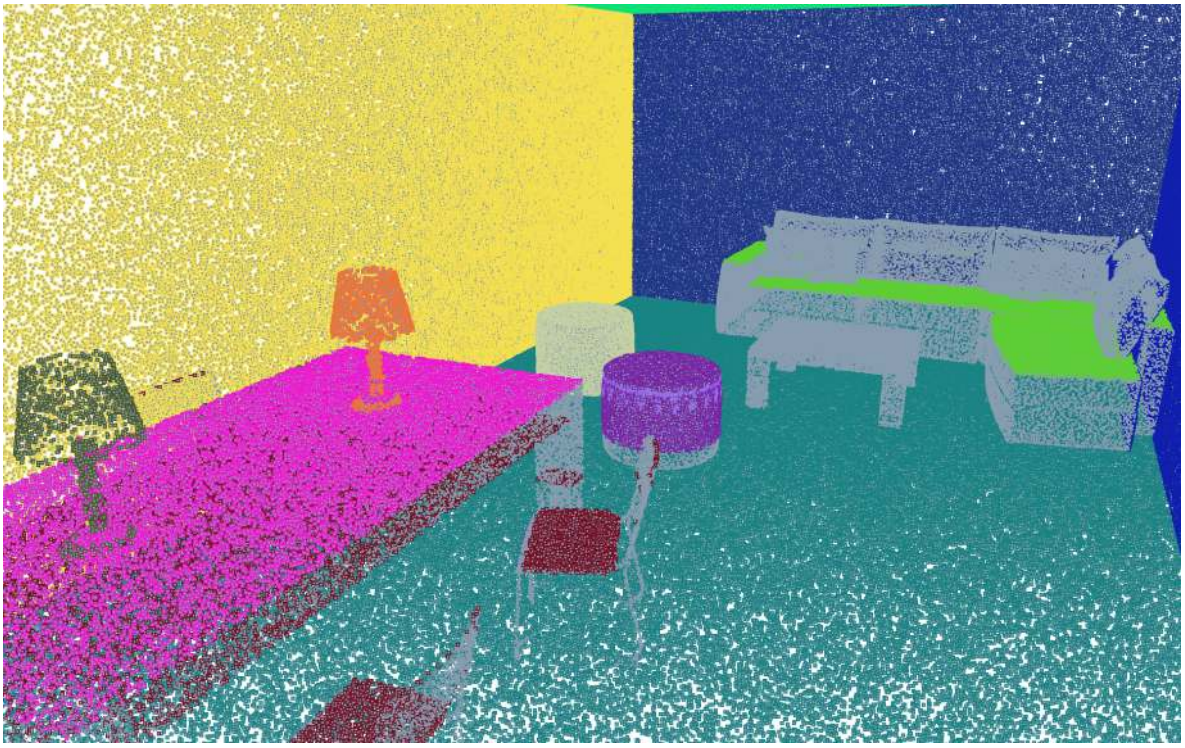


Figure 2.1.: Planar segmentation is part of the annotation process of a point cloud.

2.1. Software for the General Processing of Point Clouds

Data without meaning is not of great use. A proper analysis aiming to extract semantic and geometric properties is usually essential to understand the structure and thus the full capability that a given data set can provide if correctly applied.

Point clouds are usually the product of some kind of optical sensor device which captures a real-world environment by point-sampling it at a regular frequency. Unfortunately, these points are quite raw as they rarely feature more than pure 3D coordinates. Before being effectively used in the application field, a preprocessing evaluation needs to be accomplished. Automation in such processes is increasing as technology progresses but the human component may still play a relevant role in the correct interpretation of particular features. Therefore, some annotation operations need to be carried out manually or at least require human supervision.

Many general-purpose processing tools for mesh and point cloud data exist. This section provides a few example of powerful, but not optimised for pure annotation, tools. Such tools allow in fact to select groups of points but no functionalities for an efficient manipulation of these is available. Nevertheless, they represent good analysis tools for visualisation purposes and data understanding during the initial stages of an annotation process.

2.1.1. MeshLab

Cignoni et al. [12] presented this 3D mesh processing viewer in 2008 with the general purpose to provide advanced research tools to an audience ranging from research and professional frameworks to common users. For this purpose, MeshLab[1] was developed in a way that powerful tools are made available without sacrificing ease of use.

MeshLab is C++ based and supports a wide variety of 3D formats such as PLY, 3DS, OBJ, COLLADA, PTX, X3D and VRML among others. The loaded mesh can intuitively be inspected by means of common mouse inputs such as dragging and clicking. The processing tool has been optimised so that even huge scanning meshes can be efficiently imported and elaborated in real-time. According to a flat approach enabling the layers to be handled together or separately, multiple meshes can be simultaneously visualised in the rendering environment, which is based on OpenGL. Figure 2.3 displays a snapshot of how the MeshLab user interface (UI) may appear upon loading. Since frequent topological and geometry issues in the meshes may need to be detected, various rendering options exist. A given mesh can be rendered using its vertices as single points or triangulated.

Figure 2.2.: a) Visualisation of bounding boxes, b) transparent rendering and c) self intersecting faces automatically detected are examples of inspection tools provided by MeshLab. Own image and image courtesy of [12].

[1]Download and documentation available at `https://www.meshlab.net`. Accessed December 18th, 2020.

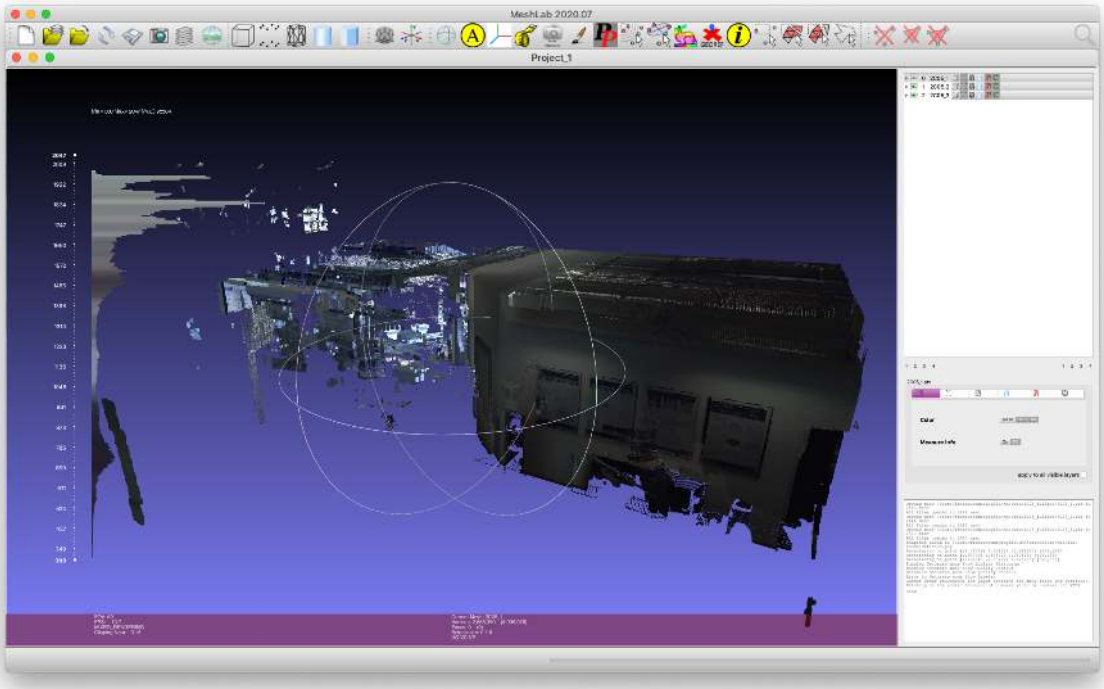


Figure 2.3.: A snapshot of MeshLab after a point cloud has been imported as a collection of PTX files. On the left, one of the provided assessment tools, the histogram, enables the evaluation of the model quality.

Wireframe and bounding boxes are easily displayed as well as geometric properties such as normals. An incoherent orientation of the last can thus immediately be recognised and corrected by means of flipping filters for example. Meshes can also be displayed completely transparent thus revealing possible ghost geometry within the object. Other tools can automatically detect geometrical inconsistencies such as self intersecting faces. Once identified, many of these issues can be interactively solved by mesh cleaning filters to remove non-manifold faces and duplicate or unreferenced vertices Some cleaning filters also automatically fill undesired holes in the model. Examples of inspection tools are visible in Figure 2.2.

Even though the focus is less on the design process of 3D meshes but rather on the editing and processing of them, MeshLab offers many more general-purpose tools than only specific features for the annotation of point clouds. Among these we find sets of remeshing filters targeting mesh simplification and smoothing, or surface reconstruction, many colourisation and inspection filters to enhance edges, curvature and distances from borders, and tools for slicing and various measurements.

Meshes produced by 3D scanner devices, as often the case of point clouds and range maps, are automatically registered into the global frame at loading time and specific tools for cleaning and merging are additionally available. As an example of a practical application in this regard, MeshLab was used to produce clean 3D models from Arc3D data. Arc3D [35] is a web service providing the conversion of photo sequences into 3D raw data. This data has then be processed in MeshLab using the interactive 3D scanning software tools to perform triangulation while managing outliers and noise. Figure 2.4 illustrates an example of 3D mesh reconstructed from photographic data by combining the functionalities of Arc3D and MeshLab.

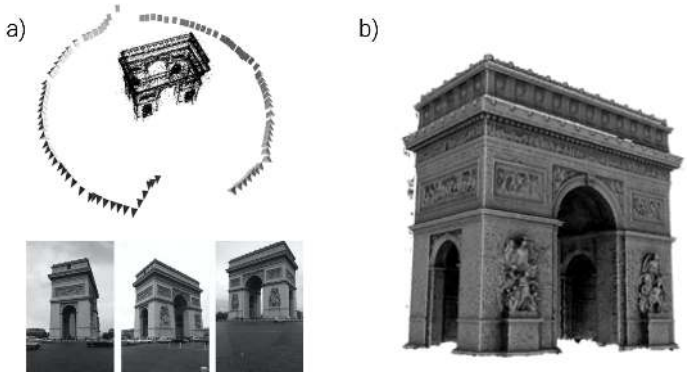


Figure 2.4.: As an example of practical application, MeshLab can be combined with Arc3D to reconstruct 3D raw data a) from a sequence of images taken from different viewpoints around the object and b) eventually obtain a 3D mesh model, in this case, of the Arc de Triomphe in Paris. Image courtesy of [35] and [12].

2.1.2. CloudCompare

The field of emergency mapping served as motivation in 2004 for the creation of a tool named CloudCompare. In particular it was necessary to quickly detect changes in 3D scanning data depicting environments during an emergency so that the evolution in time could be represented on digital or paper maps. These products can be of great help since they efficiently provide support in terms of quick, adapt response and management of rescue resources [17].

The name itself expresses the initial purpose of the tool, which indeed mainly consisted in allowing the comparison between point clouds describing objects or environments at different moments in time. Hence, CloudCompare [19] was first launched in 2006 for internal use by the French electric utility EDF to support the building and landslide monitoring as well as hydrology hazards and historical preservation. By the time of the last release in 2019 the tool had become completely open source and had gathered users in a variety of fields such as academical remote sensing, geology and archaeology but also surveyors, forensic experts, medical professionals, architects and 3D designers.

CloudCompare is based on C++ and Qt, and supports various data formats including PLY, LAS, PTX, PCD, OBJ, STL, SHP, DXF, GeoTIFF for raster data, and TLS among others. The loaded data set can then be manipulated and processed by means of the provided interactive tools for transformation, cross section and segment by detecting connected components and extracting the profile. Visualisation can be further improved by converting and mapping the colours and by displaying local geometric features such as normal vectors. Other mesh operations similar to the ones provided by MeshLab are available and comprehend the mesh creation from a point set by means of a 2.5D Delaunay technique, smoothing, cleaning, point picking and other measurement tools. An example of loaded model in CloudCompare with the result of point picking is illustrated in Figure 2.5.

CloudCompare also features analysis tools as 2.5D volume estimation, the computation of geometric properties including roughness, curvature and density [22], and the fitting of plane, sphere, quadric and other 3D shapes. The possibility to rasterise a given data set is also provided as well as some raster operations such as the computation of contour plots. Point clouds can also be processed based on scalar fields in which each point is represented by a colour value, statically or dynamically, mapped to distance, intensity, density, roughness, confidence, curvature, temperature, time or any other feature. Values within a scalar field can be mixed, spatially filtered, interpolated, imported and exported as coordinate values or be statistically analysed.

Figure 2.5.: A snapshot of CloudCompare after a point cloud has been imported as a PLY file. The little window appears to show the content of a point just picked.

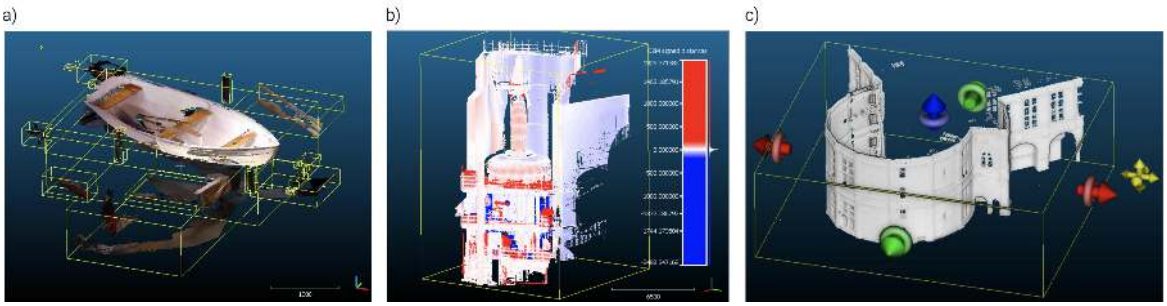


Figure 2.6.: Among the tools provided by CloudCompare we can find a) the segmentation of a model in connected components, b) the computation distances with relative colour-encoding and c) the possibility to transform and cross-slice a model by simply moving the displayed arrows. Image courtesy of [19].

The support of scalar fields facilitate operations of segmentation and subsampling on the point clouds. Applications of some of these tools are reported in Figure 2.6. Similarly to MeshLab, CloudCompare also enables the registering of point cloud scans that are internally handled as octrees, structured grids or full waveforms. Tools for the advanced processing of such data embodies the measurement of various intra-objects distances (*e.g.,* cloud-to-cloud or cloud-to-mesh), the classification with CANUPO [8], Cloth Simulation Filter [41] and a compass mode to interpret and analyse the structural geology of geophysical models [36].

CloudCompare is further extendable with other existing or own developed plugins.

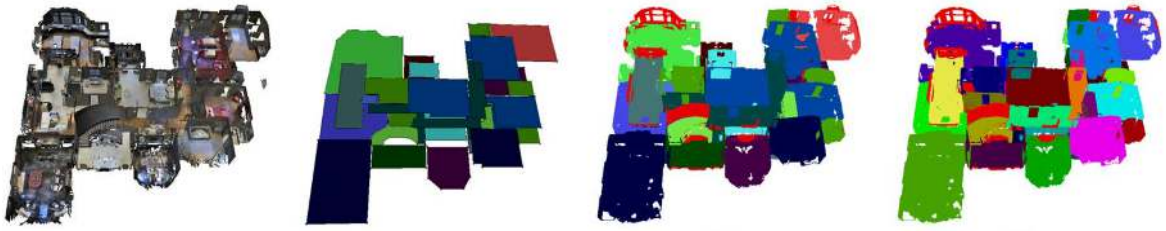


Figure 2.7.: The first step in the Matterport3D annotation process consists in classifying the regions representing a building's rooms. Image courtesy of [10].

2.2. Specific Annotation Tools for Point Clouds

Besides being very powerful tools, performing accurate annotation on point clouds can be very demanding and time-consuming. In fact, the processing functionalities provided by tools similar to the ones just described are not optimised to work on subparts of a given model. In order to annotate a selected set of points in MeshLab, for example, users should first export the considered set as a separate mesh and then load it again in the tool. Once imported, the points of interests can be further analysed by performing fitting or analogous operations. Every time the user needs to process a subsample of the considered cloud, this procedure must be repeated. It is thus clear that the global overview on the annotation is quickly lost. Moreover, the only option to assign a label to the points is by naming the file accordingly, which is also quite unpractical.

Given the above-mentioned limitations, it is common practice to overcome the lack of suitable tools by developing individual ad-hoc solutions. Research groups working in the field of point clouds and especially dealing with deep-learning techniques usually produce an annotation tool in which the proposed method or data can be applied and elaborated.

A good example of specific-purpose annotation tool is developed by Chang et al. [10] to manipulate their data set for indoor environments, Matterport3D, that will be discussed next.

2.2.1. Matterport3D: a large-scale RGB-D data set for indoor environments

The idea of Matterport3D [10] was born in response to the lack of a substantial data set of RGB-D images to train 3D models and solve many vision tasks in application fields such as computer vision including augmented reality and perception assistance, graphics and robotics. Therefore, the group of researchers collected 90 building-scale scenes consisting of globally aligned RGB-D images, the relative raw depth and HDR images, as well as semantic annotations in 3D. The indoor environments were captured with a Matterport camera[2] mounted on a tripod which uniformly sampled the environment by slightly rotating along the gravity direction, from top to down. For each viewpoint, this is repeated towards multiple directions to produce a complete 360° panoramic view. This stationary mechanism avoids the presence of motion blur within the captured images and thus enables highly precise analysis even of small features. The Matterport camera is provided with sensors for the three colour components RGB and one for the depth. The captured buildings are not only academic buildings as most data sets usually feature, but mainly consist of private homes, and can thus contribute to achieve significant knowledge for scene understanding, virtual reality, elderly assistance and home robotics applications.

[2]Official webpage at `https://matterport.com`. Accessed December 24th, 2020.

Figure 2.8.: As part of the annotation process within Matterport3D, the instance objects inside a room are assigned to categories and respectively coloured. Image courtesy of [10].

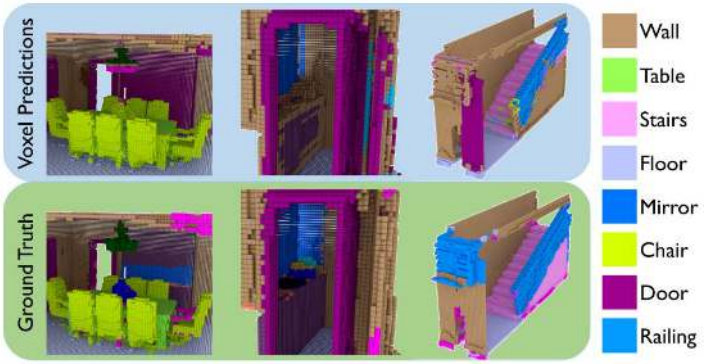


Figure 2.9.: The results of semantic voxel labelling on a test scene from Matterport3D. Image courtesy of [10].

The semantic annotation is facilitated by the global registration of the images which are thus combined to build a surface mesh later investigated to detect objects, regions and labels. Deep, semantic segmentation networks may then be trained using the images featuring the 2D projected labels. Additionally, tasks such as keypoint matching to determine correlations between image data, view overlap prediction and surface normal prediction from colour values can be performed on the various aligned panoramic views of a building.

The annotation process starts with the classification of the regions composing each floor level of the considered building. As Figure 2.7 illustrates, each region corresponds to a room in the building, which is annotated manually. From this planar annotations, 2D volumes to approximate the room volume are then extruded up to the ceiling. The last step of the annotation consists in applying accurate labels to all the 3D objects within each room.

Matterport3D [10] could further provide support for other semantic scene understanding tasks. An example is represented by the technique of semantic voxel labelling which aims to predict a meaningful label for each voxel in the 3D environment. As Figure 2.9 depicts, the data is first densely voxelised into a grid, each voxel of which is associated to a semantic category. Random subvolumes are then evaluated against their occupancy rate. If the percentage of validly annotated occupied volume is too low, the subvolume is rejected. This is just an example on how the Matterport3D [10] data set can be effectively used to perform annotation tasks, whose results can later be used for testing and training other scenes.

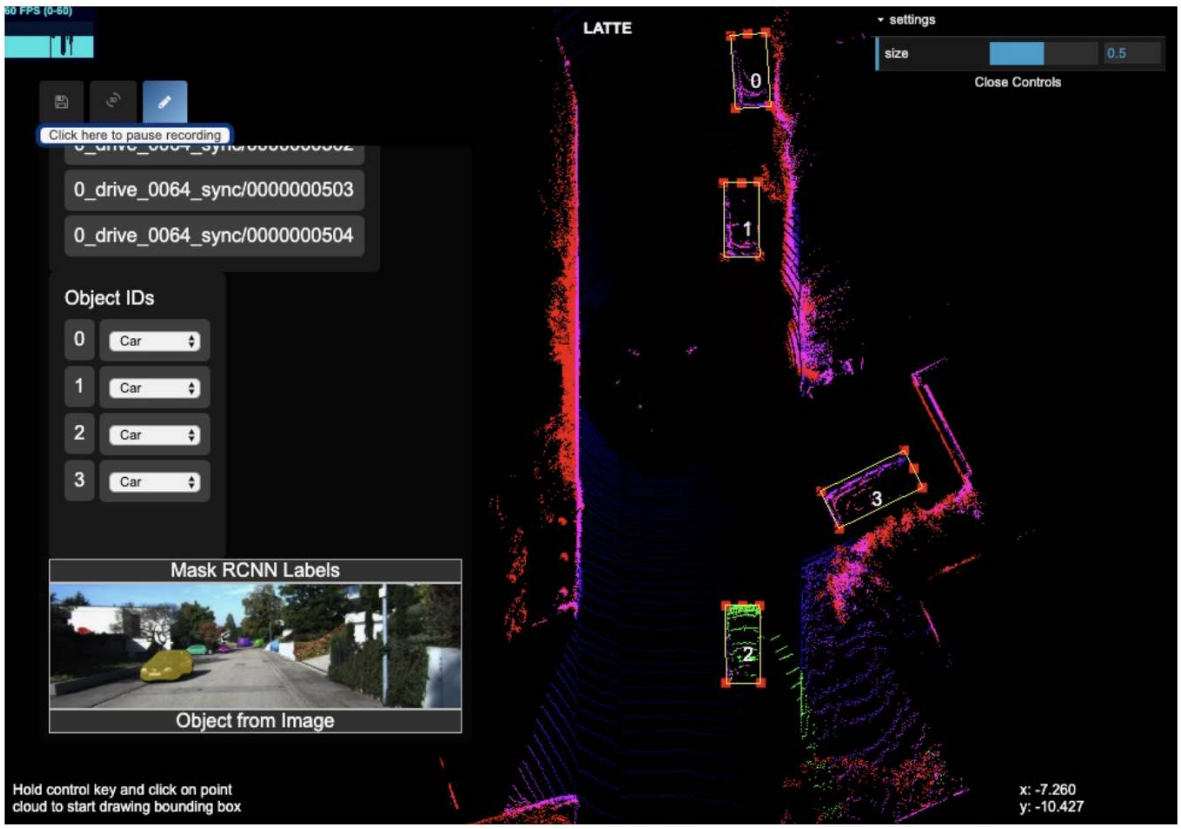


Figure 2.10.: The interface of the annotation tool for LiDAR point cloud data, LATTE. Image courtesy of [38].

2.3. Software for the General Annotation of Point Clouds

If any research group dealing with deep learning in the domain of large point cloud data sets would have to develop their own annotation tool, not only this would represent a huge overburdening but it would also lead to a huge waste of resources. On this basis, many researches have committed to find general solutions. Following, some of the general-purpose tool for the annotation of point clouds are presented.

2.3.1. LATTE

In order to solve the issues related to the annotation of LiDAR point clouds, Wang et al. [38] developed LATTE, an open-sourced tool based on sensor fusion, one-click annotation and tracking to accelerate the entire process. The project was motivated by the increasing exploit of deep-learning techniques to solve LiDAR-based detection of a given environment, especially for applications in the context of autonomous driving vehicles. The interface of LATTE can be seen in Figure 2.10. Instead of uniquely rely on raw LiDAR data, Wang et al. [38] utilise information from the camera usually paired with LiDAR sensors.

This fusion allows for higher resolution – cameras' distinctive feature – and for the application of more technically advanced image-based detection algorithms to automatically assign labels that are later transferred to the point cloud. The second innovation introduced by Wang et al. [38] is

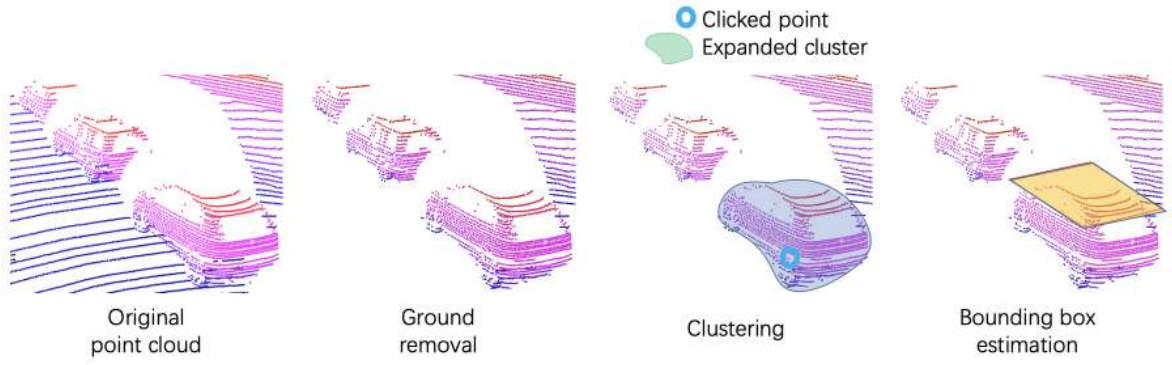


Figure 2.11.: The one-click annotation pipeline of LATTE. A clustering algorithm on the model without the ground-points collects the points around the clicked point to form the entire object. Eventually, a 2D bounding box of the object as seen from a top-view perspective is estimated. Image courtesy of [38].

represented by a technique that reduces the complexity of an annotation process to one single click on the target object.

As a result of this simple operation, a more intricate pipeline is executed to eventually estimate a 2D bounding box around the object as seen from a top-view perspective. The corresponding pipeline is depicted in Figure 2.11. In order to correctly perform the one-click annotation, the neighbouring points fitting to the ground plane are removed.

After a user click, the entire object is obtained by applying a density-based clustering algorithm with noise (DBSCAN) [15] to gather all the belonging points surrounding the clicked point. Finally, the 2D bounding box is estimated using a search-based rectangle fitting [42]. The annotation efficiency is further accelerated on sequences of frames by the integration of a tracking system based on the Kalman filtering proposed by Bishop et al. [5]. According to this technique, the centre of a previously created bounding box is predicted from the previous frame so that the label associated to it can be transferred between sequential data frames without requiring the user to repeat the semantic annotation.

The accurate and fast performance of LATTE have been proven by experimenting with human annotators working on the KITTI data set [18].

2.3.2. SAnE

The Smart Annotation and Evaluation tool [4] is the most recent among the tools presented in this chapter and aims to solve some of the issue related to LATTE by proposing a semi-automatic annotation tool for point cloud labelling that is conveniently usable by both experienced and common annotators. The open-source tool is based on two techniques: a one-click annotation based on a denoising pointwise segmentation strategy and a guided-tracking algorithm.

According to Arief et al. [4], since the one-click annotation developed by Wang et al. [38] is based on a noisy cluster, the predicted bounding box may result inaccurate. For this reason, despite taking inspiration from LATTE's implementation, Arief et al. [4] opted for a noise-free segmentation technique, which simplifies and increases the speed of the one-click annotation, and additionally does not require the prior ground removal as in LATTE. Their denoising method works by assigning penalties, which are stronger near ground and objects' boundary areas. The values of the penalties affect the behaviour of the deep-learning model which thus is less prone to wrong predictions.

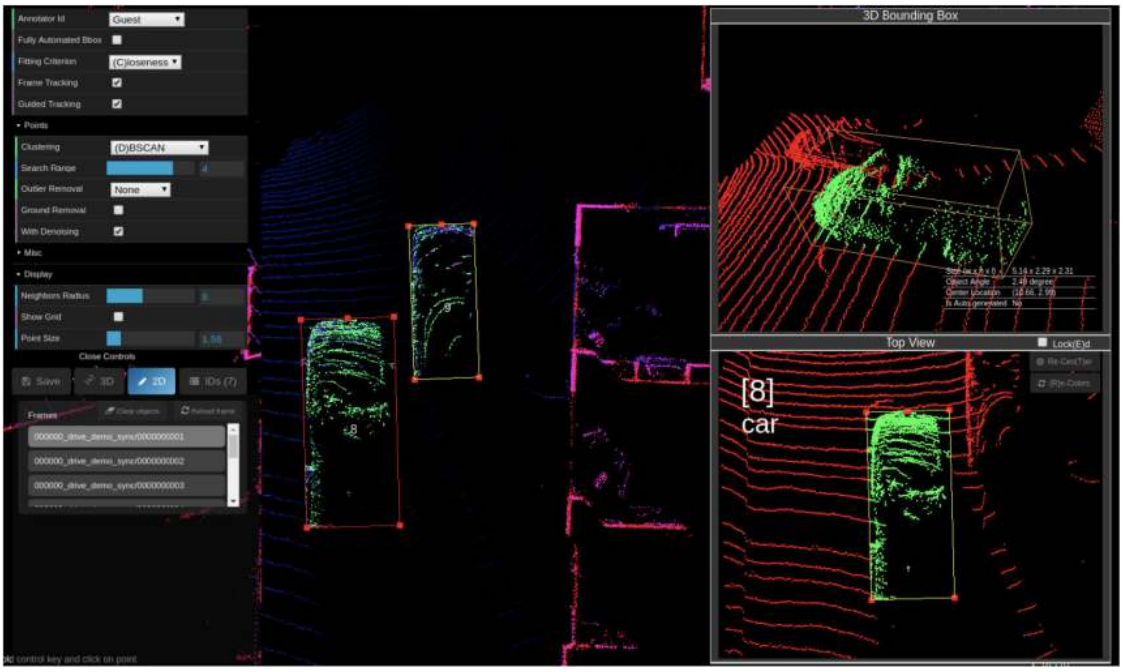


Figure 2.12.: A snapshot during the use of SAnE to annotate point cloud data. Image courtesy of [4].

Furthermore, the rather slow DBSCAN algorithm in LATTE is substituted by a region growing technique based on a nearest neighbour search.

Following the idea of LATTE to boost the annotation of sequential data frames, Arief et al. [4] expanded the motion model technique by developing their own guided-tracking algorithhm based on greedy search and a backtracking to link the frames. However, the tracking algorithm can be changed to match the annotators' will.

In order to facilitate the usability and the interaction so that a wider range of users can be targeted, SAnE complements the previously mentioned artificial intelligence (AI)-based functionalities with UI-based features such as multi-user environments, user-adjusted parameters, side-view refinements for the bounding boxes, keyboard-only annotation by means of predefined hotkeys, object recolouring to enhance the contrast of points belonging to a selected bounding box, and more.

Crowdsourcing experiments on the same KITTI data set [18] used by LATTE were able to show the robustness of SAnE in providing highly accurate annotation labels in a fast and efficient manner while also being interactively easy to use, which thus makes it a competitive alternative to LATTE.

2.3.3. IGRA Group Annotation Tool

In the context of general-purpose annotation tools, Boyko [7] proposes the design of three interfaces for object labelling in large point clouds from LiDAR scans.

The first method is completely user-guided (UG) and allows the user to navigate the displayed point cloud and select the objects to annotate. Upon selection, a label is predicted by a machine learning algorithm so that the user can evaluate the accuracy of the proposed label. When correct, the user simply accepts the label for the selected group. Such mechanism accelerates the annotation process. A prediction example is depicted in Figure 2.13.b). Additional functionalities include group selection and filtering. For example, label-based filtering of objects is ensured by the constant online update of the predictions.

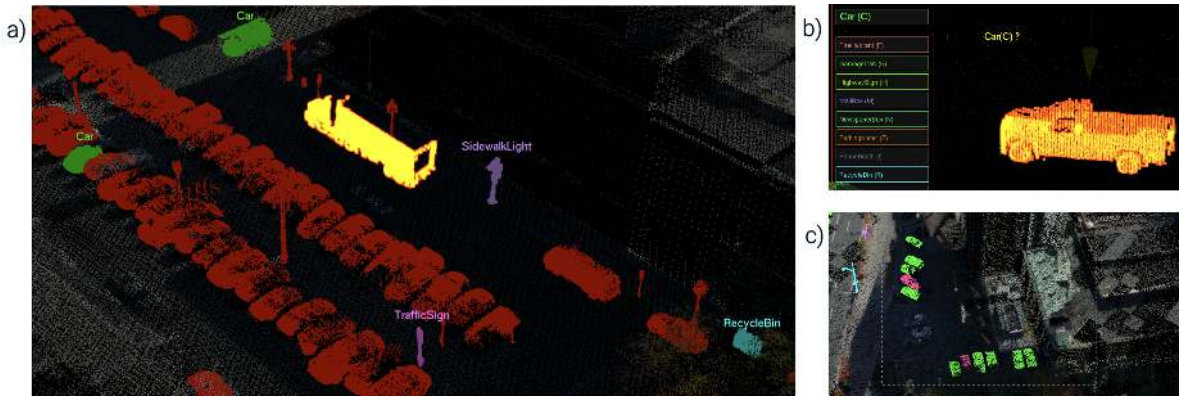


Figure 2.13.: The user-guided interface proposed by Boyko [7]. a) Selected objects are displayed in yellow, while labelled objects with the assigned colour and label. b) A label prediction is suggested to the user, and c) the dashed rectangle represents the region selection of multiple objects. Image courtesy of [7].

In practice, a label can be assigned to a clicked object by either a UI button or more rapidly by shortcut. The point cloud is visualised with a Level of Details (LOD) system, that adjusts the resolution based on the distance from the currently considered area in the model. Zooming closely to desired object is thus possible, enabling a fine-scale object selection while still keeping context awareness of the global scene. Other than selecting single objects the annotator is also provided with a group selection tool in the form of a rectangular region that will include the objects to select, see Figure 2.13.c).

To summarise the UG interface in one sentence, the user is in full control of the annotation process, while, however, being provided with support tools to increase the efficiency. Figure 2.13.a) shows the UG interface featuring labelled objects, a selected object yet to annotate and other objects still to be processed.

In the second interface tasks not completely related to annotation, such as navigation and selection, are removed from the user's control to be automatically performed by means of active learning. It is justified assuming that each object in the scene must be eventually annotated. Therefore, it is possible to reduce the user's active contribution to the confirmation of a given label prediction only, while the objects' selection, and proper visualisation, are left to the machine. From this working principle the name Active Learning (AL) interface is derived. AL provides not only the advantage of reducing overall time costs, but also avoiding the loss of focus by continuously switching between labelling and analysis tasks, in terms of where to navigate and which object, and thus points, to select next. The complexity if the UI also benefits since many elements associated to tasks, now performed in background, are removed from it. The problem of how to display the object currently selected by the machine to the user, is solved by orbiting the camera around that object. The system of label predictions, and relative confirmation, implemented in UG is kept in AL without modifications. A drawback of AL with respect to UG, consists in being limited to the annotation of a unique object at a time.

In summary, AL delegates any operation to the machine so that the user is left with the single task of providing labels.

As a combination of elements form the first two methods, a third and last interface is presented by Boyko [7], the so-called Group Active (GA) interface. Such interface is able to automatically detect groups of objects, that can be labelled together by the user. This way, both advantages of human

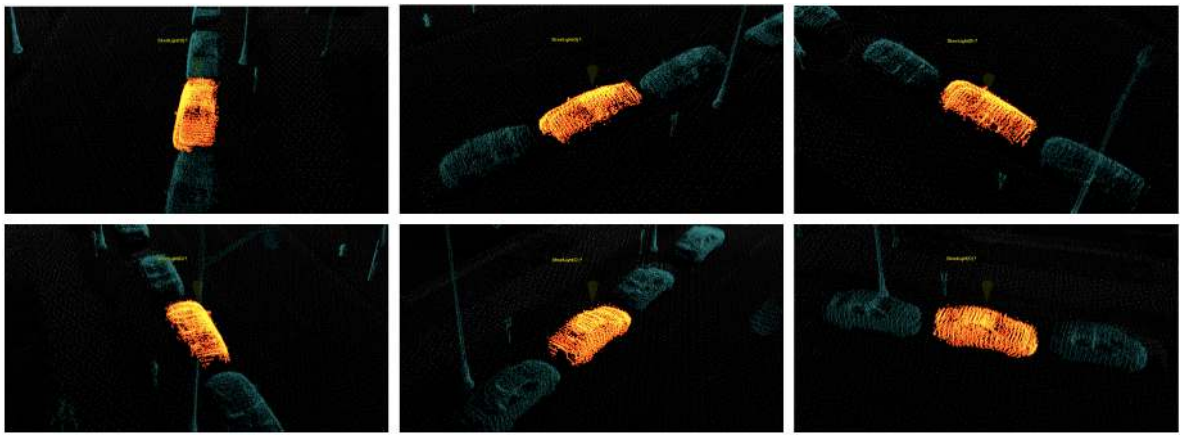


Figure 2.14.: A machine-selected object is orbited in the IGRA's AL interface to allow the user providing the most suitable label. Image courtesy of [7].

capability of better recognising groups of objects instead of single ones and the reduced burdening of work by leaving non-essential tasks to the machine are exploited.

The workflow of GA consists in iteratively show groups of objects to the user with a suggested label that is machine-predicted as likely representing the current group. A this moment the annotator is asked to react by either confirming the suggested label, specifying a new label or rejecting it by asking the system to modify the group in the form of expansion or contraction. The process is optimised also in terms of UI interaction. In fact, the previously mentioned actions correspond to a space bar hit, a selection from the label menu or typing of a new label, and right-arrow or left-arrow key hit, respectively. As the user proceeds with the annotation, the system gathers useful metrics so that the accuracy of group suggestions increases as the number of labelled groups increase. The groups are iterated until all objects in the point cloud have been assigned to a label.

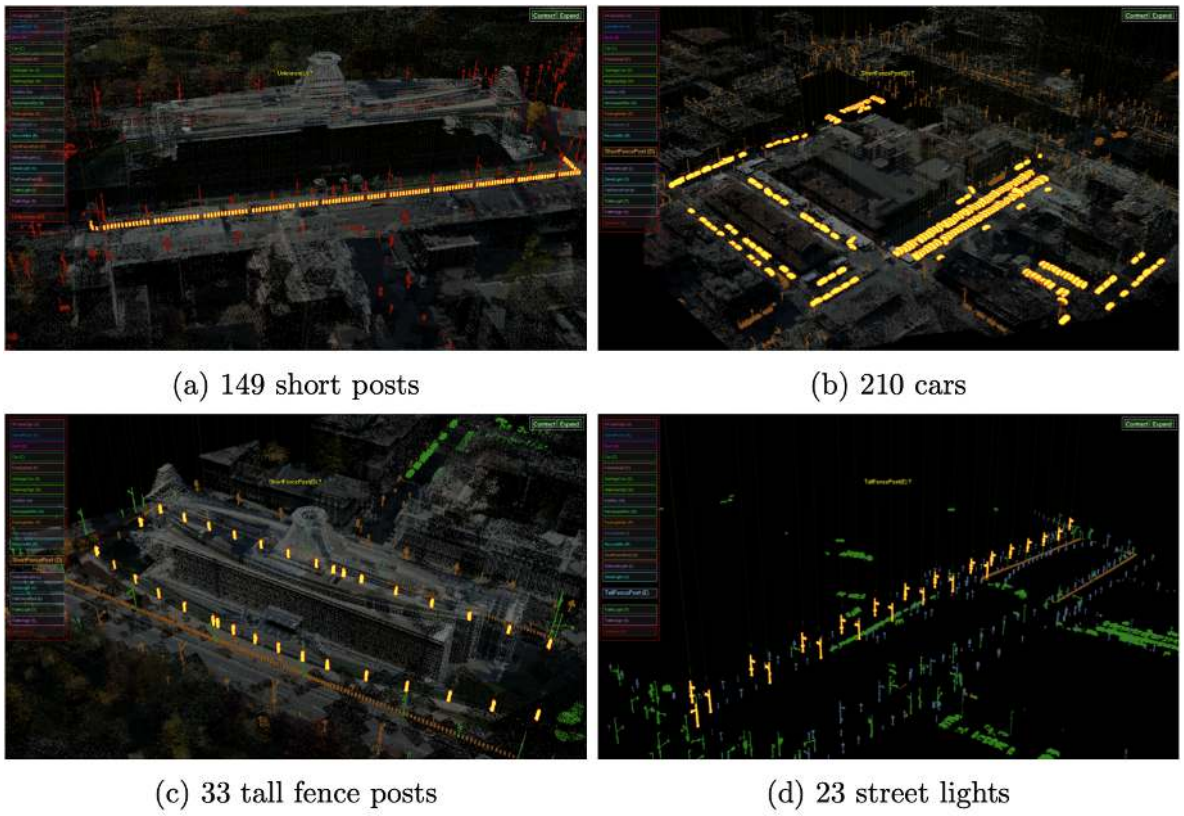
The detection of groups is internally done by means of a tree structure which is traversed to pick the node having highest objective function $B(G, L)$ value and which thus corresponds to the currently selected groups that the user will be asked to label. The objective function for a group of objects, G, and a label, L, is defined as follows:

$$B(G, L) = P_{Label}(G)T_{1\times1}(G, L) - T_{Group}(G, L)$$

where $T_{Group}(G, L)$ is the expected response time from the user, $P_{Label}(G)$ is the expected probability of the user accepting G, and $T_{1\times1}(G, L)$ represents the expected time required to label all the objects in the data set individually. This function aims to estimate the expected time benefit of annotating groups of objects over a single handling of the objects.

In a few words, the GA interface features a mostly machine-controlled system which is however supervised by the data perception of the human annotator.

Human user studies on a large LiDAR point cloud of pre-segmented objects, served to empirically evaluate the three interfaces, which eventually proved to be 1.7 times faster than other standing methods without in turn loosing accuracy.



(a) 149 short posts

(b) 210 cars

(c) 33 tall fence posts

(d) 23 street lights

Figure 2.15.: The GA interface suggests groups of objects, in yellow, the user can label or reject. Image courtesy of [7].

3. Technical Background

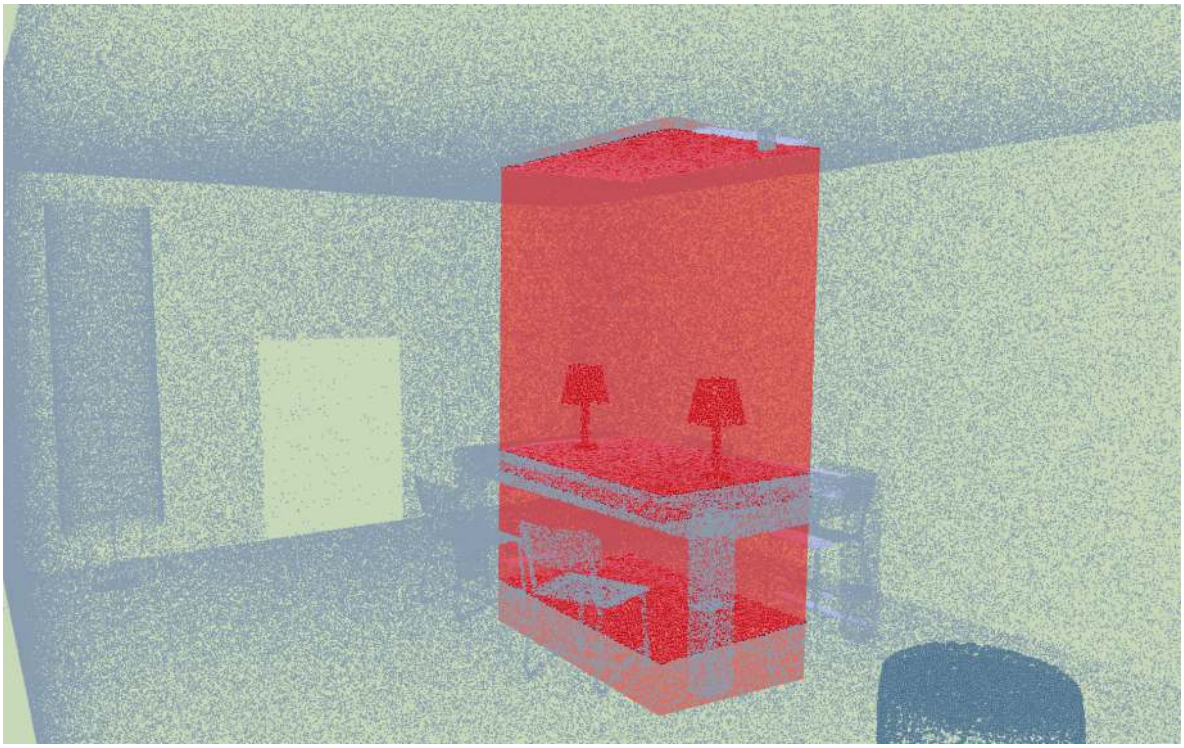


Figure 3.1.: The intersection of 3D shapes with point cloud models allows the selection of multiple points.

3.1. 3D Mouse Picking

Mouse picking [21] is a technique which allows users to interact with 3D objects in the virtual environment by a simple mouse click. As Figure 3.2 illustrates, when the framebuffer (FBO) implements a Multiple Render Targets (MRT) feature, multiple attachments – in form of textures or render-buffer objects – can be written by the fragment shaders and read at any time. Common contents are the IDs associated to each primitive or to the entire object, the depth of the fragment, and other per-pixel or per-primitive attributes.

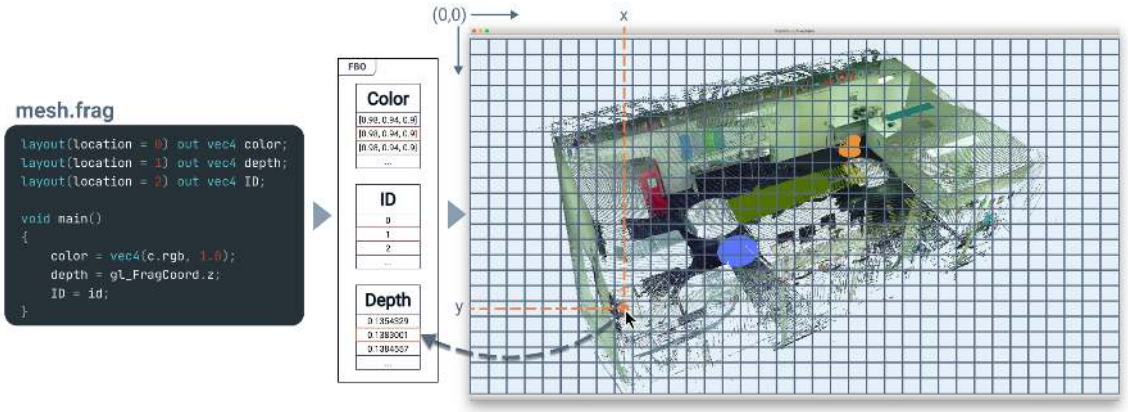


Figure 3.2.: Interaction with virtual objects by mouse clicking is allowed by reading the information associated with the clicked object directly from the FBO using OpenGL specific functions.

3.2. Bounding boxes

Bounding boxes is a convenient way to define 3D objects when collisions among them need to be detected. There are two main categories [20]: *Axis-Aligned Bounding Boxes* (AABB) and *Oriented Bounding Boxes* (OBB). As Figure 3.3 shows, both have the purpose to enclose the object they represent by approximating their volume by excess. As the name already suggests, the faces and edges of AABBs are parallel to the main axes of the considered coordinate system. Therefore, they can be computed by taking the minimum and maximum values for each component x, y, and z, among the vertices forming the bounded object. The OBB of the same object also consists of a bounding parallelepiped, but instead of being axis-aligned, its orientation is arbitrary with respect to the basis frame.

According to Eberly [13], the set of vertices $\{v_k\}$ forming an OBB can be formally defined by a centre C, an orthonormal set of directions $\hat{A}_0$, $\hat{A}_1$, and $\hat{A}_2$, and their extents, a_0, a_1, and a_2 along each of the axes:

$$\{v_k = C + \sum_{i=0}^{2} x_i \hat{A}_i : |x_i| \leq |a_i| \ \forall\, i, k \in [0, 7]\}$$

Eberly [13], also provides some indications about the construction of an OBB from a set of input points. The OBB's axes are defined by the unit-length eigenvectors of the covariance matrix

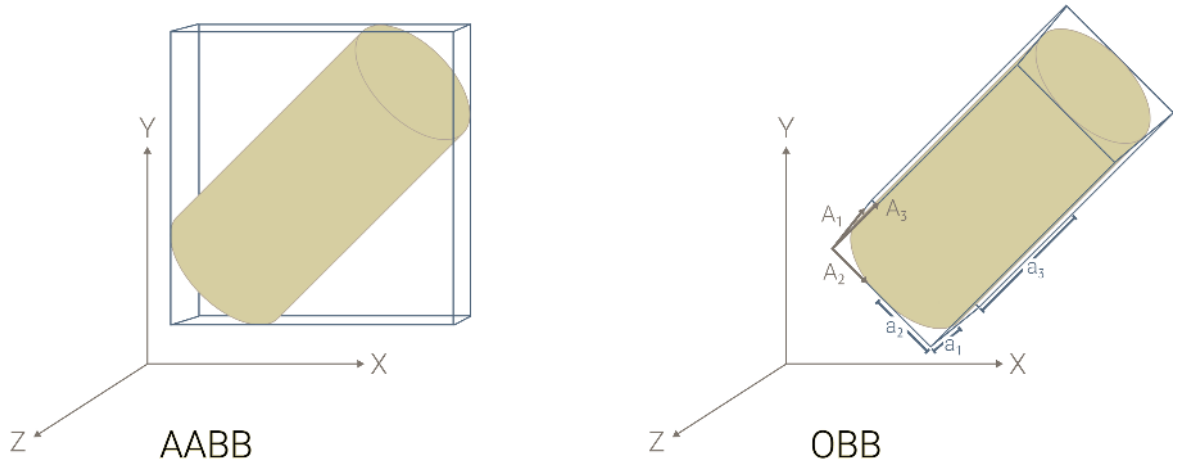


Figure 3.3.: AABB encloses the object with axes aligned to the main frame, while the axes of the OBB are oriented to minimise the volume.

$$M_C = \frac{1}{k}\sum_{i=1}^{k}(p_j - C)(p_j - C)^\top \mid j \in \{0, 1, 2\}$$

where k is the amount of points to consider, in this case eight. The extents along these axes are given by

$$max_j \left| \hat{A}_i \cdot (p_j - C) \right|$$

Given their arbitrary nature, intersections involving OBBs are less straightforward to handle than when dealing with AABBs only. As Huynh [23] suggests, the separating axis theorem can be exploited to detect collisions between two OBBs. According to said theorem, if a separating axis exists, then two OBBs do not intersect. Cases for all faces and edges need to be considered resulting in a total of 15 inequalities: six for the basis axes and nine for the cross-products among these axes. If and only if all cases are false, *i.e.,* no separating axis exists, the inspected OBBs are guaranteed to intersect. Details about the inequalities can be found in Huynh [23] and in its derived implementation in the `MathHelper` class.

Beside intersections between boxes, it may be necessary testing for points-in-OBB collisions, for example, when intersecting a 3D object – such as an orthogonal frustum – with an octree structure storing a point cloud. As Listing 3.1 shows, when a leaf is reached, the points it contains must be tested against such type of collision. In order for a point p to be contained in an OBB, defined by the above-mentioned parameters, the following condition must be satisfied:

$$\left| \hat{A}_i \cdot (p - C) \right| \leq a_i$$

The projection of p over each basis axis $\hat{A}_i$ expresses the distance of the point from C. For all three axes, this distance must be smaller than the axis' extent a_i.

3.3. Octree Traversal

One of the advantages offered by octree spatial structure is the efficient means to access sub-partitions of objects' volume. Its hierarchical structure, indeed, optimises operations such as collision detection.

The following considerations refer to the octree structure as it is provided by the Point Cloud library (PCL), see Section 5.3. As presented in Figure 3.4, each node in the octree is a voxel, which has at most eight more children. The entire structure can be easily traversed – either in depth- or breadth-first order – by means of a specific type of iterator, provided by the *octree* module. Other methods enable the recognition of the currently considered node's depth and type (*i.e.,* leaf or internal node), as well as identifying the boundaries of the voxel, which is actually an AABB. The implementation of a depth-first traversal algorithm to find the points contained by a frustum shape is illustrated in Listing 3.1. As per standard procedure, each node is handled differently based on its type. Generally, if a node intersects the frustum, then the search for collisions must be refined by inspecting its children, otherwise traversing its subtree is skipped as surely no child-node intersects with the frustum. More specifically, the boundaries of internal nodes intersect with the frustum according to the procedure introduced in Section 3.2. Since AABBs are regular versions of OBBs, the general algorithm to detect collisions between OBBs can be applied.

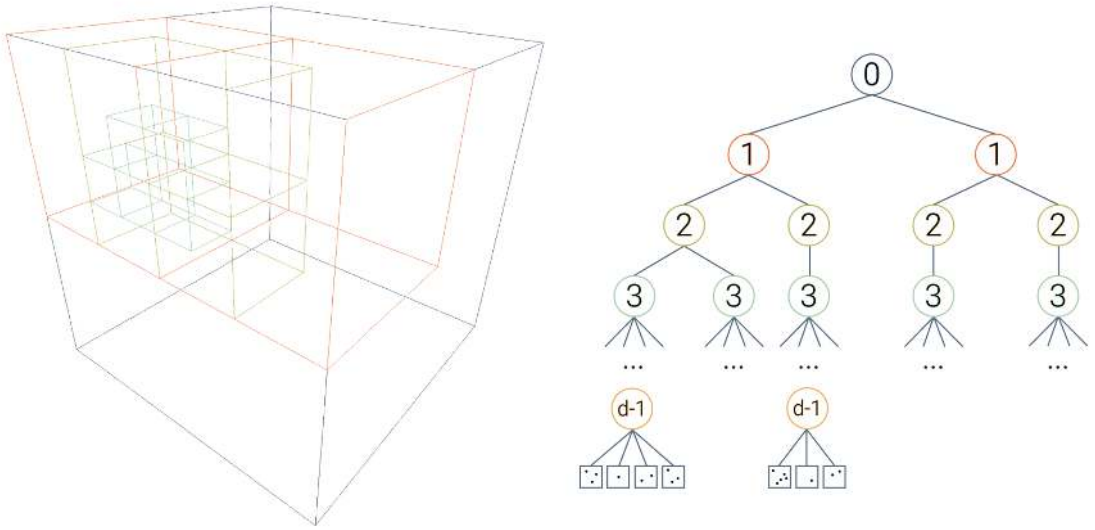


Figure 3.4.: An octree is a spatial acceleration structure, which hierarchically divides 3D space into voxels. A common practice consists in storing actual data in the leaf-nodes located at a depth d from the root-node.

The variable `levelToSkip` in Listing 3.1 indicates the depth level from which child-nodes should be skipped when traversing the tree. When a node at depth d fails the intersection test, `levelToSkip` is set to $d + 1$, since all its children have larger depths than d. This way, possible siblings – having also depth d – are still visited to check for intersections. In fact, when a node intersects the frustum `levelToSkip` is reset to a depth value that can never be reached by any node. Consequently, all nodes visited next, pass the initial skip-condition until a non-intersecting node is encountered again.

```
1  void MathHelper::findInFrustum(OBB _frustum, std::shared_ptr< OctreeT >& _tree,
2                                 std::vector< int >& _result)
3  {
4      /* Variable to control the testing procedures.
5       * Has value equals to one depth more of leaves' depth, i.e. no node can reach it */
6      int unreachableLevel = _tree->getTreeDepth() + 1;
7      int levelToSkip = unreachableLevel;
8      int currentLevel = 0;
9
10     // Depth first iteration over entire octree
11     for (auto it = _tree->depth_begin(); it != _tree->depth_end(); ++it) {
12
13         // Skip the children of non-intersecting nodes
14         currentLevel = it.getCurrentOctreeDepth();
15         if (currentLevel >= levelToSkip) { continue; }
16
17         if (it.isLeafNode()) {
18             // Get point indices in leaf
19             std::vector<int> leafIdx;
20             const pcl::octree::OctreeNode* node = it.getCurrentOctreeNode();
21             auto* l = dynamic_cast <const OctreeT::LeafNode*>(node);
22             (*l)->getPointIndices(leafIdx);
23
24             // Intersect points in leaf with OBB frustum
25             for (auto idx: leafIdx) {
26                 glm::vec3 p = glm::vec3(_tree->getInputCloud()->points[idx].x,
27                                         _tree->getInputCloud()->points[idx].y,
28                                         _tree->getInputCloud()->points[idx].z);
29
30                 // Take index of each intersecting point and add to result
31                 if (MathHelper::pointInOBB(_frustum, p)) { _result.push_back(idx); }
32             }
33         }
34         else if (it.isBranchNode()) {
35
36             // Get AABB of the current node and compare with frustum
37             Eigen::Vector3f vox_min, vox_max;
38             _tree->getVoxelBounds(it, vox_min, vox_max);
39             AABB bb = std::make_tuple(glm::vec3(vox_min.x(),vox_min.y(),vox_min.z()),
40                                       glm::vec3(vox_max.x(),vox_max.y(),vox_max.z()));
41
42             // AABB (node voxel) vs OBB (frustum)
43             bool intersect = MathHelper::OBBInAABB(bb, _frustum);
44
45             /* If do not intersect set the levelToSkip to the current node's children so
                    that the subtree of the current node
46              * is skipped, but its siblings (i.e. having the same depth)are still
                    checked for intersections
47              */
48             if (!intersect) { levelToSkip = currentLevel + 1; }
49             // If intersects, reset the levelToSkip so that the current node's children
                   are checked for intersections
50             else { levelToSkip = unreachableLevel; }
51         }
52     }
53 }
```

Listing 3.1: Octree traversal to intersect a 3D frustum object. The testing-branch terminates as soon as a parent-node does not intersect with the given frustum. The content of the leaves is tested using a point-in-OBB procedure.

Leaf-nodes contain the actual point data, so when such node is reached, all points held in it are retrieved. Each point is tested as explained in Section 3.2 to verify if it is actually comprised in the

frustum. Upon successful outcome, the ID of the considered point in the leaf is inserted in the resulting set of points to return.

3.4. Segmentation Methods

The operation of segmenting can be applied to 2D images or to 3D objects, such as point clouds. It consists in partitioning the processed object into multiple non-overlapping clusters, based on specific conditions [37]. The result of an automatic segmentation – only controlled by means of some parameters – can be manually refined to obtain a more semantic partition that is thus not purely geometry anymore.

The followings report the contribution of segmentation methods for this thesis.

3.4.1. Region Growing

In order to segment an entire image, multiple initial seeds, usually randomly selected, are needed as starting points for the growth of multiple regions. In the context of this work though, users have the possibility to select points or pixels from the point cloud. Region growing has been chosen to be an effective method for the selection of multiple pixels from the panoramic image associated with the point cloud. Therefore, a single seeded region growing has been implemented on the basis of the technique proposed by Om Prakash *et al.* [37]. The use case of this thesis has the additional advantage of a non-critical selection of the initial seed, since it exactly corresponds to the pixel selected by a user's mouse click.

The method of Om Prakash *et al.* [37] follows the basic principle of imagery region growing technique. It consists in aggregating homogeneous pixels, starting from an initial seed, and iteratively checking the neighbouring pixels against a similarity condition. The stopping criteria determines the similarity degree within the regions. Om Prakash *et al.* [37] adopts the adaptive threshold resulting from Otsu's method [29]. This value determines the optimal segmentation threshold by exploiting the constant relationship between intra-region and inter-region variance.

Despite the stability and effectiveness of such method, a larger degree of freedom is provided in the region growing method in *PointCloudAnnotator* by introducing a dynamic threshold setting, based on mouse motion. More details about this mechanism is discussed in Section 5.5.2. Listing 3.2 illustrates the adapted version of the algorithm proposed by Om Prakash *et al.* [37]. Starting from the initial seed px_s, the first eight neighbours n_i in the $3x3-$matrix (surrounding px_s) are retrieved and colour-compared with the initial seed. If the colour-distance is larger than the threshold, the neighbour is discarded. If instead the distance between the pixels is within the threshold, then the ID of n_i is added to the region and to the queue of next seeds, $seeds$, used to grow the region. The algorithm continues with the next iteration, which evaluates the similarities between the next element in $seeds$ and its eight neighbours. The $processed$ list controls the termination of the algorithm by preventing an already assessed pixel from being used again as seed, causing an infinite loop. When no more seeds are found, the region is complete and can stop growing. Additionally, in the *PointCloudAnnotator* another version of this region growing procedure has been implemented which checks for similarity based on the angle deviation between normal vectors.

```
1  std::vector<int> MathHelper::growRegionColor(const Pmf& _img, glm::vec2 _seed,
2                                               float _dist, RegGrowParams _params)
3  {
4      // Compute threshold based on mouse distance form the initial seed
5      float threshold = MathHelper::mapDistToThreshold
6                                      (0, glm::length(glm::vec2(_img.nCols, _img.nRows)),
7                                      _params.minThreshold, _params.maxThreshold, _dist);
8
9      // Resulting region containing index locations (in considered pmf) of similar pixels
10     std::vector<int> regionIdxs;
11     // UVs of next pixels to grow
12     std::queue<glm::vec2> pixelsToGrow;
13     // Index of the pixels already grown, needed for terminate condition
14     std::vector<bool> processed;
15     processed.resize(int(size.x * size.y), false);
16     // Since pixels are stored col-wise, index is: col-ind * height + row_index
17     int u = _seed.x;
18     int v = _seed.y;
19     regionIdxs.push_back(v * _img.nRows + u);
20     pixelsToGrow.push(glm::vec2(u,v));
21
22     // Grow region until there are pixels to grow
23     while (!pixelsToGrow.empty()) {
24
25         // Pick pixel to grow, aka current seed
26         glm::vec2 px = pixelsToGrow.front();
27         pixelsToGrow.pop();
28         // Derive its index for properties' retrieval
29         int ind = px.y * _img.nRows + px.x;
30         // Get seed's colour
31         glm::vec3 growPxCol(_img.pixels[ind].color);
32
33         // Get all neighbours of the pixel to grow (8 at most)
34         std::vector<glm::vec2> neighbours = getNeighbours(_img, px.x, px.y);
35
36         // Test all neighbours against similarity
37         for (size_t j = 0; j < neighbours.size(); ++j) {
38
39             // Get neighbour colour
40             int nIndex = neighbours[j].y * _img.nRows + neighbours[j].x;
41             glm::vec3 n = glm::vec3(_img.pixels[nIndex].color);
42
43             // Compute colour distance
44             double dist = glm::l2Norm(growPxCol, n);
45
46             /* If within threshold, insert not-already-grown neghbour in region
47              * and update list of pixels to grow
48              */
49             if (dist <= threshold) {
50                 // Add to region and as pixel to grow if not already used
51                 if (!processed[nIndex]) {
52                     regionIDs.push_back(nIndex);
53
54                     pixelsToGrow.push(neighbours[j]);
55                     processed[nIndex] = true;
56                 }
57             }
58         }
59     }
60     return regionIdxs;
61 }
```

Listing 3.2: Colour-based region growing inspired by Om Prakash *et al.* [37]. Neighbours of a pixel seed are evaluated based on a similarity condition. Pixels satisfying it are added to the region and processed as next seeds.

3.4.2. 3D Shape Detection

The automatic recognition of 3D shapes can be of great help in the early stages of the annotation of a point cloud. When such purely geometric segmentation is performed, users can manually refine it by adding semantic meaning to the partition. The algorithm for shape detection featured by this thesis is proposed by Schnabel *et al.* [34]. It is based on the RANSAC method, which robustly extracts set of primitive shapes best approximating the given point data from an initial random selection of minimal point sets. Each point in the given point cloud must be associated with a normal vector. It is possible to specify one or multiple types of geometric primitives to extract. Predefined shapes are planes, spheres, cylinders, cones, and tori. The subsequent paragraphs describe the most relevant parts composing each iteration of the shape detection of Schnabel *et al.* [34].

Determination of a candidate shape by means of RANSAC paradigm

Each iteration starts with the search of shape candidates in the form of minimal sets uniquely defining the considered geometric shape. The points in such sets are randomly sampled from the main data set. A score function evaluates the quality of the candidate. The function considers a maximum distance ε between points and shape's surface, and an angle α as upper bound for the deviation between the points' normals inside the ε-band and the shape's normal. In addition, the connectivity of the remaining points is also taken into account, by means of the so-called cluster-ε variable. The set having the highest score is selected as best candidate to fit the user-specified primitive. A probabilistic component is also considered before accepting a candidate as the best one: it must be improbable enough to find a better candidate during the sampling phase. Probability also controls the termination condition of the algorithm: when no better fitting can be found, given a minimal shape size, the set of detected shapes is returned.

Refitting

Before being inserted in the final output, each best candidate shape is refined by applying a least-squares fitting algorithm, producing the final set of inliers. CGAL [27] library (see Section 5.3) implements the algorithm of Schnabel *et al.* [34] and provides a set of predefined shapes to use as input for the segmentation. Each detected shape is associated to parameters which vary depending on the primitive type. A plane is defined by its normal vector $\vec{n} = \{a, b, c\}$ and by a scalar coefficient d. A radius r defines a sphere when associated to a centre point $c = \{x_0, y_0, z_0\}$, a cylinder in combination with an axis $\vec{a} = \{x, y, z\}$, and a cone when the apex $a = \{x_a, y_a, z_a\}$ is defined together with r and $\vec{a}$. A torus can be constructed from an axis $\vec{a} = \{x, y, z\}$, a centre $c = \{x_0, y_0, z_0\}$, and two radii – a minor radius r and a major radius R.

3.5. Normal Estimation

As seen in the previous section, some operations require the input point cloud data to be associated with a normal vector for each point. Usually, scanning devices producing such data set only capture raw points in form of 3D locations and do not provide any other geometric properties. Therefore, point cloud must undergo a process of normal estimation, which can be either performed at loading time or during a preprocessing step.

Normal vectors of a surface are influenced locally. This explains why common approaches involve a Principal Component Analysis (PCA) over a spatial or k-neighbouring subset of points to estimate a tangent plane. The normal $\vec{n}$ of such plane corresponds to the normal of the point being considered.

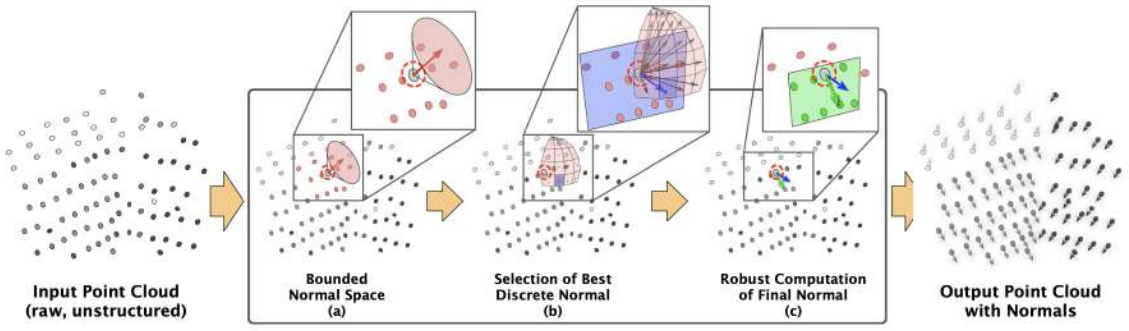


Figure 3.5.: According to the normal estimation method proposed by Mura *et al.* [25], for each point in the input cloud, the solution space for normal is limited by means of PCA on the local neighbourhood (a). Such subregion is then discretised to obtain a set of normal candidates for each patch (b). Among these, the normal having minimal median distance to the neighbourhood is selected as best candidate to use in the last iterative refinement (c). Image courtesy of [25].

Mura *et al.* [25] propose a robust method for the estimation of normals in unstructured 3D point clouds, summarised in the following steps:

1. Limitation of the solution space for the normal.
2. Extraction of the best candidate normal from the spherical discretisation of the limited solution space.
3. Robust iteration to compute the final normal.

A visual representation of the entire process is provided by Figure 3.5. For a point p, the solution space is defined by an upper bound angular deviation α between the normal $\bar{n}$, obtained from the PCA of p's neighbours N_p, and the correct normal. This limitation produces a cone of solutions having axis $\bar{n}$ and aperture angle α. Such angle also defines a sector on a discrete unit sphere, centred at p. The sphere is composed of rectangular patching of constant height, along the latitudinal direction and varying longitudinal width, according to the accumulator design proposed by Borrmann *et al.* [6]. Each patch defines a candidate normal vector starting at p and going through the centre of such patch. For each candidate, it is computed the median of the distances between each neighbour in N_p and the plane defined by that candidate normal and p. The candidate normal being associated to the minimal median is selected as best discrete normal. This initial estimation is then refined in the last iterative step to obtain the final normal for p. Here, a PCA is performed on the set of neighbours lying within the median distance from the plane defined by p and the current normal estimation n_i. The resulting normal n_{i+1} is accepted as new estimation, if the angle between n_i and n_{i+1} is within a given threshold. After three iterations the algorithm terminates.

The method described above has been first implemented for this thesis. However, another approach has been undertaken with the attempt to increase its performance. The algorithm currently adopted in the *PointCloudAnnotator* is based on a manual implementation of the approach used by PCL [3]. For each point p in the input data set, the computation of its associated normal vector is reduced to a least-squares fitting estimation problem. The neighbours within a radius around p form a surface, whose tangent plane's normal well approximates the normal for point p. In order to define such plane, we

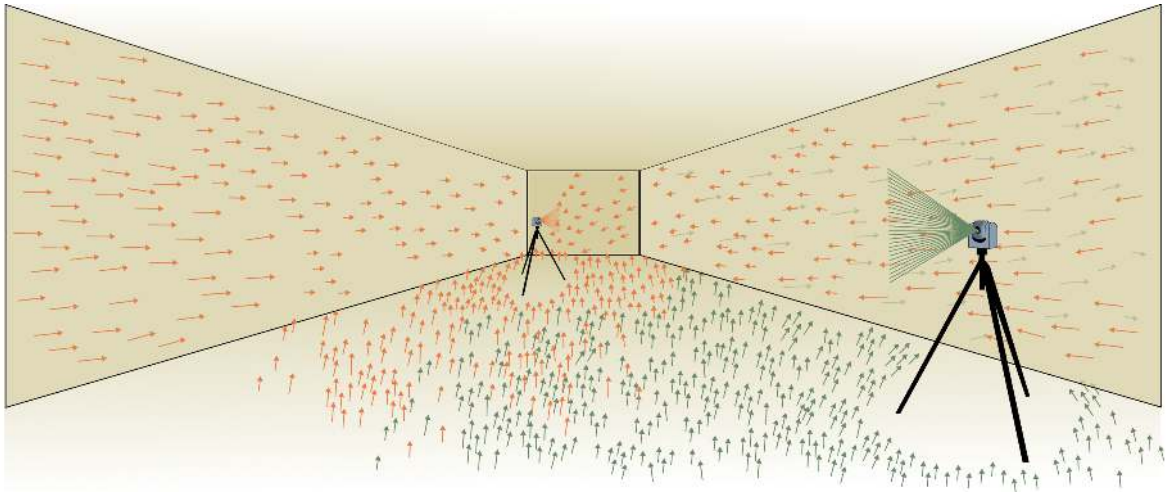


Figure 3.6.: A room can be scanned from multiple viewpoints. Each scan produces a panoramic map of the environment. During *registration* into a global reference frame, the normals must be correctly orientated towards the respective viewpoint, otherwise they may point outside the room, as in the case of the walls in this artificial example.

first need to build a covariance matrix using the 3D centroid of the nearest neighbours, as defined in Section 3.2. According to a PCA approach, the covariance matrix is then decomposed into eigenvalues and eigenvectors, which provide a solution for the surface normal. The final normal of p corresponds to the eigenvector associated to the lowest eigenvalue.

When a point cloud is composed by multiple panoramic depth maps, we need to ensure that the normals are consistently orientated. Each depth map was captured from a specific viewpoint v_p. Thus, normals $\vec{n_i}$ of all points p_i, originally belonging to that map, must be turned towards v_p satisfying Equation (3.1). Figure 3.6 shows this concept. If the angle between $\vec{n_i}$ and the view direction is larger than $90°$, the direction of $\vec{n_i}$ is inverted.

$$\vec{n_i} \cdot (v_p - p_i) > 0 \tag{3.1}$$

Eventually, the resulting point cloud presents normals correctly oriented towards the corresponding 3D point in which the scanner was located during the measurements. A before-and-after comparison is illustrated in Figure 3.7.

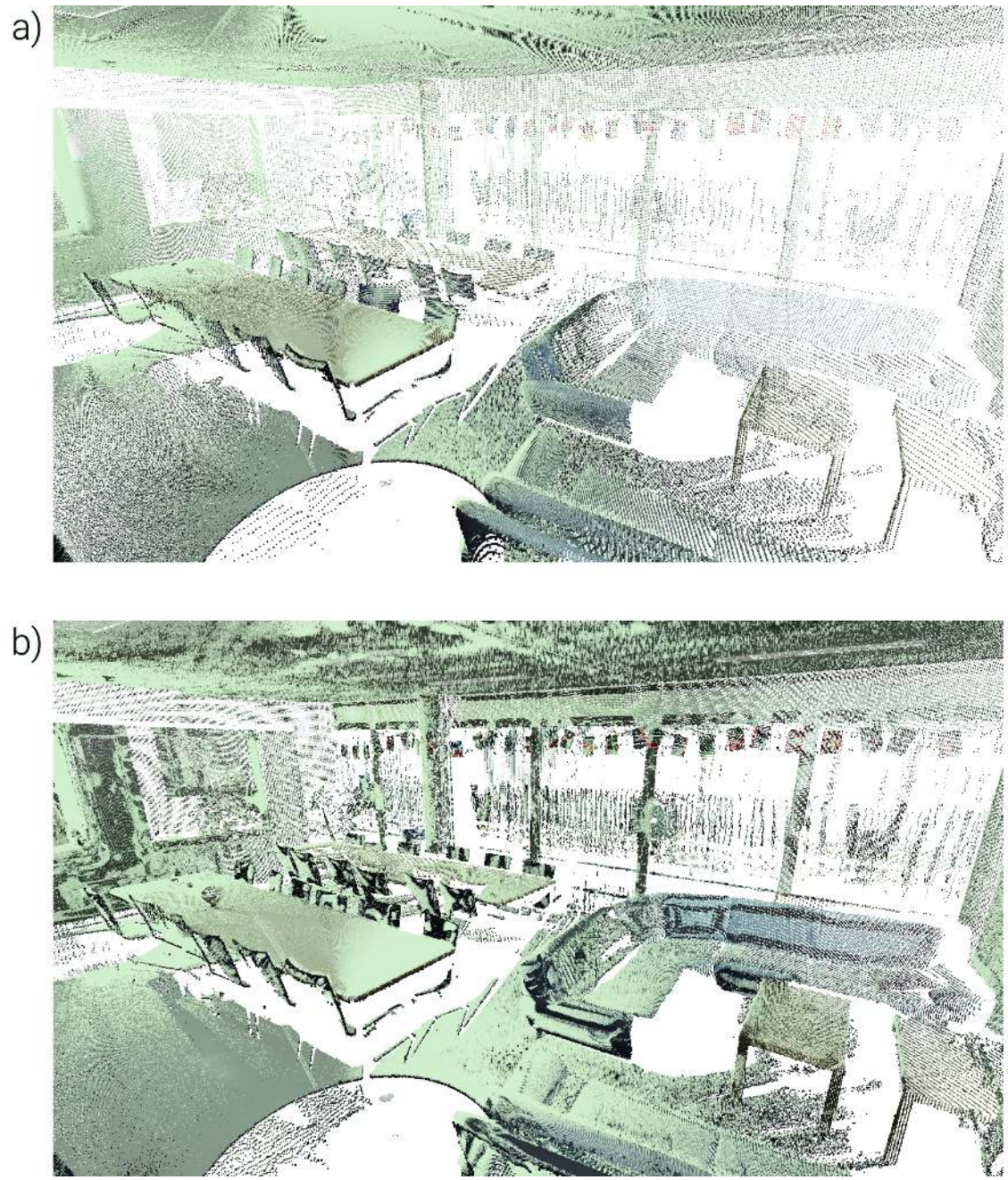


Figure 3.7.: a) Normals correctly oriented towards the respective viewpoint. b) Without such correction the normals are not consistently oriented and most of them point towards the wrong direction.

4. Data

Figure 4.1.: Three point cloud data sets of two rooms and one apartment were used in the development of the *PointCloudAnnotator*.

4.1. 3D Point Clouds

Indoor environments, ranging from single rooms to entire apartments and even multi-storey buildings, can be conveniently represented by a collection of 3D measurements known as point cloud data set. Laser scanners and RGB-D cameras are some of the various acquisition devices to produce such accurate and densely measurements. As technology progresses, they are becoming increasingly affordable and accessible to ordinary users [24]. This trend is further endorsed by a news of just few weeks ago regarding the technology company Apple, which presented new mobile devices featuring an integrated LiDAR scanner [2]. It is stated that such feature was mainly introduced to improve photo camera capabilities and Augmented Reality (AR) experiences, thus is far from reaching the standard quality of professional scanners [26]. Yet it demonstrates well how 3D capturing technology is becoming easily available.

Terrestrial and airborne laser scanners are commonly used to map large outdoor environments, such as portions of the Earth's surface, exploiting an optical remote-sensing technique called Light Detection And Ranging (LiDAR) [14]. Since the 2000s, swisstopo has been gathering data using LiDAR sensors to produce digital terrain models and other products describing the Swiss landscape [28]. Nevertheless, the application of LiDAR technology has been recently extended to the mapping of interior environments as required by services or building management sectors to improve navigation by means of interactive maps, security, or even energy management [31], as well as by expanding fields such as AR and Virtual Reality (VR) in the entertainment industry.

The principle of LiDAR is the same for both terrestrial and airborne scanning, as shown in Figure 4.2. The difference lies in the location of the sensor over the time of capturing. While terrestrial acquisition ca be made either in a static or dynamic manner, aerial ones are usually performed in motion. In order to map limited indoor environments, such as a single room, it is usually sufficient to scatter light rays from a static position along predefined directions, so that the distance between the sensor and the first real-world surface encountered by each ray can be measured. Having precise knowledge of the measurement's origin point, each distance can be converted into a 3D point along the direction of the ray that produced it, resulting in a 3D point cloud representing the surveyed environment [24].

Differently from images, where data is organised in dense grids providing some sort of topological information [14], point clouds store geometric data without any particular order. Therefore, they are defined as unstructured data sets [24].

Despite the lack of structure, many 3D scanners typically output a data format which allows the reconstruction of the surveyed real-world scene. Such format can consist of one single file containing enough samples to fully represent the captured subject, not requiring further preprocessing. If, however, the realism degree needs to be increased or if a single acquisition cannot capture the entire complexity of an environment, then multiple scan phases must be performed from various viewpoints [24]. The resulting scans can then be merged undergoing a process, often referred to as *registration*.

The next section describes two of the said formats in more detail, each representing one of the output categories.

4.2. Input Formats

The *PointCloudAnnotator* accepts point clouds from three different sources. Two of these are common formats, namely Polygon File Format (PLY) and PTX, and need to be converted into a custom format at loading time. This custom format represents the standard input and output format of the *PointCloudAnnotator*, which is further explained in Section 4.3.

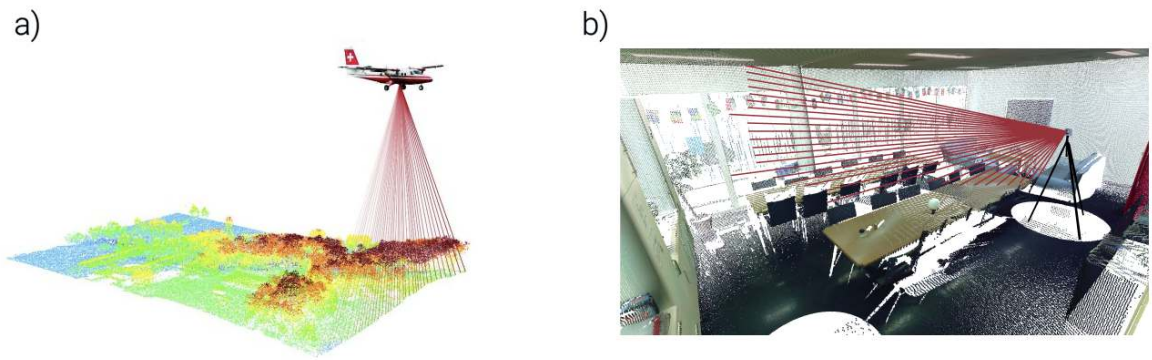


Figure 4.2.: a) Outdoor environments can be captured by airborne LiDAR – image courtesy of [28] – b) whereas terrestrial solutions best suit the scanning of indoor environments.

4.2.1. PLY

A 3D scan stored in a single PLY file consists of a header, followed by an unstructured list of samples, each representing the real-world point where the scan ray hit a surface and bounced back to the scanner. The header allows a correct interpretation of the file content by providing useful information about encoding and properties of the point data. As illustrated in Figure 4.3, the header always contain the standard encoding, the amount of points, and the list of properties describing each point entry. The actual data comes right after the header. Each line represents a point and stores exactly one value for each property specified in the header. In the specific case of Figure 4.3, each point, p, is defined by the following properties:

- Three values $[x, y, z]$ for the coordinates in local-space, usually in meter.
- Three values $[nx, ny, nz]$ for the surface normal at the point location.
- Four values $[r, g, b, a]$ representing the colour at p.

When the points are not associated to a normal vector, these need to be estimated in a preprocessing step, as described in Section 3.5.

4.2.2. PTX

Point clouds can also be constructed from a collection of panoramic depth maps, also known as RGB-D images. In this case, each scan produces a range image, whose pixels are associated to colour, intensity, and 3D location. The latter is recovered by the depth value captured by the acquisition device [31].

For this thesis, data sets of such a structure are provided in PTX format. Similarly to PLY, also files in PTX format present a header followed by a body containing the list of measured 3D points. However, the complete representation of a scenario is not given by a single file, but rather by a collection of files. Each PTX stores the result of a scan from a specific viewpoint. Thus, the coordinates of all points are in the local coordinate system and arranged according to the acquisition pattern.

As explained by Mura [24], usually each point p_{grid} is not only associated to a set of 3D coordinates, but also to a pair of 2D coordinates $[u, v]$. This results from a range-grid parameterisation of the scanner's field-of-view, as shown in Figure 4.4. Therefore, the uv-pair indicates the location of the

```
ply
format ascii 1.0
comment VCGLIB generated
element vertex 974957
property float x
property float y
property float z
property float nx
property float ny
property float nz
property uchar red
property uchar green
property uchar blue
property uchar alpha
element face 1921007
property list uchar int vertex_indices
end_header
-5.916499 -3.148408 0.8332924 6.260137 -0.07181954 0.2483732 176 162 141 255
-3.905072 -0.8437831 2.507934 0.3507019 -6.247457 -0.4019194 167 163 171 255
-4.396089 -1.207121 2.782564 0.3467845 -0.3081231 -6.242799 138 128 121 255
-5.250011 -2.760319 2.782175 0.0905648 -0.155496 -6.211409 140 122 102 255
-5.196424 -0.9713749 0.2161478 1.439837 1.181131 5.994089 135 130 126 255
-4.104257 -3.551718 2.829512 1.887193 2.663707 -5.052205 135 123 110 255
-5.89389 -3.244169 0.4059058 4.414335 1.237475 -2.834455 90 86 88 255
-2.713048 -1.318476 0.6882212 -6.256344 -0.02492289 0.07869491 162 146 134 255
-4.258155 -2.842693 2.806698 -0.004187984 -0.62884 -6.239729 110 103 77 255
-3.136596 -0.8755839 0.4222163 -0.1373205 -6.189478 0.6432647 99 97 103 255
-5.901421 -0.8653617 2.668411 -1.644418 -5.2482 -0.9609842 120 118 116 255
-2.655535 -2.718531 0.1222335 -1.590696 -0.1266687 0.115496 197 192 190 255
-4.071464 -3.09857 2.81323 0.229028 -0.3322661 -6.233681 112 105 79 255
```

Figure 4.3.: Initial part of a file storing a point cloud in PLY format.

relative point in the parameterisation. As in classic images, where the pixel location is implicitly derived from the underlying grid, the $[u, v]$ pair is directly inferred by knowing the storage order of points within the PTX: either row- or column-major. The complete environment is the result of the *registration* process, in which the various local scans are aligned to a global reference system. A 3D point cloud model before and after *registration* is illustrated in Figure 4.5.

Figure 4.6 is an illustration from different perspectives of PTX files' extracts, storing the data of the same point cloud.

The first two variables express the dimensions of the scanning device's uv-discretisation in terms of rows' and columns' amount, n_{rows} and n_{cols}. The next four lines represent the scanning viewpoint and the three primary axes (X, Y, Z) in global coordinate system [1]. These are followed by the rigid body transformation matrix, so-called registration matrix, stored in column-major order. This matrix must be applied to each point during the *registration*. Subsequently, the list of points in the parameterisation grid is stored line by line. Each point is associated with the following properties:

- Coordinates $[x, y, z]$ in the local reference system, expressed in meter.
- A scalar, i, holding the intensity of the point in range $[0, 1]$ (if stored as floating number) or in range $[-2048, 2047]$ (if stored as integer) [1].
- Three colour values $[r, g, b]$ in range $[0, 255]$.

It is worth stressing that points in a PTX file are not always assigned to a *valid* set of 3D coordinates. When a point is located at infinity – *i.e.,* the scanner ray did not hit any real-world surface – it is marked as *invalid* and will not be considered for the construction of the geometric point cloud model.

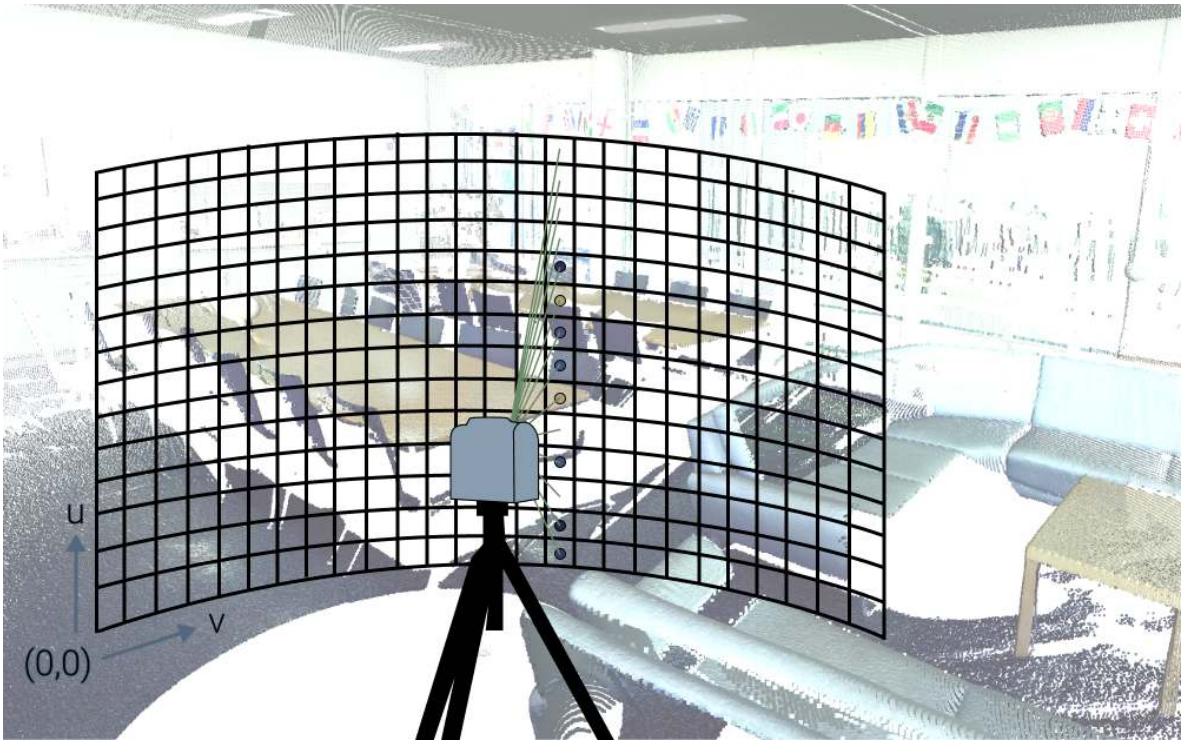


Figure 4.4.: When a point cloud is stored as a panoramic depth map, each 2D point location can be retrieved by considering a uv-parameterisation. To note is the presence of invalid pixels, which are not associated to any geometric point. This happens when no real-world object is encountered by the laser ray.

To summarise, while a panoramic map PM may contain N both *valid*, p^v, and *invalid*, p^{inv}, type of points

$$PM = \{p_i^v \cup p_j^{inv} \mid i = 1 \dots n, j = 1 \dots m, N = n + m = n_{rows} * n_{cols}\}$$

a PLY file P of the same size will contain only *valid* points

$$P = \{p_i^v \mid i = 1 \dots N\}$$

In this work, especially in Chapter 5, this distinction is expressed by using the terms $point$ to denote a *valid* geometric point, and $pixel$ to refer to an element of the parametric grid, or simply panoramic image. The latter can be either associated with a *valid* 3D point or not.

Some point cloud operations, such as the detection of shape features (see Section 3.4.2), require additional properties other than colour and 3D location. Relevant for this thesis is the computation of the normal vector associated to each point. Panoramic maps already provide the necessary data to estimate correctly oriented normals, according to the procedure explained in Section 3.5.

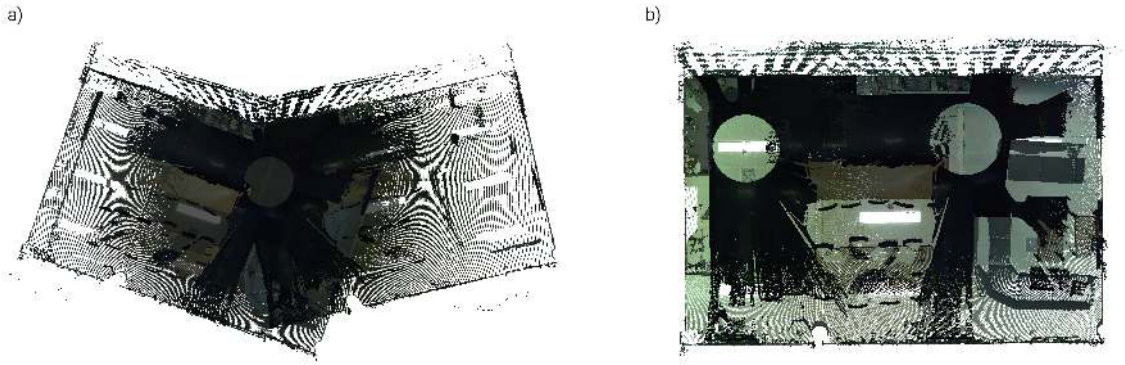


Figure 4.5.: a) The scanning files of a model taken from different viewpoints b) need to be registered into a global frame to consistently represent the original point cloud.

4.3. Output Format

Given a scanned point cloud model, either as single PLY or as a series of PTX files, the *PointCloudAnnotator* outputs a format consisting of four different types of binary file. A Point Cloud File (PCF) – holding the geometry information of all valid points within the input data set – two distinct files for geometric and semantic annotations, GLF and SLF respectively, and finally, when the input consists of multiple depth maps, a Panoramic Map File (PMF) is created for each map storing some of the properties associated to each pixel in the corresponding map. The next paragraphs provide a high-level description of the content of each said files.

4.3.1. Point Cloud File

Since this file represents the actual point cloud, exactly one PCF is produced for any scanned model. In fact, it holds the complete geometry of the data set in form of header followed by a list of 3D points. The header is composed by:

- Number of panoramic map files, n_{pmf}, in which the scanned model can be divided. Zero when the input is a PLY file.
- List of PMF filenames.
- Number of *valid* 3D points in the point cloud, *i.e.,* the size of the points' list.

In the list accompanying the header, each point has the following properties:

- ID of the point.
- $[x, y, z]$ coordinates.
- ID of the panoramic map containing the point.
- Normal vector $\vec{n} = [n_x, n_y, n_z]$
- $[u, v]$ coordinates of the point inside its panoramic map, where $u \in [0, n_{rows}[$ and $v \in [0, n_{cols}[$.

```
2578
1068
0.00000000 0.00000000 462.07976183
-0.11866899 0.99289775 -0.00846890
-0.99292812 -0.11863497 0.00441436
0.00337830 0.00893285 0.99995439
-0.11866899 0.99289775 -0.00846890 0.00000000
-0.99292812 -0.11863497 0.00441436 0.00000000
0.00337830 0.00893285 0.99995439 0.00000000
0.00000000 0.00000000 462.07976183 1.00000000
0.75981 -0.00091 -1.32350 0.37909 34 43 63
0.76487 -0.00091 -1.32475 0.34392 30 39 58
0.76867 -0.00092 -1.32382 0.34001 30 38 57
0.77596 -0.00093 -1.32883 0.34783 30 38 59
0.78378 -0.00094 -1.33468 0.35955 31 38 61
0.00000 -0.00000 -0.00000 0.34001 30 38 57
0.78405 -0.00095 -1.32024 0.32829 29 35 55
0.78842 -0.00096 -1.32020 0.32438 29 35 54
0.79526 -0.00097 -1.32425 0.32047 28 36 54
0.80152 -0.00098 -1.32726 0.33219 28 36 57
0.80794 -0.00099 -1.33050 0.33610 29 38 57
0.81277 -0.00099 -1.33107 0.35955 32 38 60
0.81740 -0.00100 -1.33129 0.34392 31 39 57
0.82156 -0.00101 -1.33072 0.33610 31 37 55
0.82093 -0.00101 -1.32241 0.33219 30 36 55
0.82773 -0.00102 -1.32607 0.32047 28 37 54
0.83045 -0.00103 -1.32319 0.32829 29 37 55
0.84058 -0.00104 -1.33205 0.33610 30 38 56
0.83566 -0.00104 -1.31706 0.33610 29 37 57
0.84459 -0.00105 -1.32394 0.32829 28 36 56
```

Figure 4.6.: Initial part of a file storing a point cloud, in PTX format.

As a note about the size of the list, when the scanned model was provided as PLY, the number of *valid* points, p_i^v, equals the amount of entries in the body of the PLY file. If instead the data source is given as collection of panoramic maps, PM, the size can be formally expressed as

$$\sum_{k=1}^{n_{pmf}} \sum_{i=1}^{n_j} p_i^v \in PM_k \qquad n_j = N_j - m_j, N_j = n_j^{rows} * n_j^{cols}$$

where m_i is the number of *invalid* points in PM_j.

4.3.2. Panoramic Map File

When the input data set consists of a collection of panoramic depth map, a PMF is created for each map. The main purpose of such file is to represent each single scan as a panoramic image. Its structure looks as follows:

- Number of rows of the panoramic map, n_{rows}.
- Number of columns of the panoramic map, n_{cols}.
- $4x4$-alignment matrix that brings this scan into the global reference frame, also known as *registration* matrix.
- List of pixels of the panoramic map.

Each pixel in the list holds the following information:

- $[r, g, b]$ colour values.
- Reflectance or intensity value.
- ID of the corresponding 3D point in the PCF file. When the pixel is associated to an *invalid* point, this ID is set to -1.

4.3.3. Geometric Labelling File

The GLF holds the information regarding the geometric annotation of the point cloud. Users has the possibility to associate a specific primitive type with a subset of points or pixels. For example, if after performing an automatic shape detection on a subset, G, a primitive of type "cylinder" results as approximating G the best, the user can decide to assign that specific primitive to G. The following information will be thus stored in the GLF:

- Number of primitives in which the data set is segmented.
- List of primitives, each consisting of:
  - ID associated to the primitive type
  - Name
  - Parameters to uniquely define that primitive (see Table 4.1 for the list of parameters by shape).
- List of assigned primitive labels – for each point of the PCF, the ID of the primitive to which it belongs is stored.

If the user does not perform any operation, or any selection of points or pixels, the file remains empty.

Primitive	Parameters	Definitions
Sphere	Centre	$c = [x_0, y_0, z_0]$
	Radius	r
Cylinder	Axis	$a = [x, y, z]$
	Radius	r
Cone	Apex	$a_p = [x_0, y_0, z_0]$
	Axis	$a = [x, y, z]$
	Aperture angle	α
Torus	Centre	$c = [x_0, y_0, z_0]$
	Axis	$a = [x, y, z]$
	Minor Radius	r
	Major Radius	R
Plane	Normal	$\vec{n} = [a, b, c]$
	Coefficient	d

Table 4.1.: The automatic shape detection can be performed on some predefined primitives, each of which is defined by a set of shape-specific parameters as provided by CGAL [27].

4.3.4. Semantic Labelling File

The second component of the point cloud annotation is stored in the SLF. Each subset of the scanned model, G, can additionally be assigned to a label, expressing the semantic meaning associated to G. For instance, users can define one subset for each chair in the scene and assign each of these subsets to the label "Chair". In order to recover the correct semantic information for each subset, the SLF contains the following information:

- Number of objects considered as "semantic entities" in which the data set is segmented.
- List of semantic labels associated to the entities, *i.e.,* a list of names.
- List of assigned semantic labels, *i.e.,* a sequence of IDs – each corresponding to a semantic label – having the same size as the list of points in the PCF. In fact, the position index within the sequence of semantic IDs corresponds to the ID of the point.

A default name is always automatically assigned to any subset G, created by the user. When no semantic label is assigned to G, this is exported as unlabelled and can still be retrieved when the SLF is imported in the *PointCloudAnnotator*. This mechanism preserves the integrity of the user manual segmentation.

5. Implementation

Figure 5.1.: 3D clusters resulting from a segmentation operation are also displayed on panoramic images.

5.1. Overview

Given a scanned model of an indoor environment, either in the form of unstructured 3D geometry or panoramic maps, the *PointCloudAnnotator* allows users to perform both geometric and semantic analysis of the point cloud. The result is stored together with the initial structure of the model, according to the format introduced in Section 4.3.

The *PointCloudAnnotator* accepts input models , either from standard point cloud formats – such as PLY and a collection of PTX files – or from a packet of already toolkit-specific-formatted files. In the last case, no new PCF or PMF are generated, but it is possible to update the content of the annotation files. In case of PLY files a requirement is specified. Since the information contained in a PLY file is not sufficient to correctly estimate – and above all orientate – the normals, the input file must provide normal vectors associated to the points. If this is not the case, a script is available to preprocess a collection of PTX files, producing the corresponding point model with normals as PLY. This script is further discussed in Section 5.2.

Depending on the original input source of the point cloud, its visual representation differs. If the data set is provided as a PLY file, only the geometric model displaying the 3D points is rendered. When the source is a collection of panoramic maps, the *PointCloudAnnotator* displays the geometric cloud together with a panoramic image for each input depth map.

In order to annotate a given point cloud, users can create *group* entities to be filled with points. A *group*, representing a semantic object, can be labelled using an appropriate name. A predefined primitive shape can also be assigned to it. For example, when a *group* describes a table, it is probably labelled as “Table” and associated to a “plane”.

A user can add single points and pixels to a *group* by direct selection, as well as performing a group selection of points. In the geometric model, the user can choose between a sphere and a frustum selection.When satisfied, the user can add the resulting group of points to the desired *group*. Group selection in the panoramic images is performed as region growing, starting from the pixel clicked by the user.

Groups can also be automatically created after the execution of specific operations. The content of multiple *groups* can be merged into a new, single one, by means of boolean union. Automatic and semi-automatic segmentation operations produce additional *groups* of points as well. The full automatic segmentation is performed globally, producing a planar segmentation of the entire point cloud, hence one *group* is created for each detected plane. Similarly, five temporary *groups* are created after the semi-automatic local segmentation of non-planar shapes[1]. Users have the chance to accept the result associated to the best fitting shape and convert it into a definitive *group*. A more detailed description of interactive features provided to users for the actual annotation are discussed in Section 5.5.

Whenever the user wants to store the current progress, the information about the *groups* can be exported to GLF and SLF files, such that when a new session is started the annotation can be extended, as explained in Section 5.4 about the general pipeline of the *PointCloudAnnotator*. Section 5.3 provides an overview on the frameworks underlying the toolkit’s architecture.

[1]For a better distinction with the notion of global *planar* segmentation, the local segmentation will be generally qualified as *non-planar*, although the algorithm searches for planar shapes as well.

5.2. Data Preprocessing to Generate PLY Files

If the original data in PLY format does not specify any normal, the *PointCloudAnnotator* is not allowed to import it. This restriction is necessary as certain operations requires normals to properly function, for example the RANSAC segmentation. In order to outwit this constraint, a script is available to preprocess PTX files, which are eventually combined into a single PLY, including normal data.

In said script, points from each PTX file are arranged into a `pcl::KdTree` to accelerate the retrieval of the local neighbourhood for each point. The resulting points are then fitted to the PCA-based algorithm explained in Section 3.5 to estimate the respective normal. Finally, if needed, the normal is consistently oriented inward with respect to the indoor environment, represented by the point cloud.

In contrast to PLY, normals for data coming from PTX files can be compute at loading time, since the necessary data for estimation and re-orientation is already provided.

5.3. Frameworks

The *PointCloudAnnotator* has been written in C++ with the integration of the Qt framework[2] to build the user interface (UI). The Point Cloud Library (PCL) and the Computational Geometry Algorithms Library (CGAL) provide the necessary support to efficiently handle point clouds.

The design process of Qt-based interfaces relies in building a structure of so-called Qt widgets, connected by a system of signals and slots. A Qt widget is a UI element – a button, a slider, or even a popup input dialog – which can communicate with another widget or, more generally, another Qt object by sending a signal, to which a slot will be executed in response.

An feature of Qt worth mentioning is the opportunity to declare any (C++) class as Qt object, which is thus able to receive and send signals. This functionality enables an easy interaction between UI elements and the core engine of an application. Qt is not only a powerful framework to display pure UI items, but it also provides a special type of widget, the `QOpenGLWidget`, conceived expressly for the rendering of OpenGL graphics. Hence, Qt keeps the rendering of widgets and OpenGL objects in separate contexts, facilitating resource management.

The PCL library [3] was developed to improve the 3D perception of point clouds – mainly in the context of robotics – and thus contains solutions for feature estimation, model fitting, segmentation, and others [33]. The 3D processing algorithms in the library share a basic working principle. First, a PCL-point cloud must be created from the scan data source. This cloud structure serves then as input for the object that performs the desired operation (*e.g.,* a normal estimator or a feature detector). In this thesis, the *io*, *kdtree*, and *octree* components were mainly used to manipulate point cloud models.

As complement to PCL, the CGAL software project [4] has been chosen for this thesis. It provides various data structures and geometric algorithms contributing to the field of computational geometry. For this reason, CGAL is widely used in projects from many application areas, ranging from architecture and urban modelling to astronomy, as well as from Geographic Information Systems to Computer Vision [16]. Some geometric challenges in this work were solved through techniques offered by CGAL, especially 3D shape detection as segmentation method, explained in Section 3.4.2.

[2] Documentation available at: `https://doc.qt.io`. Accessed December 13th, 2020.
[3] Official webpage: `https://pointclouds.org`. Accessed December 13th, 2020.
[4] Additional resources at: `https://www.cgal.org`. Accessed December 13th, 2020.

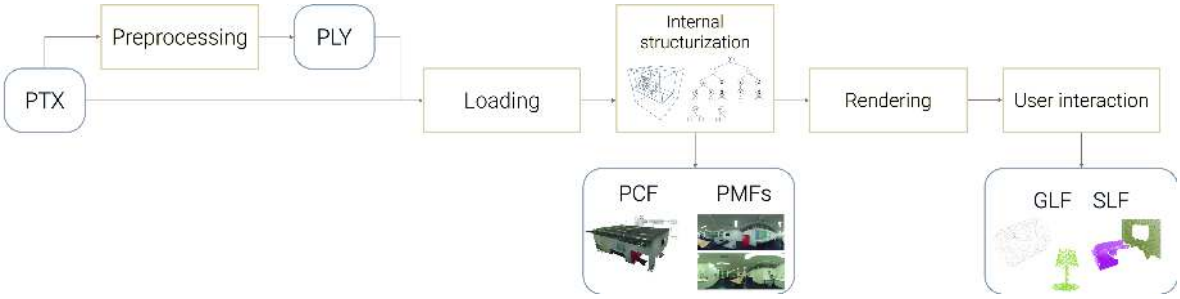


Figure 5.2.: Generally, the *PointCloudAnnotator* loads point clouds from scanning data. The user is then allowed to interact with the visual representation of the model and to perform annotations on the data.

5.4. Rendering Pipeline

The initial rendering structure was based on a VMML internal project. On top of it, the engine has been modified by integrating the Qt-based GUI and functionalities provided by PCL and CGAL. The architecture has been additionally expanded with the toolkit-specific functionalities to easily annotate a given point cloud model. The general mechanism of the *PointCloudAnnotator* is illustrated in Figure 5.2.

The *PointCloudAnnotator* combines pure engine logic and rendering mechanisms: the first is required to manage resources and users' interactions, whereas the second draws the model and feedbacks in response to users' operations. A simplified visualisation of the toolkit's architecture is shown in Figure 5.3. As the name already suggests, the `Core` class is in charge for the global coordination, among rendering- and data-specific classes. The `Core` must always ensure that each class receives all the necessary data to execute the operations it is responsible for.

The `Core` class uses the functionalities provided by the virtual class `ScanDataManager` to read the input files and, when needed, to export the annotation state in the four types of formatted files.

The data just read is passed to `PointCloudData`, which distributes it among the internal structures and the octree. These structures are further described in Section 5.4.1. As soon as the data structures are ready, another one is created for rendering. An instance of `PointCloudDrawable` exclusively holds the information required for rendering the 3D point cloud model and its associated panoramic images.

Operations such as RANSAC segmentation, search-in-tree, and region growing are defined by the general term of mathematical operations. These are performed within the `MathHelper` class. In most cases, the input and output of a mathematical operation consist of *groups* of points that are handled by the `GroupsManager` class.

The UI is an essential component of this thesis, as it enables the user to manipulate point cloud models. All classes responsible for displaying UI elements are QT objects, edged in green in Figure 5.3. Such objects can handle QT widgets and take advantage of the signal/slot mechanism, addressed in Section 5.3. The `PclAnnotator` class defines the entire UI design and captures user inputs from it. In this occurrence, it triggers functionalities within the `Core`, which in turn updates the UI through signals. The actual rendering is performed by `GLWidget` objects associated with a `Scene` to manage the content of the widget at a higher level. Other than pure object rendering, the `GLWidget` also forwards mouse and keyboard events to the `InputManager` and the `Camera`. A more detailed description of this mechanism is provided in Section 5.4.2.

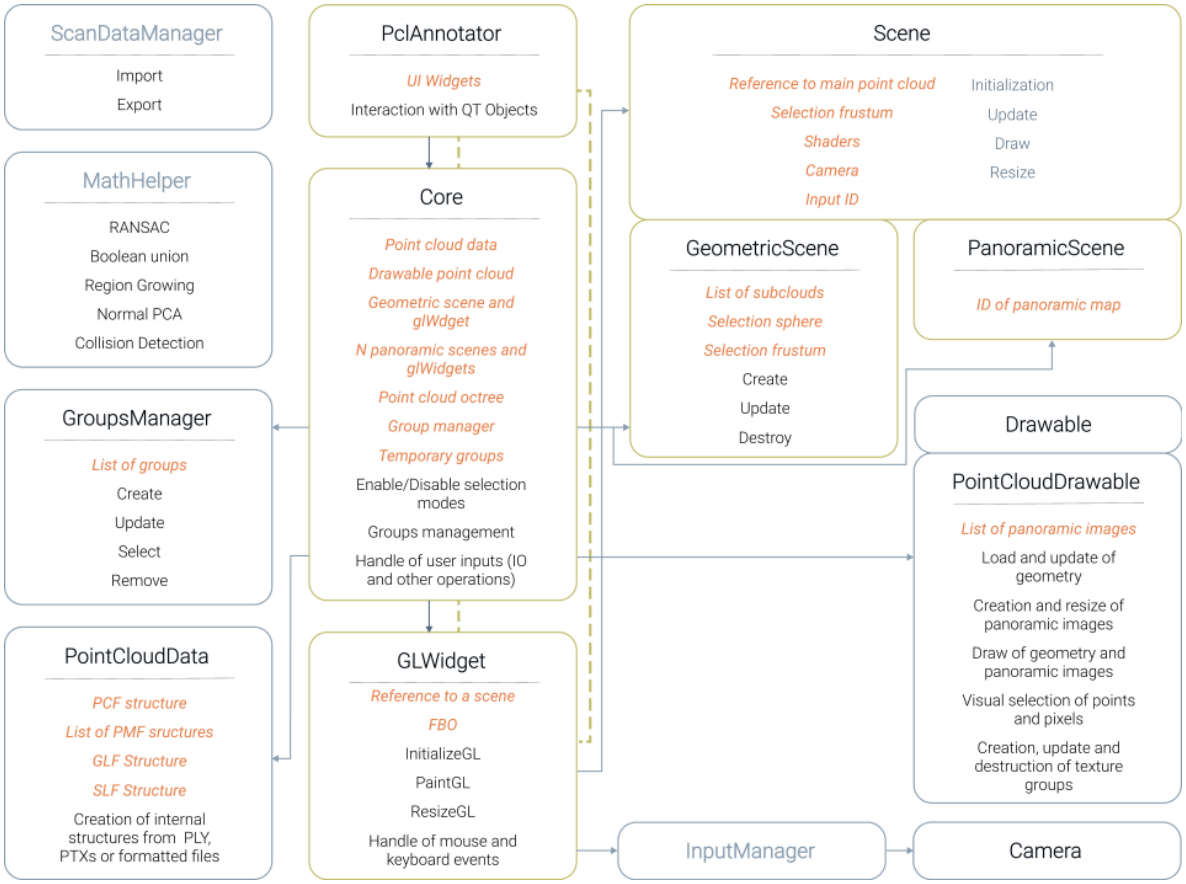


Figure 5.3.: A simplified overview of the *PointCloudAnnotator*'s architecture. In green, Qt Objects are the components that can communicate by means of signals/slots. These are mostly in charge of handling UI inputs and rendering loop. The `Core` is accountable for the global coordination among rendering- and data-specific classes.

The following sections present a thorough insight of the main *PointCloudAnnotator*'s features. A more complete vision of the thesis' application is reported in Appendix A, with the support of a class-diagram.

5.4.1. Import and Loading of Internal Structures

Regardless of the format, loaded data is internally processed in order to build the spatial structure that serves to accelerate the operations within the toolkit, as well as to facilitate the export of aforementioned format. The internal structure of the point cloud data – in the `PointcloudData` class – reflects the output structure: PCF, GLF, SLF, and possibly PMFs. The *Pcf* structure contains the information stored in the PCF file. Principally, it is a list of points consisting of 3D coordinates, normal vector, panoramic coordinates, and two IDs. One ID – which also corresponds to the index location of the point in the list – represents the point itself. The other ID indicates the panoramic map containing the point, if any is provided. In fact, while other structures may be empty, upon loading, the *Pcf* structure will always contain data. *Pmf*s are derived uniquely from PTX files.

Assuming that N PTX file are loaded, N *Pmf* structures will be created, each holding a list of pixels with colours, reflectance, and an ID which serves as foreign key to link pixels to the corresponding

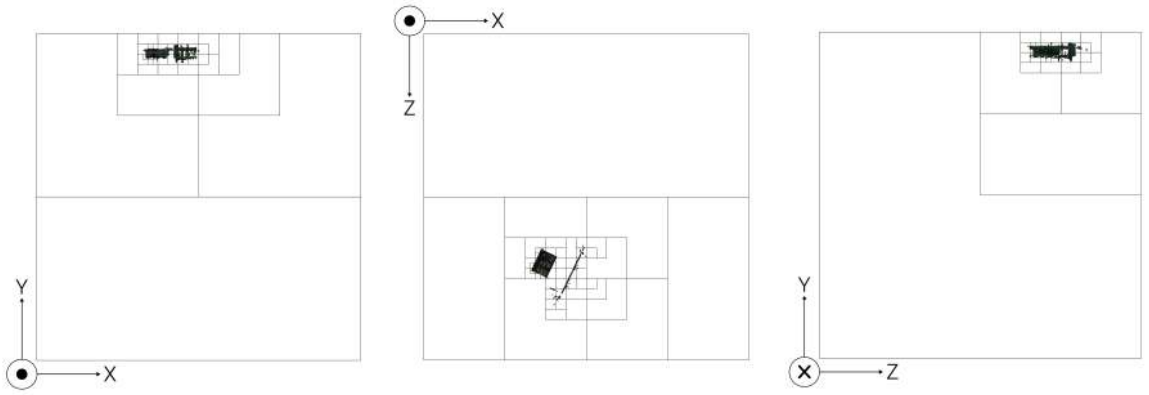


Figure 5.4.: Three different perspectives of a point cloud model inserted in the octree structure, holding the corresponding data, from left to right: front, top, and side view.

points in the *Pcf*. *Glf* and *Slf* are only filled by user request. These have similar structures, though the *Glf* stores a list of primitives, while the *Slf* records a list of semantic labels. Both keep a fix-sized list of IDs, indicating a primitive or a label respectively, which implicitly assigns a geometrical or semantic meaning to the points.

These structures help exporting the annotation process' result in the proper format, but they do not optimise handling data during the annotation, which must be performed in a real-time interactive manner. For this reason, the octree spatial acceleration structure provided by PCL is employed. An example of point cloud model with the corresponding octree is illustrated in Figure 5.4. Such structure largely improves the execution of search and intersection operations, thus allowing fast responses to spatial queries. For instance, some types of 3D-selection implemented in the *PointCloudAnnotator* traverse the octree to efficiently retrieve a set of points.

These internal structures are filled with data loaded according to a process which varies depending on the input format, as presented in Figure 5.5. A valid PLY file (*i.e.,* including normal vectors) is read using the function provided by PCL, which directly loads it into a PCL-point cloud structure.

When the user inputs a collection of PTX files, each one of them is read and combined into the `PointCloudData` structures. The normals are computed before creating the PCL-point cloud object.

The loading of already formatted binary files is much faster as the data is directly written into the corresponding structures. The only additional step required is to build the PCL-point cloud after reading the *Pcf*.

Before starting the actual rendering, the systems checks whether annotation files – GLF and SLF – were provided by the user. If so, the necessary *groups* are created as described in the *Glf* and *Slf* structures just loaded. Once the points have been assigned to the respective *groups*, the entity and necessary rendering structures are created.

5.4.2. Visualisation

The rendering system in the *PointCloudAnnotator* has been logically divided into classes in charge of the pure rendering process and classes managing the data to render. A visual representation of such system is given in Figure 5.6.

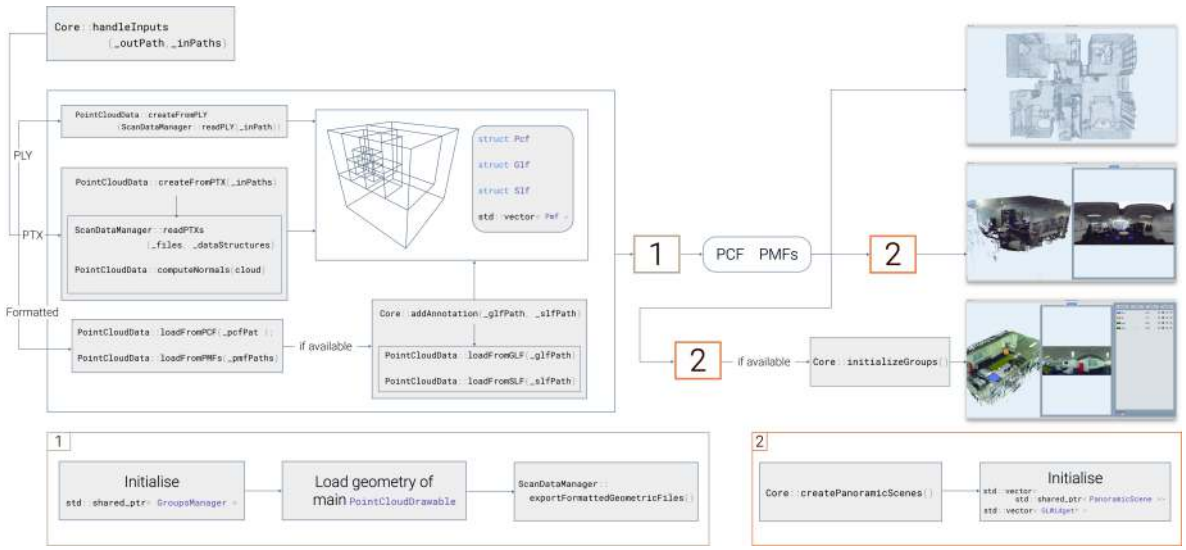


Figure 5.5.: Depending on the format of the input file, internal data structures (octree and `PointCloudData`) are filled in different ways. Afterwards, rendering structures are created and geometric files can be exported. If the input file is provided with depth maps, the associated panoramic images are additionally displayed. The content of the labelling files is loaded into *groups*, when they are available.

Basic Rendering Loop

As briefly explained in Section 5.3, the rendering of OpenGL objects happens within a `QOpenGLWidget`. The events triggered by mouse or keyboard are captured in the main UI class, `PclAnnotator`. These are forwarded to the instances of `GLWidget`, which handle them according to the type of event. A default FBO is associated to each `GLWidget` but in order to enable mouse picking at runtime, an offline FBO of type `QOpenGLFramebufferObject` has been additionally created to support Multi Render Target (MRT).

Usually a mouse event needs to be further processed in order to move the camera or to perform a conversion of coordinate system useful to next operations. Such cases are handled by the `InputManager` class which serves as interface between the `GLWidget` and the camera which is affected by mouse events. The class additionally provides helper methods for the conversion of mouse coordinates into world- or object-space.

Scenes Mechanism

The rendering system of the *PointCloudAnnotator* features `Scene` classes that manage the resources to display on screen according to the instructions from the `GLWidget`. In order to do so, any instance of `GLWidget` holds an object of the superclass `Scene`, which serves as general interface to the specific type of scene the `GLWidget` controls.

`GeometryScene` objects are responsible for the rendering of the point cloud as 3D model. At rendering time they make sure to retrieve the 3D data associated with the point cloud and render it using the proper shader program. Additionally, they also control if other objects need to be rendered in the 3D scene, for example, the 3D representation of the *groups* created by the user. Since these consist of a `PointCloudDrawable` instance holding a subset of the main point cloud data, they will be

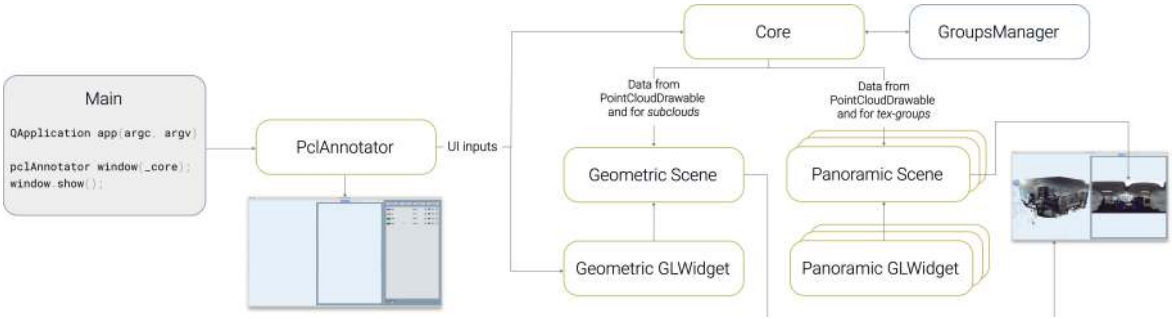


Figure 5.6.: The *PointCloudAnnotator* is a `QApplication`, whose rendering system is based on Qt objects. UI inputs trigger responses in the `Core` which updates the scenes' data. Keyboard and mouse inputs affects the `GLWidget` objects, which technically control the rendering of the scenes.

referred to as *subclouds*. Based on the information constantly received from the `GroupsManager` and from the `Core`, the `GeometryScene` is able to manage the *subclouds* so to express the current state of the *groups* in 3D. Other objects managed by `GeometryScene` are the 3D polygons representing the shapes for some selection modes. `GeometryScene` ensures that these are are displayed when needed.

Instances of `PanoramicScene` display a panoramic map associated to the point cloud as a 2D image. They hold the ID of the panoramic image they ensure it will be drawn with the correct shader throughout the rendering loop.

Basic Rendering Structures

The scenes keep a reference to the main rendering structure, which is constantly updated: the `PointCloudDrawable` object representing the main point cloud. This class is derived from the `Drawable` class which exclusively holds structures and perform functions necessary to rendering.

`Drawable` mainly manages the rendering buffers for the shaders and OpenGL draw calls. Among the traditional buffers – for positions, normals, colours, etc. – a *selection buffer* is provided to serve as a kind of mask to temporarily display which part of the object is currently selected. More about its purpose in Section 5.5

The content data in `Drawable` is provided by the derived classes such as `Sphere`, `Frustum`, `Quad` and `PointCloudDrawable`.

Rendering Structures for 3D Point Clouds

In the *PointCloudAnnotator*, instances of `PointCloudDrawable` represent the main point cloud model but also the *subclouds*, which do not necessarily exploit the full capabilities of the class. In fact, the class was mainly designed to describe a scan model in all its forms, purely geometric but also its panoramic perspective when available. The following explanation will consider the case of a point cloud represented by a collection of panoramic maps. When the source is provided as an unstructured list of points, the `PointCloudDrawable` object will simply contain geometric information without any definition of panoramic views.

The main functionalities of the `PointCloudDrawable` class can be summarised in loading and updating the geometry data – from the *Pcf* –, managing a list of structures representing a panoramic image – the *PanoramicView*s – and drawing these two types of entities.

The responsibilities involving geometric data do not require a particularly high level of complexity. It is simply a matter of transmitting data from the `Core` to the base `Drawable` class and to request the rendering functionalities upon demand by the `GeometryScene`, which in turn is solicited by the `GLWidget`.

More complex and interesting is the management of the resources to display the point cloud model as panoramic maps.

Rendering Structures for Panoramic Images

Graphically a panoramic map is rendered as a textured quad, so the *PanoramicView* structure is designed to store the drawable object of type `Quad` and the texture representing the actual image wrapped as a `Texture` object. During the initialisation phase, after the `PointCloudData` is loaded, the `Core` creates as many `PanoramicScenes` and related `GLWidget` as there are *Pmf*s. At the same time the list of *PanoramicView* objects is updated with a new item which is created as follows. Two textures are generated with the same size as defined by n_{rows} and n_{cols} in the *Pmf*. The first one contains the rgb-colour of each pixel. The second texture serves as a mask to display which pixels belongs to a *group* and it is initialised to be completely white.

In fact, the mechanism to display *groups* in a panoramic image is slightly different that the system of *subclouds* managed by the `GeometryScene`. The 2D representation of the *groups* is directly managed by the `PointCloudDrawable` in response to the inputs from the `Core` and the `GroupsManager` – which has complete control of the current state of the *groups*. Therefore, each *PanoramicView* has a list of what will be called *texture groups*, each containing the colour of the *group*, a list of uv-coordinates for valid points and a list of pixel indices – relative to the location in the *Pmf*'s list – for invalid points. The advantages of storage separation between valid and invalid points will become more clear when the exact mechanism of the *groups*' management by the `GroupsManager` will be explained in Section 5.5. However, it can already be anticipated that this strategy optimises interactive operations that can be performed both on 3D points and on 2D pixels. At runtime the content of *texture groups* is updated or even completely deleted and after each manipulation the content of the mask texture associated to each *PanoramicView*'s quad is updated as well.

It is worth stressing that, to achieve a sort of transparency and avoid that already visible *groups* are hidden by new ones, the colours in the mask texture are not simply overridden, rather blended with the new colours, as Figure 5.7 illustrates.

Shaders

Four types of shaders, `cloud`, `quad`, `subCloud` and `mesh`, are responsible to render the main point cloud, the textured quad representing the panoramic image, the *subclouds* and the other meshes to help the visualisation of the 3D-selection, respectively.

Other than the standard colour, all these three shaders write an additional attachment in the FBO with some information about the fragment which will be used for the mouse picking technique described in Section 3.1. As we will see in Section 5.5.2, various selection operations need access to the following data: point ID, depth and a value to distinguish fragments belonging to objects from background fragments. To note, is that the point ID is not written by the `quad` as this information can be directly retrieved form the texture stored in the CPU memory.

Figure 5.7.: *Groups* in the panoramic images are visualised by blending the respective points into a mask texture. The points in the *texture groups* are represented by their uv-coordinates, which directly indicate the texel to mask.

5.4.3. Export

Upon loading of PLY and PTX files, the geometric files according to the format presented in Section 4.3 are immediately created at the location specified by the user. If no output path is provided, default locations are used. For PLY this corresponds to the same location as the input source, while a common name is determined among the provided PTX files. Any time during the session the user can export the current state of definitive *groups* as visible in the corresponding UI panel. Two functionalities are provided: "Export" and "Export As". Both create a pair of annotation files, GLF and SLF, in the given output location. In general, "Export" stores the files in the last path used while "Export As" always asks for the output location. However, when the output path is not internally known, both functions will ask for the output path.

Thanks to "Export As" it is possible to create multiple annotations, in terms of output files, all associated to the same point cloud model. Hence, the annotation currently loaded in the toolkit can be replaced at any time by importing a new pair of GLF and SLF files.

Upon data export, users are asked wether a mechanism to track empty *groups* should be activated. Furthermore, it is worth stressing that *groups* not associated to any primitive and having the exact same label will be merged into a single *group* when being exported to the annotation files.

5.5. Interactive Operations

The most important functionalities provided by the *PointCloudAnnotator* consist of interactive tools allowing the user to manipulate a point cloud model. These can be subdivided into following categories: mechanism of *groups* to supervise the semantic as well as the geometric annotation, selection methods for single or multiple points, and methods for the fitting of 3D shapes.

5.5.1. *Groups* Mechanism

A *group* is a collection of – either valid or invalid – points, actually a subsample of the main point cloud, that can be associated to a geometric primitive best representing the shape the points resemble, and labelled so that a semantic meaning is assigned to that *group*. In both geometric and panoramic view a *group* and all its point and pixels are highlighted by a colour layer associated to that *group*, colour that can be changed at any time.

All existing *groups* are listed into a dedicated panel, that will be presented in Section 5.6.

Groups are not allowed to intersect, meaning that a point can be associated to exactly one *group* at most. To ensure that, when new elements are being added to a *group*, all the points already belonging

to another *group* are automatically removed from that *group*. After removal, the points can be added to the new *group* without violating the "non-intersection" rule. In a sense, difference and intersection boolean operations within *groups* are implicitly implemented any time something changes in the *groups*' state.

Such behaviour was not planned from the beginning but it has been decided as the thesis was progressing mainly to ensure that the state of the *groups* could be uniquely reconstructed from the annotation files.

As a consequence, some limits are introduced as well. For example, definitive *groups* which have been already fitted to a shape and refined as the user desires, run the risk of still being modified, namely by loosing points that are added to another *group* being manipulated. In order to prevent such undesired events to happen a lock mechanism has been introduced. The user can decide to lock a *group* which then is guaranteed to remain untouched until the lock is released.

As a side note, since invalid pixels are not associated to any geometric point, they are not exported to the annotation files. They may still be added to *groups* during an annotation session, but to avoid increasing the complexity – thus affecting the performance unnecessarily – the "non-intersection" rule is not applied.

Existing *groups* can be merged together by means of a boolean operation, which as explained above does not have to deal with possible duplicates.

Every operation explained so far affects the current state of the *groups* and is thus is exclusively handled by the instance of the `GroupsManager` owned by the `Core`. The `GroupsManager` creates and deletes *groups*, updates their content and appearance, set and release locks and keeps track of which *groups* are currently selected and thus must be visualised.

A mapping between points and *groups* is used to supervise the insertion process, which is reported in Listing 5.1. The point is inserted in the *group* if this is not assigned yet or if it belongs to a non-locked *group*. In the latter case, the point is first removed from the *group* it is currently in, before being added to the new *group*. The ID of every *group* that has lost some points is inserted into a list, which is used to update the visualisation of the *groups* that have been affected by the insertion of points into another *group*. As mentioned above, this procedure only handle valid points, when invalid pixels need to be added, they are directly inserted in the corresponding list simply ensuring to avoid duplicates. In fact, every time an insert operation must be performed, it is first ensured that the set of items to insert is divided into valid and invalid points, namely in point IDs and pixel indices. This way the `GroupsManager` can perform the insertion accordingly. In this respect, a selection from a panoramic image is more demanding than the same operation on the 3D model, where the points are all obviously valid.

As soon as the `GroupsManager` performed a state update of the *groups*' structure, the *Core* is responsible to ensure that the visualisation structures of the *groups* are updated correspondingly. Hence, the buffer's content of the *subclouds* in the `GeometryScene` is updated as well as the content of the *texture groups* in the `PointCloudDrawable`.

5.5.2. Selection Methods

As already anticipated in the previous sections, the *PointCloudAnnotator* provides multiple types of selection modes for both points in the 3D model and pixels in the panoramic images. The operation of any selection mode is based on mouse inputs from the user, which are handled by the `GLWidget` instance.

```
1  void GroupsManager::removePoint(int _pointID)
2  {
3      int currentGrID = this->getGroupIndexByID(this->groupsMapping[_pointID]);
4      auto& currentGr = this->groups[currentGrID]->pointIDs;
5      currentGr.erase(std::remove(currentGr.begin(), currentGr.end(), _pointID),
6                      currentGr.end());
7      this->groupsMapping[_pointID] = -1;
8  }
9
10 // Returns true upon successful insertion
11 bool GroupsManager::insertPoint(int _groupID,
12                                 int _pointID)
13 {
14     int belongingGroup = this->groupsMapping[_pointID];
15
16     // Point is already assigned to the correct group
17     if (belongingGroup == _groupID) {
18         return true;
19     }
20
21     // Point belongs to a locked group
22     if (this->lockedGroups.find(belongingGroup) != this->lockedGroups.end() ) {
23         return false;
24     }
25
26     // Point is already assigned to another non-locked group
27     if (belongingGroup != -1) {
28         // Mark group as modified
29         this->updatedGroups.insert(belongingGroup);
30         // Remove point from previous group
31         this->removePoint(_pointID);
32     }
33
34     // The point needs to be (re)assigned
35     this->groupsMapping[_pointID] = _groupID;
36
37     // Insert at correct position in new group so that the pointIDs are sorted
38     auto& gr = this->groups[this->getGroupIndexByID(_groupID)]->pointIDs;
39     gr.insert(std::upper_bound(gr.begin(), gr.end(), _pointID),
40               _pointID);
41
42     return true;
43 }
```

Listing 5.1: Valid points are inserted into a non-locked *group* by ensuring that the "non-intersection" rule is respected.

Single point

When the user clicks on a 3D point belonging to the point cloud model, the information related to that point are read directly from the FBO as explained in Section 3.1. The useful information in this case is the point ID that is signalled by the `GLWidget` to the `Core`, which in turn executes the slot function to handle the single selection event. Here, if the user selected a *group* from the panel and if the point does not belong to a locked group, the point is added to the selected *group* entity in the `GroupsManager` as well as to the relevant visualisation structures, namely the *subcloud* and possibly the *texture group*.

Multiple points from sphere intersection

This mode allows to select a set of points resulting from the intersection between the point cloud model and a sphere. When entering this mode, the user can place such sphere by clicking on a point

in the cloud which will serve to obtain the anchor via mouse picking. The depth of the point is read from the FBO and together with the mouse position allow the computation of a 3D anchor point in world-space. The sphere can also be moved in the camera-relative xy-plane, and along the z-axis by increasing or decreasing the distance from the camera. Every time the sphere intersects the point cloud, the `pcl::radiusSearch()` is performed on the octree to retrieve the points within the sphere's radius from the sphere's centre. The resulting set consists of point IDs and is stored into a temporary *group* kept by the `Core`, which is also highlighted in the model by updating the selection buffer of the `PointCloudDrawable`.

If the user wants to add the current selection to an existing *group* the `Core` passes the content of the temporary *group* to the `GroupsManager` which inserts the point into the corresponding *group* according to the procedure in Listing 5.1, which ensures that the "non-intersection" rule is not violated also considering locked *groups*. Afterwards the rendering structures of *groups* affected by the insertion are updated, namely *subclouds* and *texture groups* if present. We have seen that in order to correctly update the visualisation of the *groups* in the panoramic images, the set of intersecting points must first be distributed among the maps. Since the temporary *group* is stored as a list of point IDs, it is sufficient to retrieve the ID of the panoramic map containing that point from the *Pcf* structure of `PointCloudData` and assign the corresponding uv-coordinates to the selected *group* of the correct *PanoramicView*.

Multiple points from frustum intersection

Thanks to this selection mode the user has the possibility to draw a rectangle on the screen, which is then used as input to build an orthogonal frustum, whose intersection with the point cloud model produces the selected set of points.

The drawing process of the rectangle starts with a click somewhere in the geometric scene. At this moment the *GLWidget* stores the mouse coordinates at this initial position. A second click will determine the point along the diagonal defining the rectangle. Given the points on the diagonal, the other two vertices of the rectangle in screen-space are straightforward to derive. The 3D frustum will be created only if the initial 2D rectangle just built contains some points of the main cloud model. This is because the frustum will be placed between the depth range of the included points. Therefore, the near plane – actually the face closest to the camera – will be at the minimal depth among these points while the far plane will be placed at the maximum of such depths. If there is only one point included in the rectangle, the frustum will have a default depth of one meter. The depth range is determined with OpenGL picking and namely by reading the region defined by the 2D rectangle in the FBO. Afterwards the four vertices of the rectangle drawn on the screen are converted into world-space coordinates by the `InputManager` which uses the mouse coordinates of each vertex and the minimal depth just found.

The set of world coordinates and the $depth$, defined as the difference between maximum and minimum depth, are passed to the `Core` and forwarded to the `GeometryScene`, which eventually generates the drawable `Frustum` object. First, it needs to compute the four vertices of the frustum's far plane. Each vertex of the near plane defines a far vertex as follows:

$$p_{far} = p_{near} + (dir * depth)$$

where dir is the vector defined between the centre of the near face and the centre of the far face. If dir cannot be determined, the normal vector of the near face is used as default direction.

When the `Frustum` object is generated, its OBB is defined as described in Section 3.2. The OBB is first converted into the model-space of the point cloud model so that it can be intersected with the

octree to produce the resulting selection of points. The traversal procedure is described in Section 3.3 and the resulting set of point IDs is stored in the temporary *group* introduced in the previous section about sphere-intersected selection. The addition of the frustum intersection's result to a definitive *group* occurs therefore in the same way explained above.

Unlike the sphere, the frustum object cannot be moved by the user but a reset button allows the generation of a new object. Nevertheless, the depth of the frustum can be dynamically changed by means of a slider in the UI. Every change in the frustum's size triggers the performance of a new search in the octree and thus the update of the temporary *group*'s content.

Single pixel

In Section 5.4.2 we have seen that the shader for the panoramic images writes the depth of the fragments in the FBO layer. When picked this depth is used to compute the position of the selected pixel in object-space, p_{obj}. The xy-components of p_{obj} are then parametrised to retrieve the uv-coordinates corresponding to the selected pixel in the texture representing the panoramic image. Listing 5.2 shows the details of such process. The function uses a helper method from `InputManager` to convert mouse coordinates from screen- to object-space, in this case the coordinate system of the quad textured with the panoramic image.

These uv-coordinates are then evaluated by the `Core` in the connected slot method. The index in the pixels' list of the *Pmf* can be computed from the uv-pair and further leads to the point ID associated to the selected pixel: if the ID is invalid the corresponding index is added to the *group* and only the *texture group* needs to be updated, otherwise the pixel is treated as point. Hence, the `GroupsManager` tries its insertion procedure, which if successful leads to the update of both *subcloud* and possibly *texture group*.

Multiple pixels from colour-based region growing

When the user clicks on a panoramic pixel while being in multiple selection mode, this pixel is used as seed to perform a region growing algorithm producing a set of pixels sharing a certain degree of similarity. The similarity condition is either based on colour variation or on angle deviation between normal vectors. Except that, the general operation of the selection procedure is analogous. The uv-coordinates of the pixel seed, are retrieved as explained previously (see Listing 5.2), these are then signaled to the `Core` that performs the colour-based region growing using the algorithm provided by `MathHelper` and explained in Section 3.4.1. Here, the similarity is evaluated based on the colour distance between two considered pixels. A pixel p_n is inserted into the *region* if the following condition holds:

$$\|p_n - p_{current\_seed}\| \leq threshold$$

where $p_{current\_seed}$ represents the pixel being currently grown. Each performance of the region growing algorithm initially computes the threshold to evaluate the similarity condition, based on the current distance of the mouse from the initial seed. Actually, the `GLWidgets` stores the screen coordinates of the initial seed so that a mechanism of dynamic threshold adjustment based on the mouse motion is enabled. Until the user keeps moving the mouse, a new region growing is performed starting from the same seed but using a different threshold, which becomes more permissive as the mouse moves farther from the initial seed. Each resulting *region* consists of a list of pixel indices which are stored in a temporary *group* similarly to what happens for the result of the intersection between the 3D model and a shape. This *group* is then visualised in the respective panoramic image by temporarily updating the content of the mask texture used by the `quad` shader to blend the actual image with the content of the *groups*.

```
1  void GLWidget::mousePressEvent(QMouseEvent* _e)
2  {
3      // ...
4
5      // Necessary data about the panoramic map in which the selection happens
6      int pmID = this->scene->getPanoramicID();
7      PanoramicView pm = this->scene->getData()->getPanoramic(pmID);
8
9      // Read pixel depth at mouse position from FBO
10     float depth = this->pickDataf(pos, 1, 1, 1)[0];
11
12     // Compute input screen position in object-space using the model matrix of the
           textured quad
13     glm::vec3 modelPos = InputManager::getMouseModelPosition(this->scene->getInputID(),
           depth, pm.base->getModelMatrix());
14
15     // Range conversion [-1,1] to [0,1]
16     glm::vec2 scaledPos( (modelPos.x - (-1.0)) / 2.0, (modelPos.y - (-1.0)) / 2.0 );
17
18     /* Get texture coordinates, in terms of row and column index of the texel
19      * according to the following mapping: [0,1] to [0,nRows] and [0,nCols] respectively
20      */
21     int row = scaledPos.y * (pm.tex->getSize().y-1);
22     int col = scaledPos.x * (pm.tex->getSize().x-1);
23
24     // Signal to Core so that region growing algorithm can start
25     emit pixelSelectionEvent(pmID, glm::vec2(row, col));
26
27     // ...
28 }
```

Listing 5.2: The uv-coordinates of a panoramic image's pixel are obtained through mouse picking. Especially by parametrising the xy-components of pixel's object-space position derived from the screen-space mouse position and the corresponding depth read from the FBO.

When satisfied the user can add the temporary *region* to a definitive *group* of choice through the UI. As for the temporary *group* storing the 3D intersection, the `GroupsManager` performs the update of the *groups*' state before the corresponding *subclouds* and *texture groups* are updated as well. In order to consistently perform all these updates, the list of pixels' indices must first be divided into valid and invalid points. The `GroupsManager` handles both lists: the valid IDs are inserted into the corresponding `Group`'s list according to the algorithm in Listing 5.1 while the list of invalid pixel indices is simply updated with new, non-duplicate items. Similarly is done by the `PointCloudDrawable` when updating the *texture groups* with uv-coordinates and pixel indices separately. The *subclouds* are updated with valid points only.

Multiple pixels from normal-based region growing

The working principle of this multiple selection method is the same as the one of the above-mentioned colour-based selection. The only difference relies in the similarity condition used in the region growing algorithm. For normal-based region growing in order to added to the resulting *region* a pixel having normal, $\vec{n}$, must fulfil the following condition:

$$\vec{n} \cdot \vec{n}_{seed} > threshold$$

where $\vec{n}_{seed}$ is the normal of the pixel currently taken as seed.

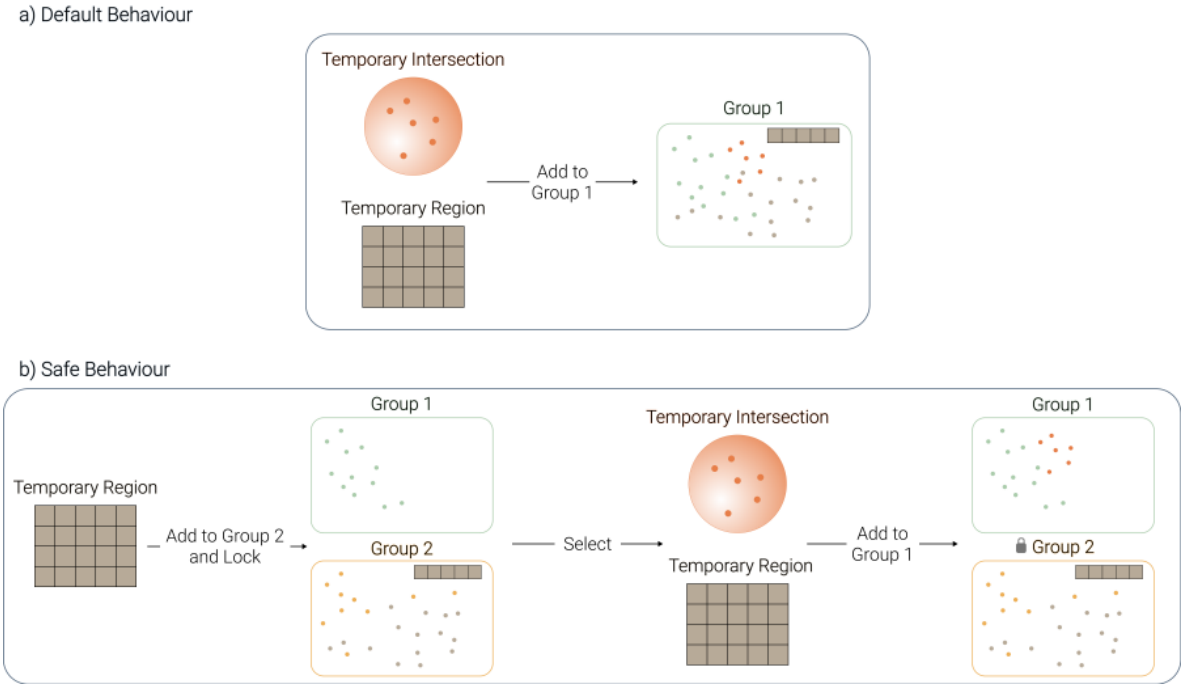


Figure 5.8.: a) When the multiple selection mode is active for both points and pixels at the same time, the chosen *group* is updated with the content of both, non-empty temporary *groups*. b) Such behaviour can be prevented by performing a first selection and adding the result to a *group* which is then locked. Afterwards, a second selection can be performed and safely added to the desired *group*.

The threshold is set dynamically based on the mouse motion, as for the colour-based region growing technique. The threshold is defined as the cosine of the maximum angle, α, allowed between the normal vectors of two pixels, for them to be considered similar enough to be part of the same *region*.

Concerning the derivation of the necessary data, given a pair of uv-coordinates, the one-dimensional index of the pixel within the list in the corresponding *Pmf* is given by $v * n_{rows} + u$. Once obtained the pixel, information such as colour and relative point ID can be accessed and from a valid point ID, geometric properties such as the normal vector can be retrieved.

Before concluding the section about the selection modes, few global considerations are mentioned. Every time a specific multiple selection mode is left, the corresponding temporary *group* is cleared as well as their visualisations, namely the selection buffer and possibly the mask texture.

Moreover, the activation of the selection modes is independent between the geometric and the panoramic scenes, meaning that, for example, the single point's selection may be the only active mode but it can also be simultaneously active to a multiple pixels' selection mode. This leads to a non-immediately intuitive behaviour of the "Add" buttons associated to the *groups* in the corresponding panel. Therefore, the user should be aware that when a multiple selection mode is active for both points and pixels, the trigger of the "Add" functionality for a definitive *group* will result in the insertion of both, non-empty, temporary *groups*. This is prevented to happen though, if the user previously added the current content of a temporary group to another definitive group, which is locked during the execution of the "Add" procedure. Figure 5.8 provides a visual explanation of the two scenarios.

Parameter	Value
Epsilon	1% of bounding box
Cluster-epsilon	1% of bounding box
Normal threshold	cos(25°) = 0.9
Minimum number of points	1% of total
Probability	0.05

Table 5.1.: The algorithm for the segmentation in the*PointCloudAnnotator* is provided by CGAL[27], which specifies the reported default values for the parameters, if these are not explicitly set by the user.

5.5.3. Geometric segmentation

As mentioned in Chapter 1, sometimes it is useful to associate a *group* of points with the primitive shape best approximating their form. In Section 3.4 a few methods to perform segmenting operations have been introduced. This section, particularly, focuses on the technique of 3D shape detection, which is applied in two different manners within the *PointCloudAnnotator*.

Global segmentation

A coarse segmentation of the point cloud may be useful in the early stages of an annotation process. Automatically performing a planar segmentation on the entire model will detect an amount of planes describing various clusters of points. The result is influenced by the parameters the user can set in the UI.

Going into more details regarding the implementation, this functionality is provided by the `segmentByPlanes()` method in `MathHelper` and is triggered by the `Core` upon user's request. The method receives the PCL-point cloud structure of the model and a set of parameters to apply. The core functionality, `Efficient_RANSAC`, is provided by CGAL and its working principles are explained in Section 3.4.2. In order to use it, a few preparation steps are necessary. First, the point cloud must be stored in a CGAL-specific structure accepted by the `Efficient_RANSAC` but also the parameters need to be properly formatted into a dedicated CGAL-object. When the user does not specify any particular parameter, the default values in Table 5.1 are used.

The shape detection is set to look for planes only and produces a list of *shapes* found as good approximations of point cloud's clusters according to the specified parameters. Afterwards, the resulting list is iterated to extract each shape. By extracting it is meant to store the list of inliers, as point IDs, and the coefficients describing the primitive, see Table 4.1. The list of extracted planes are used by `Core` to create respective definitive *groups*. This process is performed in collaboration with `GroupsManager` – which updates the current state of the *groups* – `GeometryScene` and `PointCloudDrawable` for the creation of *subcloud* and *texture group*, respectively.

Local segmentation

In addition to plane models, the `Efficient_RANSAC` is able to detect other predefined types of primitives, namely sphere, cylinder, cone and torus. The *PointCloudAnnotator* thus features the possibility to perform a user-supervised fitting of such shapes at a specific location in the global model. The *groups* selected in the panel are taken as input by the `Core` which combines all the points into a single list. Then it iterates over the enumeration of available shapes and for each the execution of

`MathHelper::segmentByShape()` is requested on the input union of n points. This function performs the `Efficient_RANSAC` set to search shapes of the currently considered type, using the default or user-defined parameters. To increase the accuracy, the actual shape detection is performed three times. Each time a set of shapes is detected, eventually, the one having the best coverage – in terms of points assigned to any shape with respect to the total amount of input points – is selected for the next stage. Here, the shapes are iterated to determine the best fitting one, which has lowest average geometric error. For each point, p_i, the geometric error is computed as the squared Euclidean distance from the shape, S, which on average is defined as:

$$\frac{1}{n}\sum_{i=1}^{n} dist(p_i, S)$$

The best fitting shape is then investigated to extract the primitive coefficients, the point IDs of the inliers, its coverage – defined as $\frac{n_{assigned}}{n_{total}}$ – and the per-point geometric error.

The result of each execution of `segmentByShape()` is used by the `Core` to create five temporary *groups*, one for each type of shape, consisting of an entity held by the `GroupsManager` in a dedicated list and a respective temporary *subcloud* for the visualisation in the 3D scene.

The temporary *subcloud* can be viewed either in plain colour or so that the geometric error is visually enhanced. In the last case, each point is coloured using its single geometric error, computed in `segmentByShape()`, mapped to a meaningful colour ramp. The error values are stored in the colour buffer of the *subcloud* and are thus accessible by the `groupCloud` shader when needed.

The user can decide to accept any of the resulting shapes from the local fitting by the provided UI button. If so, the chosen temporary *group* is converted into a definitive *group*, which contains the inliers of the respective shape. Since the `GroupsManager` inserts the points according to the algorithm in Listing 5.1 to prevent intersections, the inliers are automatically removed from the input *groups* for the fitting, which thus remain with the outliers only, unless any of these input *groups* is locked. In such case, the *group* does not loose any point but on the other side the accepted *group* will be incomplete.

As a general remark, the motivation for the adoption of the RANSAC-based segmentation method relies in the assumption that the input cloud does not perfectly fit a given shape. This is evident in the case of the local segmentation, where the input for the segmentation results from a previous selection of points by the user, which features a high degree of freedom and thus increases the probability of including points that do not belong to the best approximating shape, so-called outliers. In fact, enabling a method of selection which already discards the outliers would be too expensive and time-consuming even though a direct fitting on the selected cluster would then be possible. Hence, the RANSAC procedure is convenient when outliers are present since it actuates random tests to determine the best candidate shape before performing the actual fitting.

5.6. User Interface

The default overview of the *PointCloudAnnotator* after having loaded the model of a point cloud described by panoramic maps is illustrated in Figure 5.9. When the point cloud model is derived by a PLY file, the UI composition will miss the tabs with the panoramic images. Concerning the communication mechanism between UI and the engine core, usually it is the first one which notifies updates to other classes, mainly the `Core`, in response to user inputs. However, sometimes the reverse may happen. After operations such as the global segmentation or the boolean union, which produce

Figure 5.9.: The UI of the *PointCloudAnnotator* shows the 3D model on the left and the panoramic images on the right, when the loaded point cloud data is originally described by depth maps.

definitive *groups*, it is the `Core` that must send Qt signals to the `pclAnnotator`, the class in charge for the UI logic and appearance, to trigger a visual response in the UI.

5.6.1. Visual Representation of the Point Cloud Model

On the left side of the screen the point cloud model is represented through its 3D representation. Per default the camera is of type arcball, meaning that dragging the results in a rotation of the model around the target position. The combination "Cmd/Ctrl + C" allows to switch the camera type, from arcball to fly. In the latter, dragging the model while holding the left button results in panning it within the xy-plane parallel to the camera near plane. In both cases, scrolling controls zooming.

On the right side, a panel contains a tab for each panoramic image. Each image can be zoomed in or out and panned with the same commands for the geometric model. Here, the arcball camera is not available.

If both, geometric and panoramic, visualisations are available, any can be hidden but at least one representation of the model is always visible.

Switching between modes and the selection of functions is accelerated by keyboard shortcuts associated with the most frequently used options.

5.6.2. Menu Bar

The menu on the top bar, schematically represented in Figure 5.10, provides various functionalities subdivided into five categories.

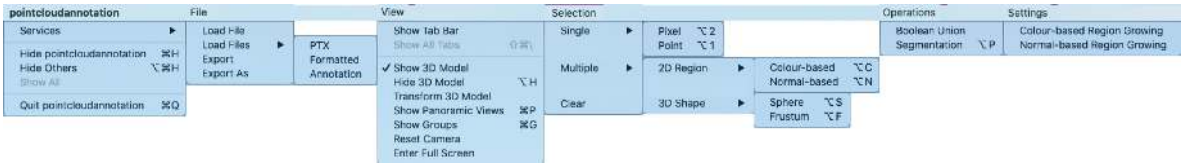


Figure 5.10.: The functionalities of the *PointCloudAnnotator* are subdivided into five categories, each corresponding to a submenu of the main top bar menu.

File Menu. The loading and exporting of files is provided by this submenu. The options for loading are either a single file, PLY, or a group of files, namely PTXs, toolkit-specific formatted files (either the complete packet or the geometric file(s) only), or single pair of annotation files.

View Menu. This set of functions controls the composition of the UI by hiding or showing specific elements. "Show 3D Model" toggles the visualisation of the 3D scene while "Show Panoramic Views" the visualisation of the panoramic images. Enlarging one of the representations allows a better working environment for the annotation as details can be detected more easily. "Show Groups" toggles the display of the control panel for the *groups* representing the current state of the annotation. Sometimes it may be useful to hide the main point cloud model in the 3D scene and visualise the *groups* only, this is the purpose of "Hide Main Pointcloud". "Clear" permanently removes all the definitive *groups* from the current session. However, the annotation state remains unchanged in the already exported annotation files until the user decides to overwrite their content using one of the export functions. When the orientation of a loaded point cloud is not consistent with the up direction of the system, a rotation can be applied to the model via "Transform 3D Model".

Selection Menu. The six selection modes presented in Section 5.5.2 can be activated within this submenu: two "Single" modes – "Pixel" and "Point" – and four "Multiple" modes – "2D Region" (*i.e.,* "Colour-based" and "Normal-based") and "3D Shapes" (*i.e.,* using either a "Sphere" or an orthogonal "Frustum").

When any of these modes is active the mouse inputs are handled differently than usual. Indeed, they do not control the motion of the model or panoramic image, but rather trigger a selection event. In single selection modes, the click results in the selection of the point or of the pixel. While the multiple selection mode for 3D points is active, the user can place the sphere in the scene by clicking on a point of the main model or draw the initial quad to build the intersecting frustum. The sphere can be traslated in xy-plane parallel to the near plane of the camera by holding "Alt + Cmd/Ctrl" while moving the mouse and pressing the left button. Along the forward direction of the camera, the sphere can be moved by scrolling with the mouse while holding "Alt + D". The frustum cannot be moved from its initial position but it can be "Reset" and created at a new location. Sliders in UI enable a volume change of these shapes: for the sphere a slider controls the radius while for the frustum its depth is increased or decreased.

A click of the mouse while a multiple selection mode for pixels is active, sets the initial seed for the region growing. Until the left button of the mouse is not released, the region growing is repeatedly executed with a threshold varying with the distance between the mouse cursor and the initial seed, as explained in Section 5.5.2. Releasing the mouse allows the user to start a new region growing at a new pixel.

In order to move the camera as by default behaviour, while any of the selection modes is active, the user needs to hold "Cmd/Ctrl" while performing the motion commands with the mouse.

Operations Menu. *Groups* can be merged together into one single *group* by means of a "Boolean Union". The resulting *group* will have a label indicating the names of the source *groups* but it will

not be associated to any primitive regardless of whether any of the source *groups* was associated with a primitive or not. "Segmentation" opens the panel to execute the global and local fitting operations discussed in Section 6.5.1. The detailed composition of the panel is presented in Section 5.6.3.

Settings Menu. As briefly introduced in Section 5.5.2, some parameters can be customised to improve the dynamic adjustment of the threshold used by the region growing algorithms. The range of threshold values can be defined here. For the "Colour-based Region Growing" the values to set represent the range of the colour distance between pixels, while for the "Normal-based Region Growing" these are the limits of the angle variations between the normal vectors.

Per default the colour threshold is set to $[0, 255\sqrt{3}]$ while the angle for the normal threshold is included in range $[0°, 90°]$. When the growing algorithm is executed, the specified threshold range is mapped to the constant range $[0, D]$ representing the distance between the current mouse location and the initial seed, where $D = \left\| \overline{(n_{cols}, n_{rows})} \right\|$ is the diagonal of the panoramic image.

5.6.3. Control Panel

On the far right side, control panels will be displayed when necessary. The panel for the management of the *groups* – Figure 5.11.b) – contains the overview of these by representing the properties such as label, associated primitive and the size expressed by the amount of points. This includes both valid points and invalid pixels. Hovering on the primitive name displays the relative coefficients as a tooltip. Per-*group* interactive functionalities are also provided:

- Colour box: represents the current colour in which the *group* is rendered in the geometric and possibly panoramic visualisations.
- "Lock" button: used to lock and unlock the *group*.
- "Select" button: when selected the *group* is shown in the visualisations, otherwise is hidden. Selected *groups* are also considered for operations such as insertion of single point or pixel, union and local segmentation.
- "Add" button: it performs the insertion of temporary *groups* for the 3D intersection and 2D region growing into the associated *group*.
- "Delete" button: permanently removes the single *group*.

The buttons associated with the last four of these functions are illustrated in Figure 5.11.a).

The semantic label can be assigned by clicking on the respective *group* in the panel and it is possible to either type a new label or selecting an exiting one from the appearing dropdown menu.

New, empty *groups* can be created by pressing the button "Add Group", above the scrollable list of definitive *groups*. Next to it, the user can lock, select, and deselect all the existing *groups*.

Segmenting operations can be performed from the dedicated panel – Figure 5.11.c) – which enables the setting of the parameters introduced in Section 3.4.2, namely ε, cluster-ε – given in meter – maximum normal deviation α – provided by the user as a degree angle – minimum amount of points in each segment and probability – ranging from 1% to 100% . These can be reset at any time. The parameters' section is followed by the section to perform the global fitting and subsequently by the tools to perform the user-supervised local fitting.

"Detect Planes" performs the global segmentation while "Detect Shapes" the local segmentation on the selected *groups*. Once completed the global segmentation, as many *groups* as detected plane primitives are added the *groups*' overview and displayed in the model representations.

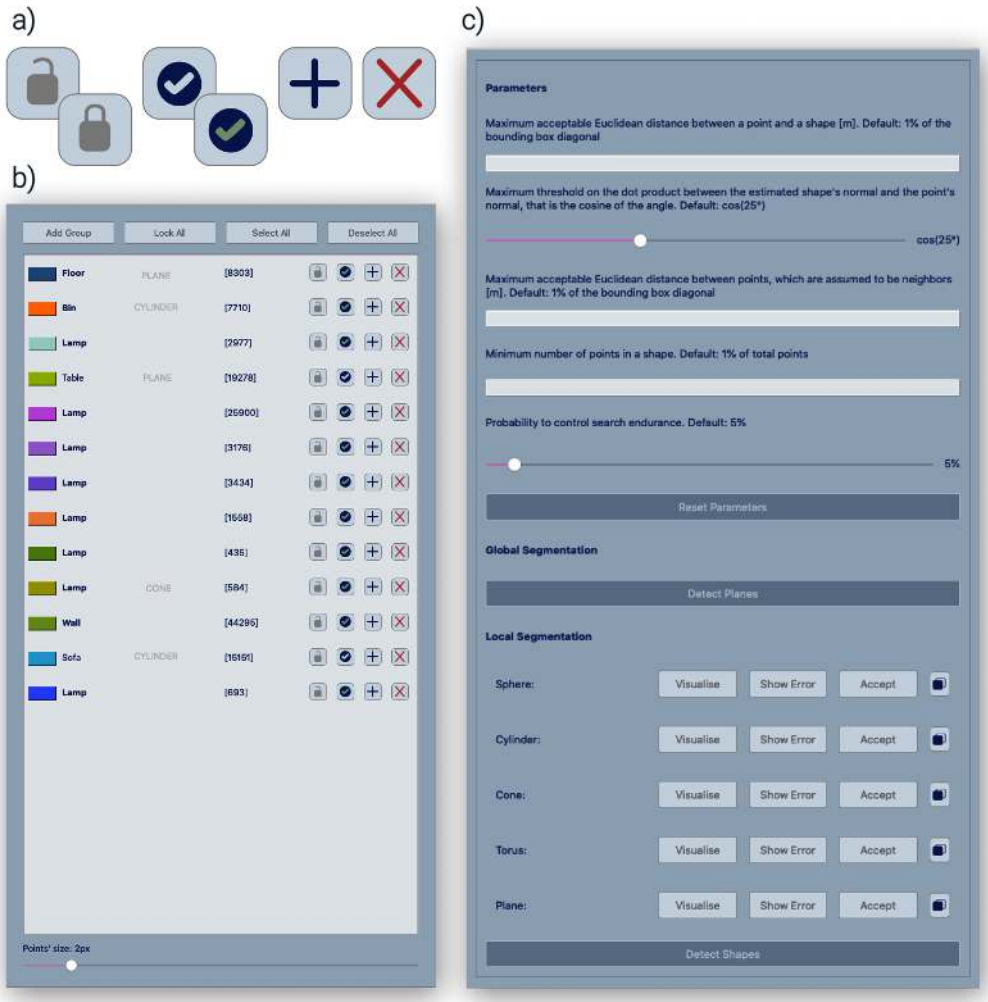


Figure 5.11.: Some interactive functionalities are displayed within UI panels. a) The buttons for *group*-specific functions. b) The panel with the annotation overview and c) the panel to control segmentation operations.

When the local segmentation is terminated, the global percentage coverage – namely the percentage of inliers over the total amount of input points – and average geometric error in meter appear in the section dedicated to the corresponding primitive type.

Here, the user can also find the buttons to "Visualise" the temporary fitting *group* in plain colour, to visually inspect the quality of the fitting by evaluating the colour-encoded geometric error of each point ("Show Error") and to "Accept" the corresponding fitting result by creating a new *group* containing the inliers. The far right button copies the coefficients defining the found primitive to the clipboard so that the user could decide to perform additional visual testing by viewing the considered shape in a specific tool for mathematical rendering. These same coefficients are also displayed inside a tooltip appearing when hovering the fitting summary consisting of the global coverage and average geometric error. All these functionalities are disabled when no primitive of that type has been detected by the RANSAC algorithm.

6. Results

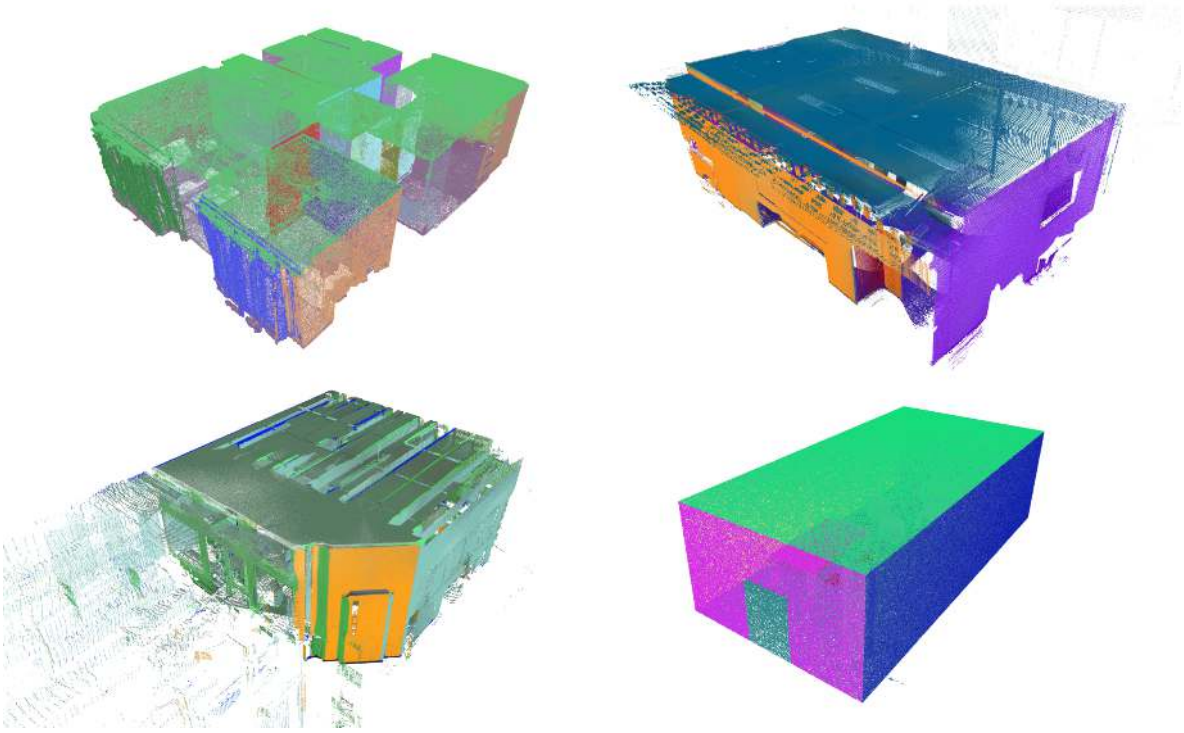


Figure 6.1.: Examples of global segmentations performed within the *PointCloudAnnotator* by using default parameters.

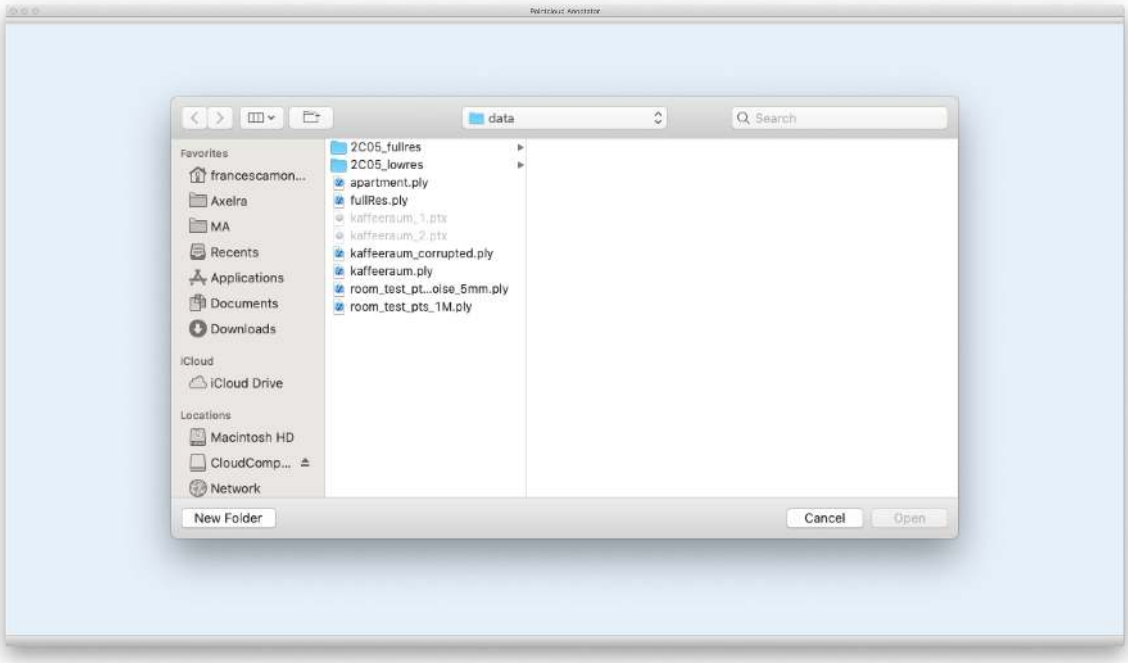


Figure 6.2.: A dialog window appears to select the data files to load.

6.1. File Loading

As explained in Section 5.4.1, there are different options to load point cloud data into the *PointCloudAnnotator*. The respective functionalities can be found in the "File" menu on top. When an option is selected, a dialog window appears, as shown in Figure 6.2.

To open 3D models in the toolkit, single or multiple PLY files can be loaded by selecting the option "Load File". The dialog window allows browsing files with *.ply* extension only, as presented in Figure 6.3.

Point clouds described by panoramic maps are loaded as PTX files by the namesake function, as illustrated in Figure 6.4. In this case, it can be noticed that not only the 3D model is displayed in the toolkit's viewer, but also its associated panoramic maps. Figure 6.5 portrays panoramic images, taken from a laser scan of a given environment, in various viewpoints. All these images combined produce the 3D model, as depicted in Figure 6.6.

In both cases, since the files will be reformatted, the user is asked to define the location to save the resulting geometric files, PCF and possibly PMF files. If left empty, default paths are taken as follows: For PLY, the resulting files are stored with the same filename and location as the original one. For PTX files, the folder is the same, while the default name is the one shared by input files. For example, given the inputs *cloud_1.ptx* and *cloud_2.ptx*, the resulting files would be *cloud_.pcf*, *cloud_0.pmf* and *cloud_1.pmf*.

When data from PLY or PTX files has already been converted to the toolkit-specific format, introduced in Section 4.3, one of the three loading procedures described in Appendix B can be chosen.

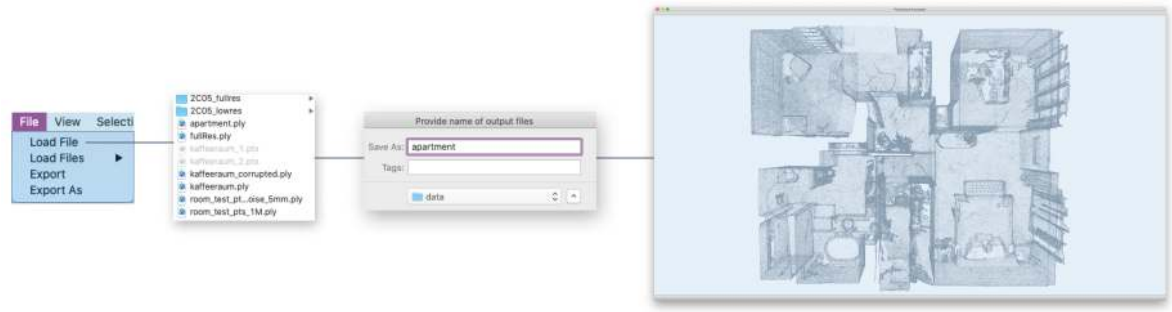


Figure 6.3.: Schematic visualisation of PLY loading into *PointCloudAnnotator*.

Figure 6.4.: The toolkit loads a 3D model, as well as its associated panoramic maps, when multiple PTX files are selected from the loading dialog window.

Annotation files can also be individually loaded at any time, as long as geometry is present. This functionality is provided by "Annotations" within "Load Files" menu option, as illustrated in Figure 6.7.

6.2. Export

As introduced in Section 5.4.3, all necessary formatted geometric files are immediately created at the location specified at loading time. By contrast, the annotation files are created as the user exports an annotation's state for the first time, as depicted in Figure 6.8. Two export functions are available: "Export As", which requires defining a location for the yet-to-create GLF and SLF files, and "Export", which automatically overwrites the last stored annotation files. Worth mentioning is that non-labelled *groups* are named using the default label "Unlabeled_ID" – see Figure **??** – as they are imported in the toolkit.

6.3. Groups

Groups in the point cloud representations – both 3D and 2D – are displayed using the associated colour, see examples in Figure 6.9 and Figure 6.10. The colour can be adjusted with help of the interactive coloured box (Figure 6.11).

A *group*'s name is the label which associates a semantic meaning to what the points in that *group* represent. This label can also be modified by clicking in the UI item of the corresponding *group*, as shown in Figure 6.12.

Figure 6.5.: Three panoramic images of the same indoor environment, taken by placing the scanner at different viewpoints.

Figure 6.6.: This indoor environment 3D model is generated from the combination of various panoramic maps – shown in Figure 6.5.

Figure 6.7.: Single pair of annotation files are loadable when a geometric model is present.

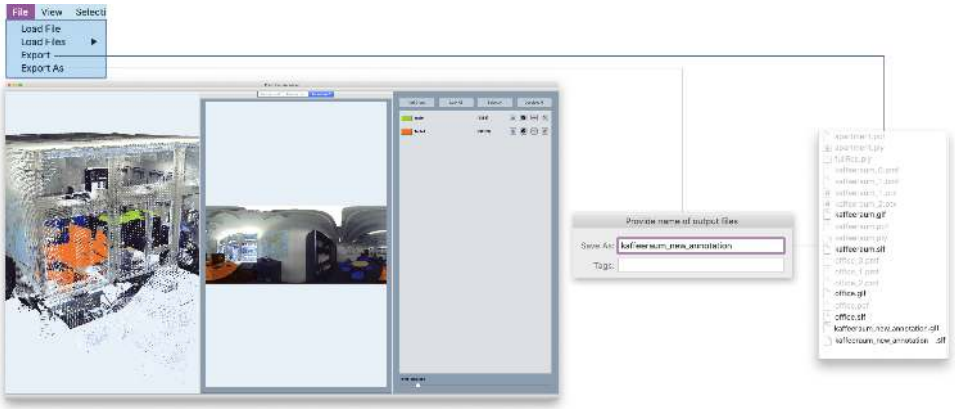


Figure 6.8.: The current annotation's state is exported by either using "Export" or "Export As" at the specified location.

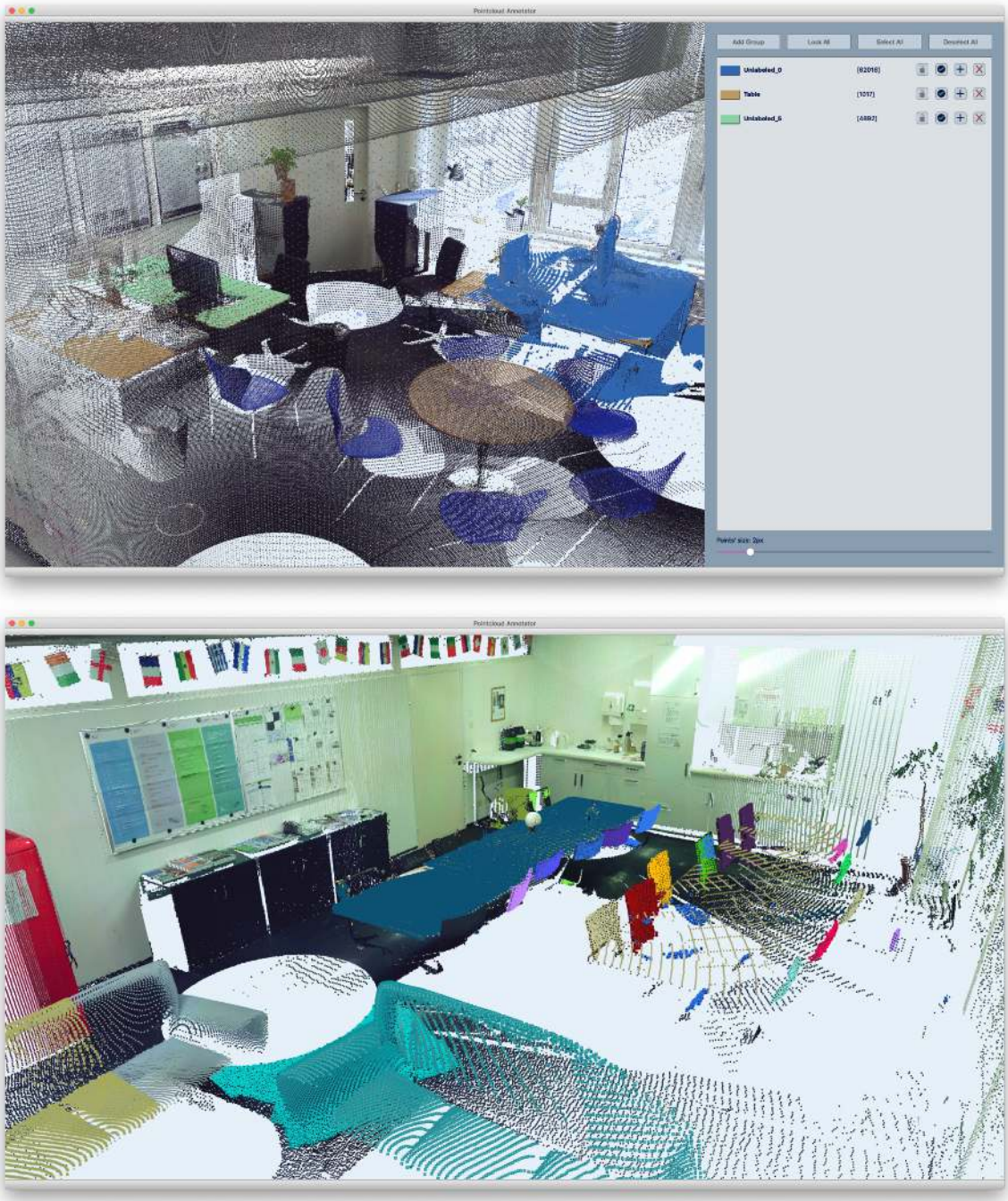


Figure 6.9.: *Groups* visualised within the 3D model.

Figure 6.10.: *Groups* visualised within the panoramic maps.

Figure 6.11.: The colour assigned to the *groups* can be modified by clicking on the coloured box in the corresponding UI item.

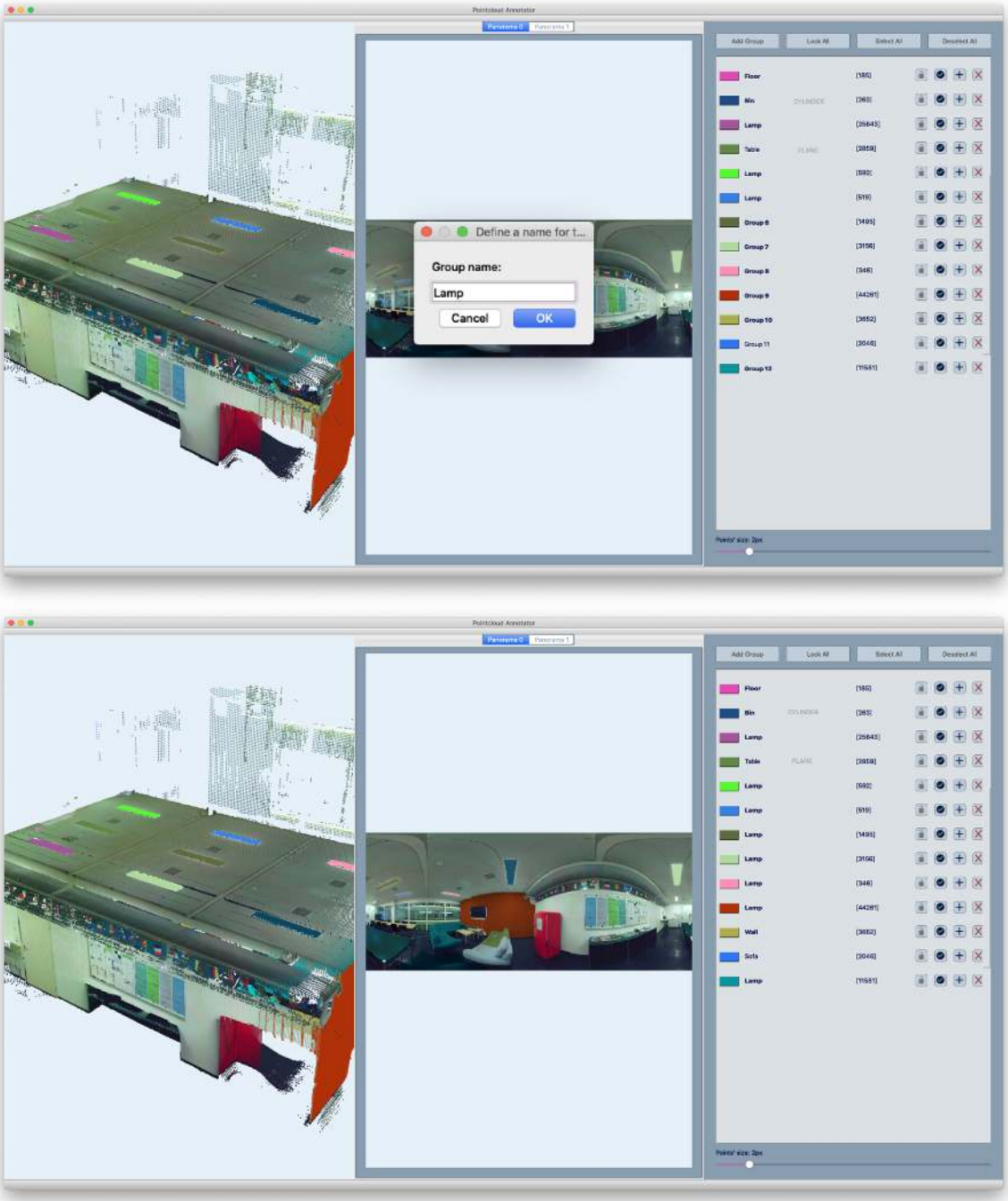


Figure 6.12.: Clicking on an item in the UI list allows the user to define a semantic meaningful name for the *group* or to select one from the currently assigned labels.

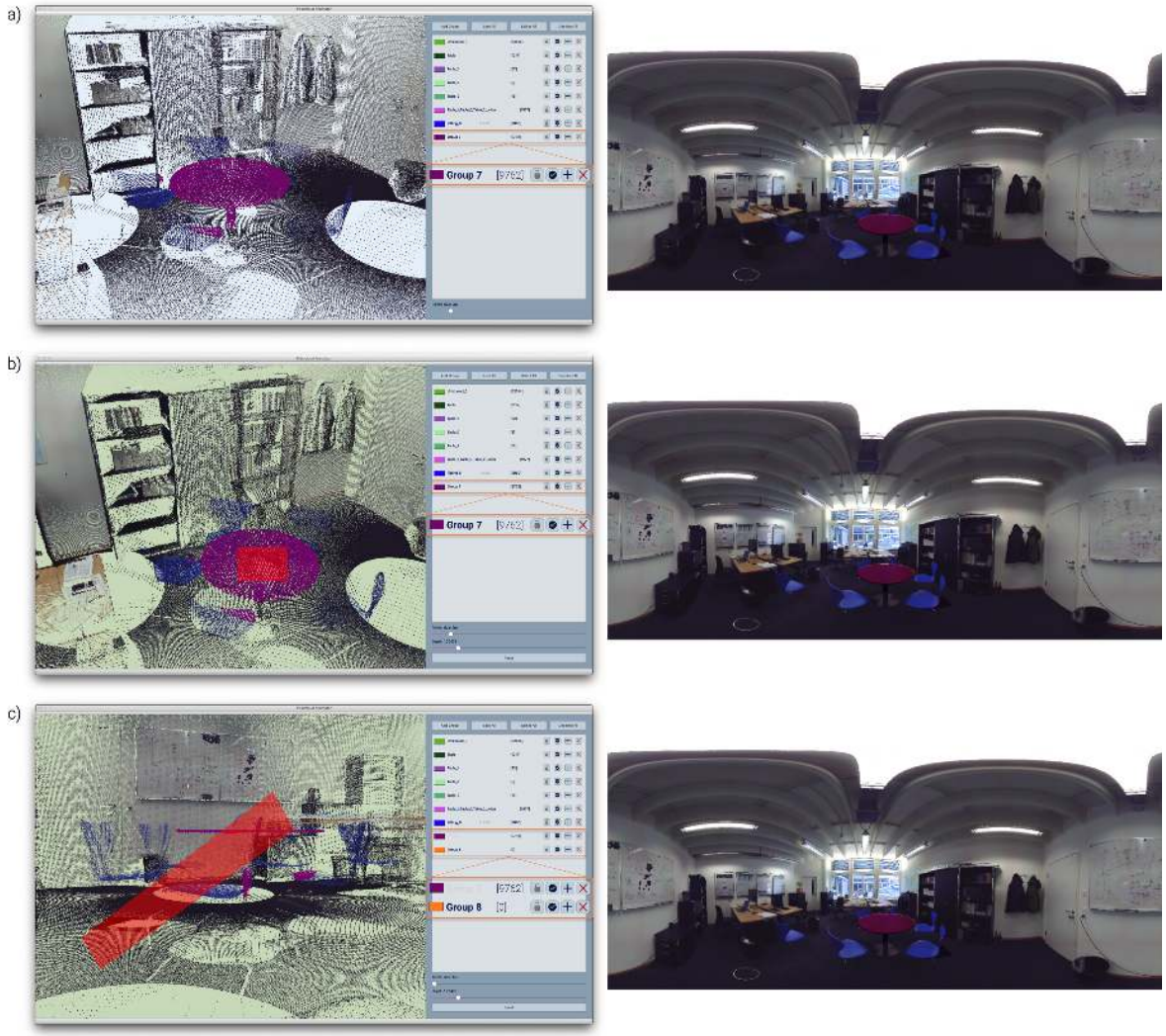


Figure 6.13.: How to preserve a *group* by lock – Part 1. a) Existing *group* A. b) An overlapping set of points is selected with the frustum. c) An empty *group* B is created, A is locked.

As complement to the introduction in Section 5.6, some considerations about the buttons accompanying the UI item for each *group* are following reported. From the left, the first button controls the lock mechanism introduced for the reasons explained in Section 5.5.1. When the "Lock" is active, the *group* points' state are not allowed to change, *i.e.,* no points are added or removed. Figures 6.13 and 6.14 provide a comparison between an insertion operation affecting two overlapping *groups*, once when one is locked and afterwards when the *group* is released. Figure 6.13.a) shows the first considered *group*7. In Figure 6.13.b) a selection frustum intersecting *group*7 has been created. *Group7* is locked and then a new empty *group*8 is created, see Figure 6.13.c). Figure 6.14.d) illustrate how the content of the locked *group*7 remains unchanged, while only a portion of the intersecting points is actually added to the target *group*8. In Figure 6.14.e) *group*7 is unlocked and the same insertion operation of the temporary intersecting *group* in *group*8 is repeated. Figure 6.14.f) and more clearly Figure 6.15 show the result of this operation – performed as both *groups* are released.

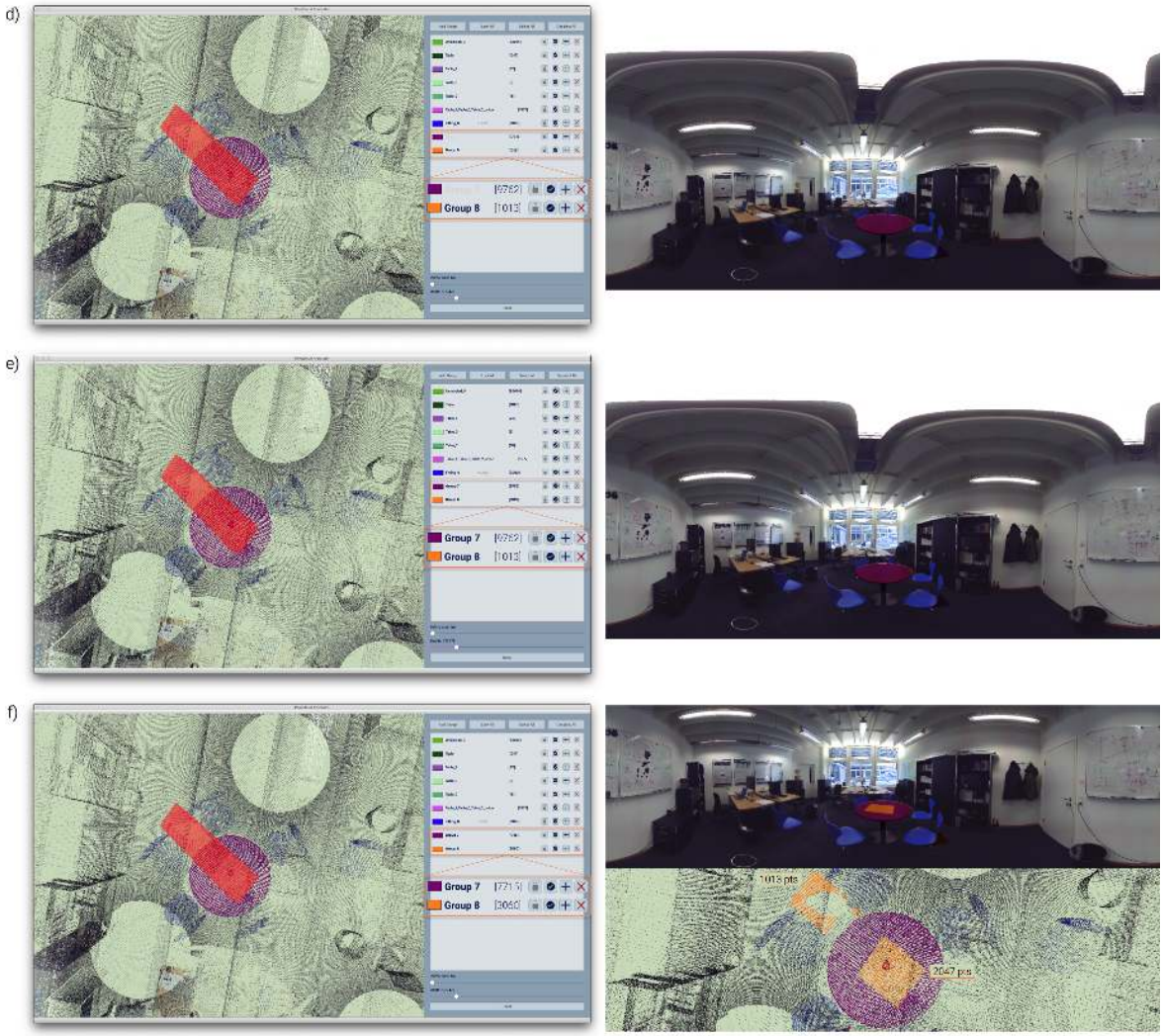


Figure 6.14.: How to preserve a *group* by lock - Part 2. d) Add the temporary *group* to B so, as a result, only the 1013 non-overlapping points are added to B. e) A is unlocked. f) The same temporary *group* is added again to B, but this time the entire intersection is successfully added to B.

In this case, the overlapping points' set within the temporary *group* – that previously could not be added to the target *group* since belonging to *group*7 – is removed from the unlocked *group* and successfully inserted in the target *group*8.

The second button is the "Select" button to affect visualisation and indicate which *groups* are considered in local operations. For example, two or more selected *groups* can be merged into a new, single one. Such operation can be useful to precisely select an object through the panoramic images.

Considering a desk as an example of object to select, the user may perform a normal-based region growing on the surface from a panoramic image, producing what it seems to be a complete selection of the object. By controlling the resulting *group* in the 3D model, it may be noticed that the object is incomplete. This is due to the fact that the 3D model is a result of the combination of the data from several panoramic maps. Hence, selecting from a single panoramic map does not allow including all

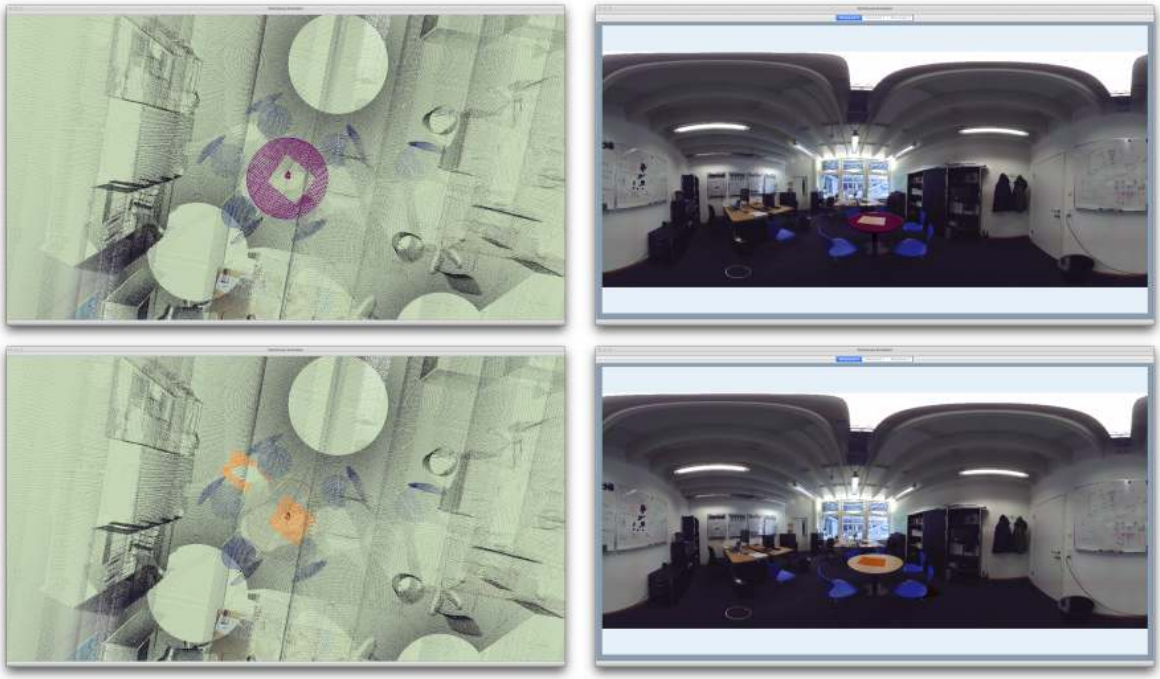


Figure 6.15.: Individual representations in 3D model and panoramic map of the *groups* resulting by the example illustrated in Figures 6.13 and 6.14.

points belonging to the same subject, which was captured form different perspectives. Therefore, the user could select other sets of points by region growing on the related panoramic images and merge the resulting *groups* into another *group*, which contains the complete representation of the object. The main steps of this procedure are depicted in Figure 6.16. It is worth stressing that the "non-intersection" rule (mentioned in 5.5.1) is not applied to pixels that are not associated to any geometry, thus invalid. Therefore, they may belong to multiple *groups* simultaneously. Figure 6.17 shows that after the execution of the boolean operation, the input groups (on top) are shrunk in size, but not completely empty. In fact, they still contain the invalid pixels, artificially highlighted in the lower part of the image. Even though these *groups* are not strictly empty, the export functions will consider them as actually empty, since invalid pixels are not relevant for the annotation.

The "Add" button allows increasing the content of the relative *group* by adding the current content of the temporary *groups*, whereas the fourth button deletes the *group* from the current annotation. Since this operation does not affect annotation files, deleted *groups* can still be recovered, as long as the new annotation's state does not override the respective exported files prior the deletion event.

At the bottom of the panel, a slider enables altering the size of the points. Per default, the size is set to one pixel, but the system allows up to ten pixels for point. This value must be changed carefully, as it considerably affects the performance of the tool. As a matter of fact, large size values immediately drop the display's frame rate. Nevertheless, an increased size of the points can improve visibility, especially in case of "close-ups". The visualisation of the *groups* in 3D can also be enhanced by hiding the main model. Figure 6.18 and Figure 6.19 demonstrate said visual improvements.

Figure 6.16.: Multiple *groups* can be merged into a single *group* by means of boolean union.

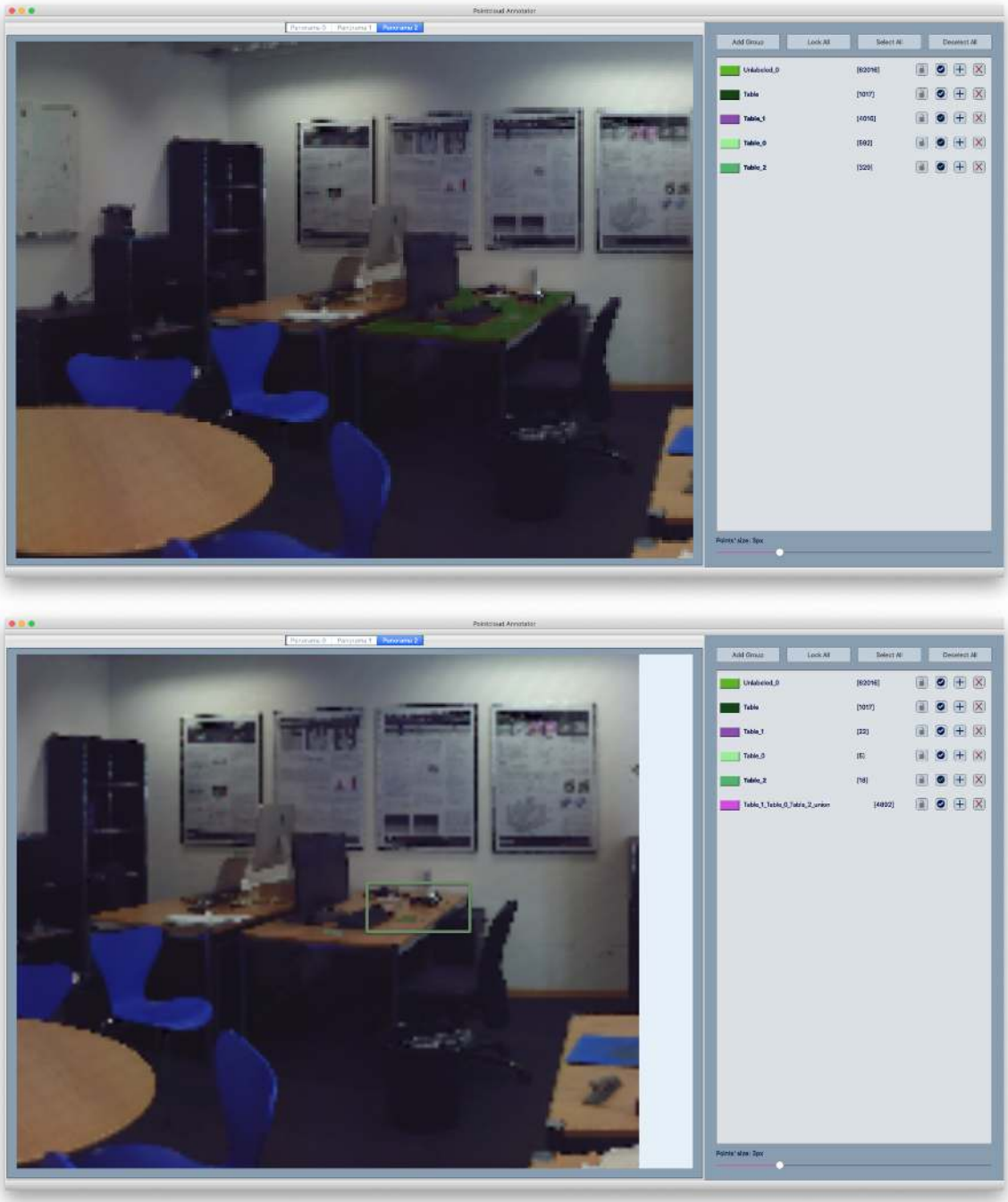


Figure 6.17.: When the input *groups* (top) of a boolean union contain invalid pixels, they remain in the original *group* after the execution of such union (bottom).

Figure 6.18.: A slider in the UI allows dynamically resizing the rendered points. Here, from left to right, the size increases from 1 to 2, to eventually 10 pixels.

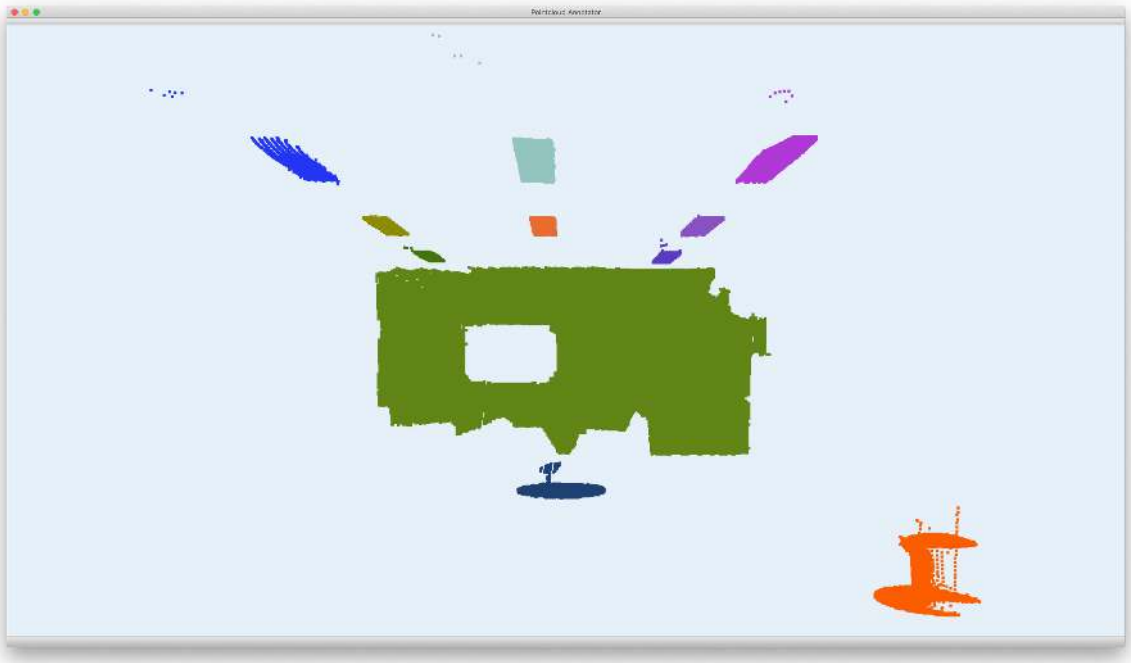


Figure 6.19.: *Groups* in the 3D environment can be highlighted by temporarily hiding the main point cloud model.

6.4. Selection

This section aims to provide snapshots taken while performing the selection techniques explained in Section 5.5.2.

As previously introduced, when exactly one *group* is selected, the single selection mode works correctly. Otherwise an error message is displayed on the screen. When the selection mode is active for the points, the user can click on any point in the 3D model to insert it in the selected *group*. When the same mode is active on pixels, these can be clicked within the image. As a consequence, the selected pixel is added to the chosen *group* within the image and, if valid, to the *subcloud* rendering structure as well. Figure 6.21.b) shows the addition of a point and Figure 6.21.c) of a pixel to an empty *group* – in Figure 6.21.a). For a clear visualisation purpose, the selected pixels in these images are artificially highlighted.

Multiple points can be inserted by means of sphere or frustum intersections with the point cloud. Figure 6.20 illustrates the sphere the user can place within a 3D scene, when the respective multiple selection mode is active. Additionally, traslations in the xy-plane and along the z-axis are shown. An example of selection by sphere intersection is provide in Figure 6.22.

An orthogonal frustum is another multiple points' selection shape the user can use to intersect with the main model. Figure 6.23 shows the process of frustum creation (in red) by drawing a 2D rectangle on the screen (in grey). When entering the respective selection mode, the user can start drawing with mouse click. As explained in Section 5.5.2, the resulting frustum is placed between the depth range created by the 3D points contained in the 2D rectangle. The change in frustum's depth is additionally displayed in Figure 6.23. The selection process by means of frustum intersection is displayed in Figure 6.24.

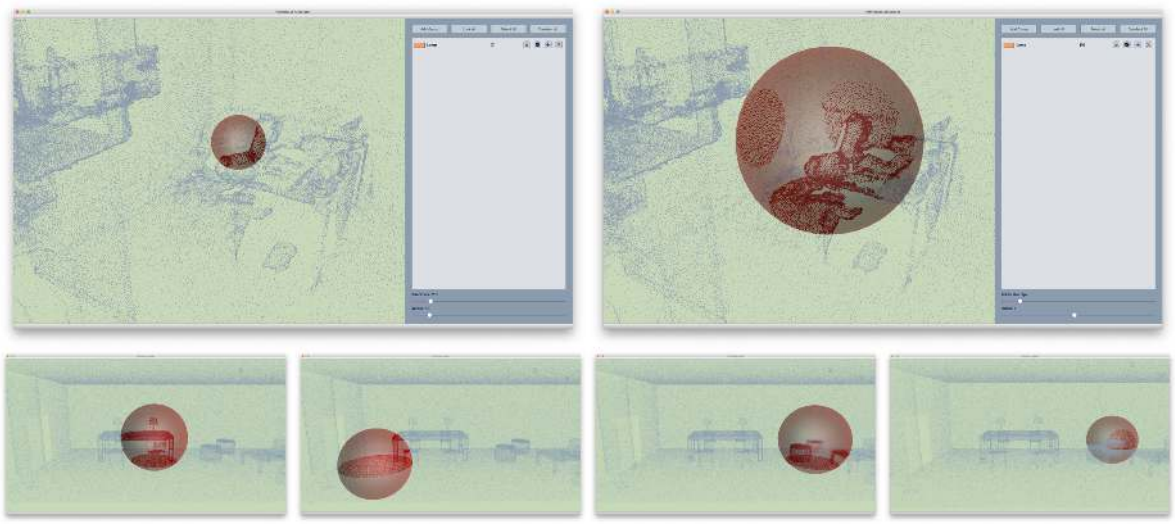


Figure 6.20.: The top images show the resize of the sphere, whereas its motion options are displayed in the bottom images.

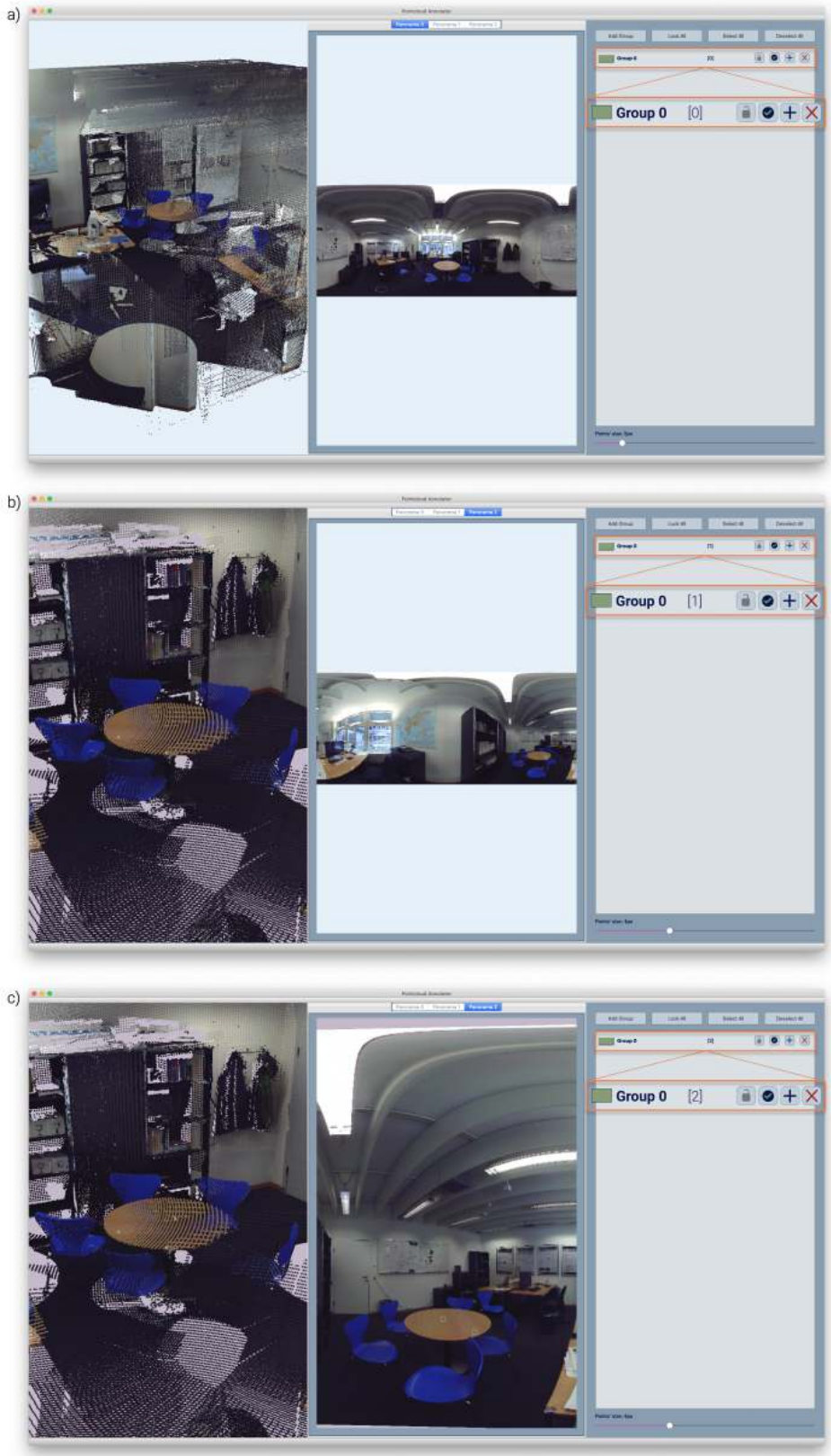


Figure 6.21.: Examples of single selections. a) The considered *group* is empty. b) A single point is added by clicking the 3D model and c) a pixel is added from the panoramic image. Since the pixel is valid, the corresponding point is highlighted in the 3D model as well.

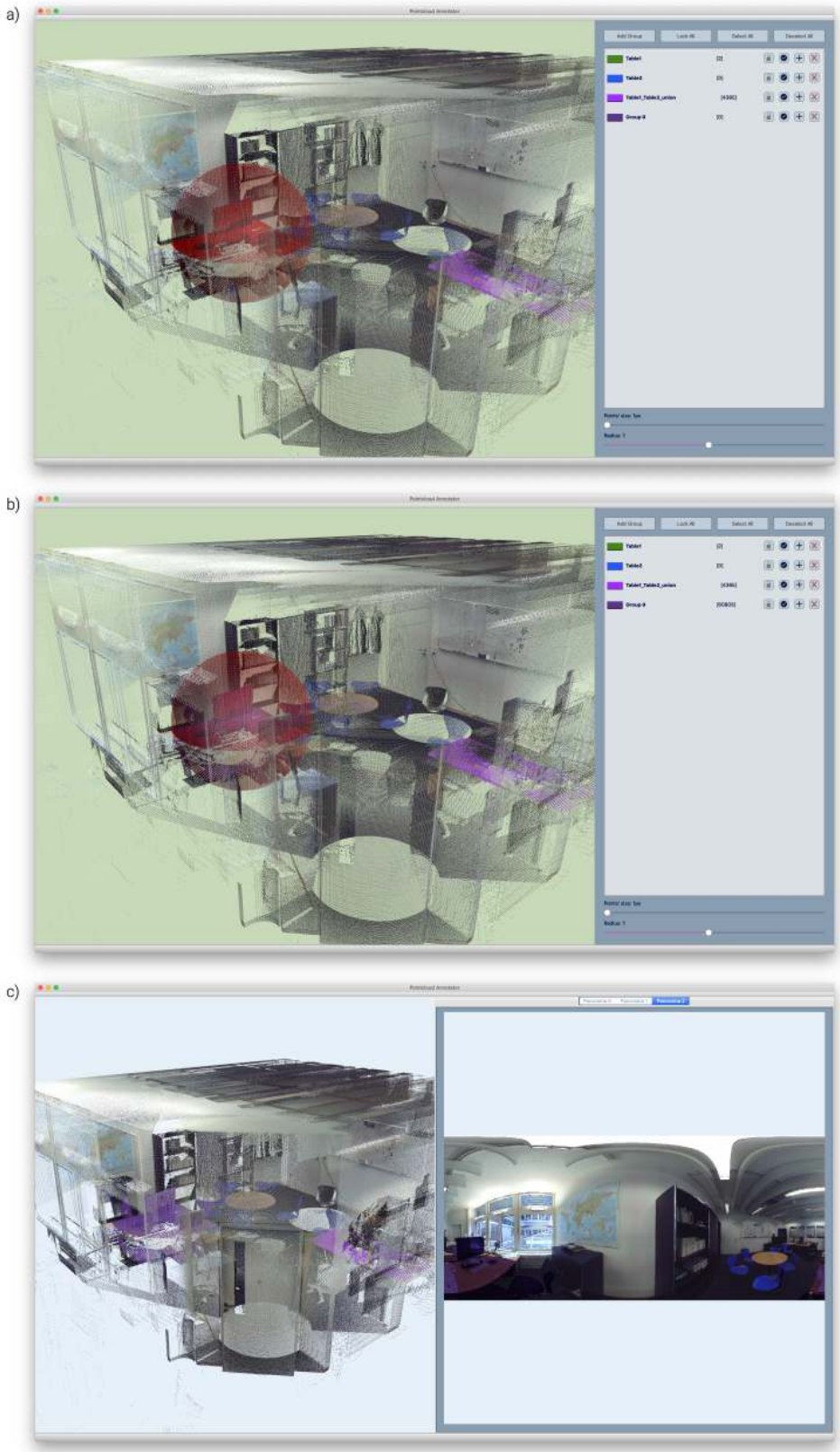


Figure 6.22.: Selection of multiple points by sphere intersection. a) The sphere is placed and the intersecting points in the temporary *group* are highlighted in red. b) The temporary *group* is added to a definitive *group*. c) The previously intersected points are now part of a *group* and thus accordingly coloured.

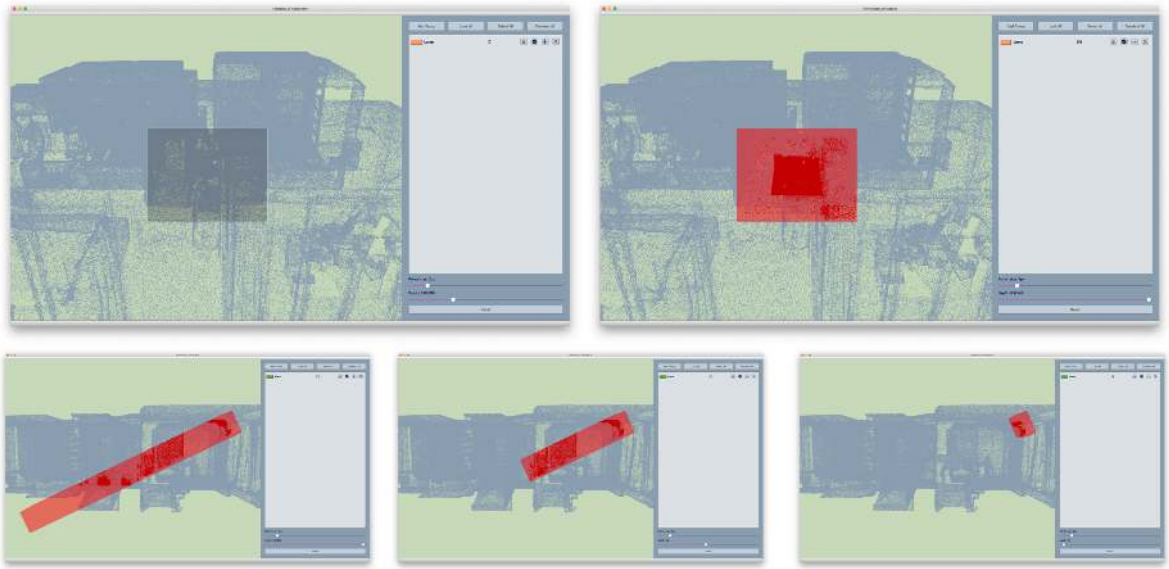


Figure 6.23.: The images on top shoe the creation of a selection frustum (right) by drawing a 2D rectangle on the screen (left). In the lower part changes in the frustum depth are illustrated.

Figure 6.24.: By means of frustum selection (on the right), the user inserts points into a previously empty *group* of choice (on the left in yellow).

Multiple pixels can be selected from a panoramic image as discussed in Section 3.4.1 and Section 5.5.2. The selected pixel is taken as initial seed for the region growing algorithm. In order to be performed, this procedure additionally requires a threshold which is dynamically determined based on the distance between the mouse cursor and the initial seed. The best way to visualise the change of the region growing's result upon dynamic change of the threshold, would consist of a short animation, which unfortunately is not supported by a written thesis. Therefore, Figures 6.25 and 6.26 simulate the animation effect by displaying a sequence of frames taken while moving the mouse far from the initial seed. The two images respectively display the performance of a colour-based and normal-based region growing technique. Similarly to the multiple selection modes in 3D, the resulting set of pixels is stored into a temporary *group* upon release of the mouse button.

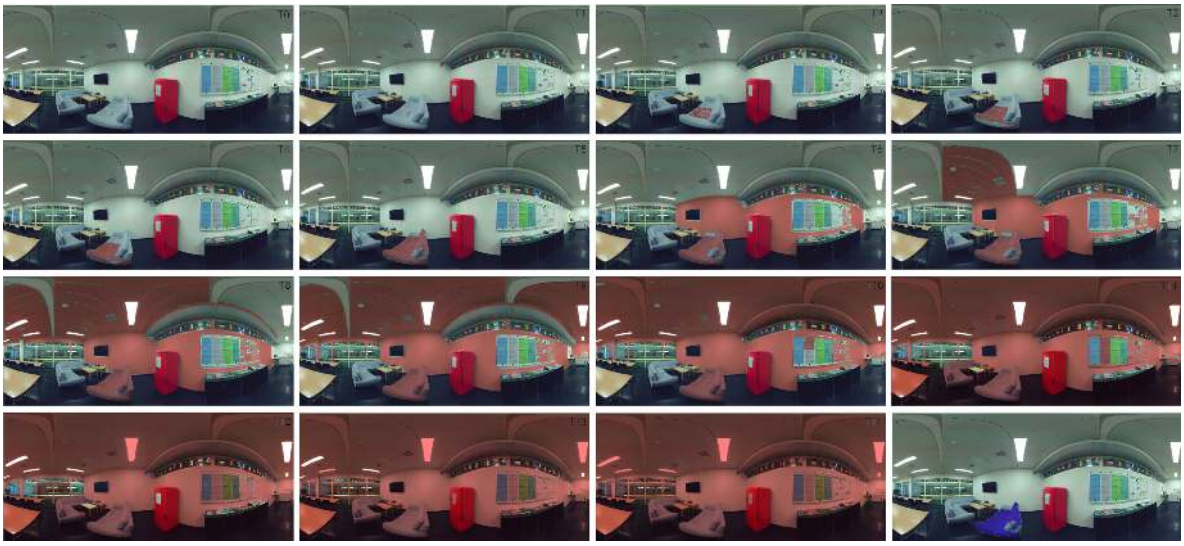


Figure 6.25.: Series of images to show the dynamic selection of pixels by colour-based region growing, which is eventually added to a (purple) *group*. The selection seed is placed on the sofa.

Figure 6.26.: Two examples of normal-based region growing selection. On top the seed is located on the cupboard, while at the bottom it corresponds to a pixel on the ceiling.

Global Segmentation			
ε [m]	1000	1% BB = 0.328381	9749
Cluster-ε [m]	1000	1% BB = 0.328381	9749
Normal threshold [cos(α°)]	1000	0.9	9749
Minimum number of points	1000	1% total = 9749	9749

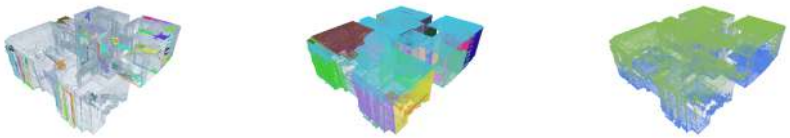


Figure 6.27.: The set of parameters affects the result of the global segmentation. Restrictive parameters increase detection precision, whereas larger coverage is achieved by setting more permissive parameters.

6.5. Segmentation

The user is provided with the possibility to perform either a global segmentation or a local segmentation, which are controlled through the dedicated panel.

6.5.1. Global Planar Fitting

The segmentation performed on the entire data set consists in fitting the points to randomly selected planes as explained in . The result of such an operation is the creation of a *group* for each detected primitive, which is also added to the UI list and visualised in the available representations of the model, 3D and possibly 2D. Figure 6.1 illustrates examples of global segmentations performed on the various test data sets. By testing the same model multiple times we can assess the effect of each parameter. An example of such test is provided in Figure 6.27. A more restrictive set of parameters produces smaller clusters than the default values. By more permissive parameters, smaller shapes cannot instead be precisely detected. However, the loss in precision is counterbalanced by larger coverage, as all the points in the cloud are classified.

The influence of single parameters on the segmentation's result is instead illustrated in Figure 6.28. For each parameter, the global operation is performed once with a value smaller than the default – see Figure 6.27 – and once with a larger one.

6.5.2. User-supervised Local Shape Fitting

The user is also allowed to perform a fitting on a local region of the main point cloud represented by selected *groups*. When no *groups* are selected the system notifies the impossibility of proceeding with a popup window. The result can be refined by changing the parameters taken by the RANSAC algorithm. So far the process is almost automatic as the global segmentation, but the user supervises the selection of the best fitting primitive type. For each, the user can visualise the resulting temporary *group* in plain colour or having the geometric error, in terms of distance from the shape, encoded into per-point colours. Figure 6.29.b) shows the cylinder fitting resulting from the local segmentation of the *group* in Figure 6.29.a). The same result is displayed in Figure 6.29.c) using the colour-coding of the geometric error for each inlier.

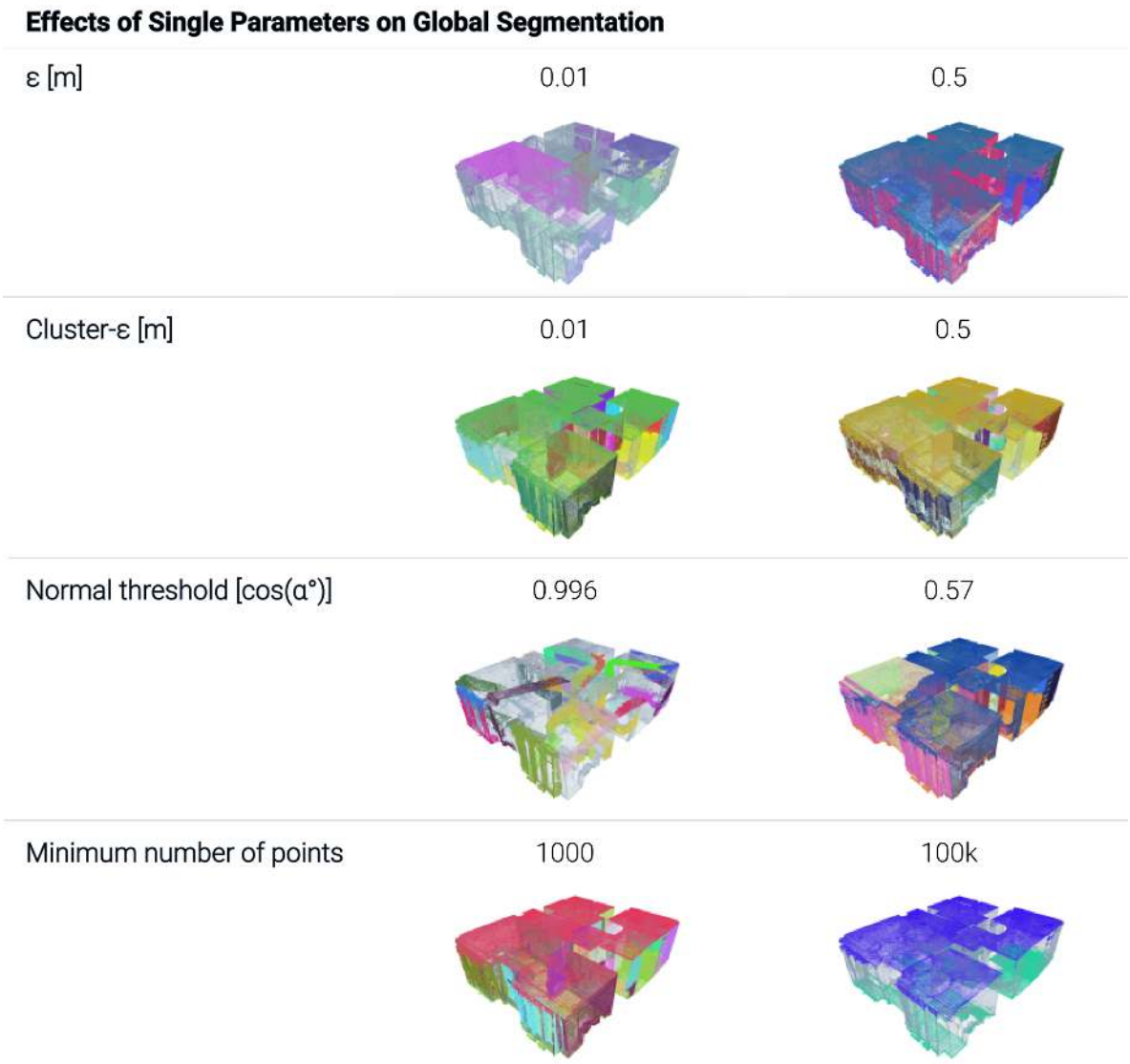


Figure 6.28.: The effect of each parameter emerges by segmenting the same model. Each time only the value of one parameter was changed. The other were set as default.

The accepted *group* is shown in Figure 6.29.d). A comparison between Figure 6.29.a) and Figure 6.29.d) reveals that the newly created *group* does not contain outliers.

Examples of the fitting of each shape type for a given *group* is given in Figure 6.30. Here, default parameters are compared to a set of more permissive values. In most cases non-default values perform better. The impact of each parameter on the local segmentation for each shape type is provided in Figure 6.31. In each test only the considered parameter was changed, the other ones had default values.

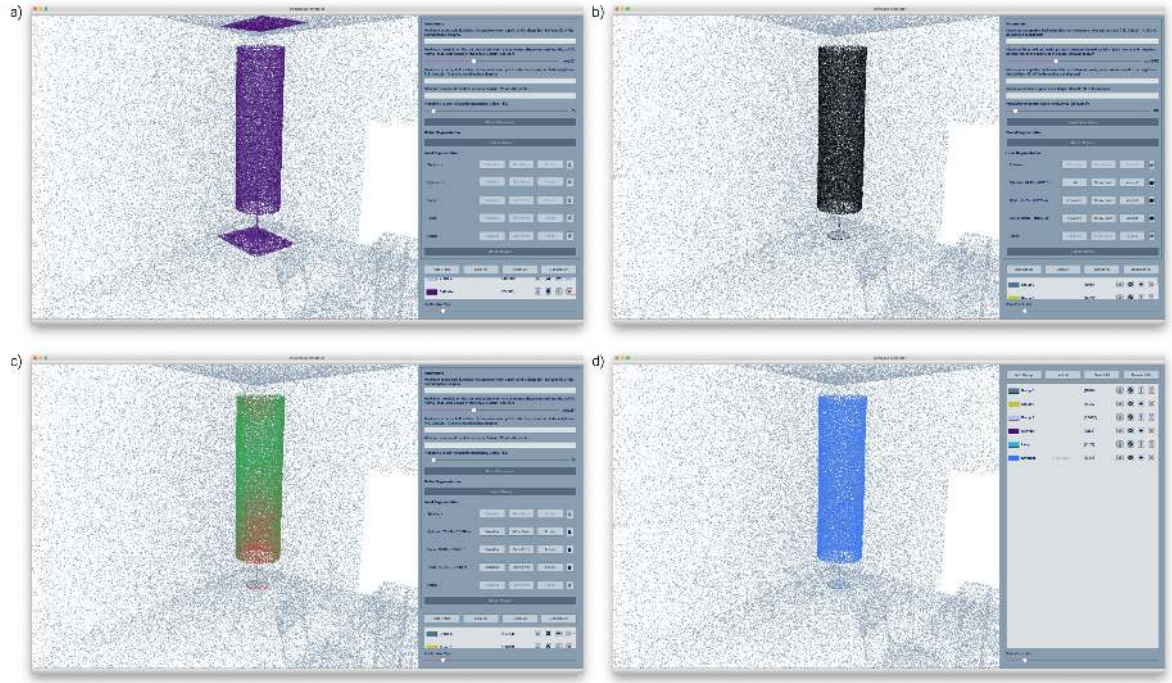


Figure 6.29.: Cylinder shape from a local segmentation on the *group* in a). The fitting result is visualised in plain colour b) and colour-encoding the geometric error c). The accepted *group* is shown in d).

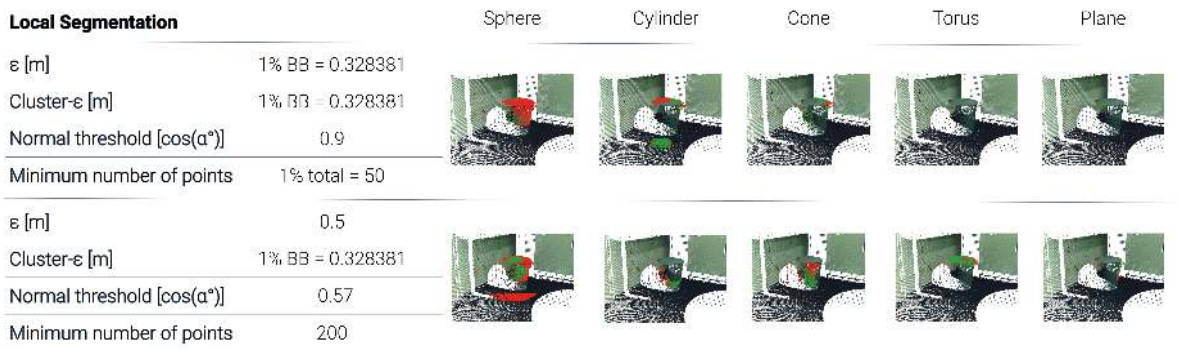


Figure 6.30.: Comparison between default, on top, and user-defined parameters, at the bottom, used on a local segmentation.

Figure 6.31.: For each shape type, a comparison between different parameter values is provided.

7. Discussion

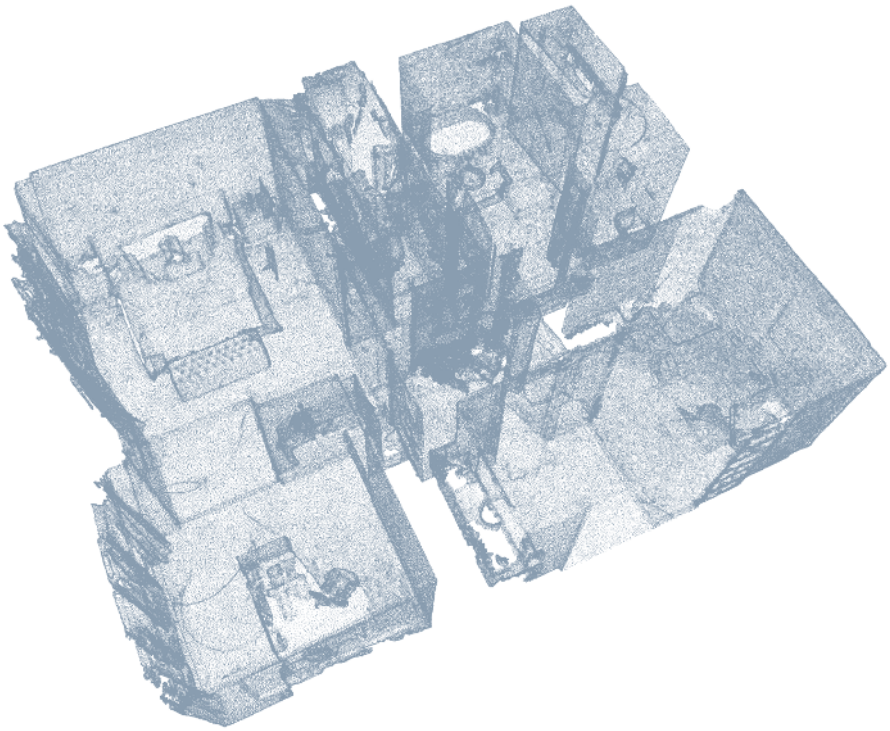


Figure 7.1.: Example of point cloud model originated from a single unstructured list of points.

7.1. Evaluation

One of the major advantages provided by the toolkit presented in this thesis, consists in the integration of the relative panoramic maps to support and improve the accuracy of the annotation process. In fact, the effort and time involved by the selection of certain objects can be considerably reduced when done directly within a panoramic map rather than via intersection of 3D shapes. The selection of a regular table from the 3D scene, for example, would require the combination of at least five frustum intersections, in addition to refinement processes, such as planar fittings, to remove all the outliers. The same result could be achieved much faster by performing a single region growing routine starting from a seed on the table surface. Figure 7.2 compares the result of a table's selection by normal-based region growing (on the left) and by frustum intersection (on the right). The resulting selections are quite similar, the region is less complete than the intersection, but the latter contains more outliers than the previous. But most importantly, the region was obtained with a single operation, while the intersection required placing the camera to have the suitable viewpoint, then creating the frustum and intersecting it, the whole repeated for five times.

Mostly derived from a matter of limited time, some aspects of the application are still subject to drawbacks and could benefit from certain progresses. From a raw technical perspective, the design of the structure can surely be improved by increasing the degree of consistency within the classes. Especially, incrementing the modularity by better allocating the relevant responsibilities to each class. An example in this regard is represented by the management of the *texture groups* that is currently a task of `PointCloudDrawable` but that is probably better suited to the `PanoramicScene` class.

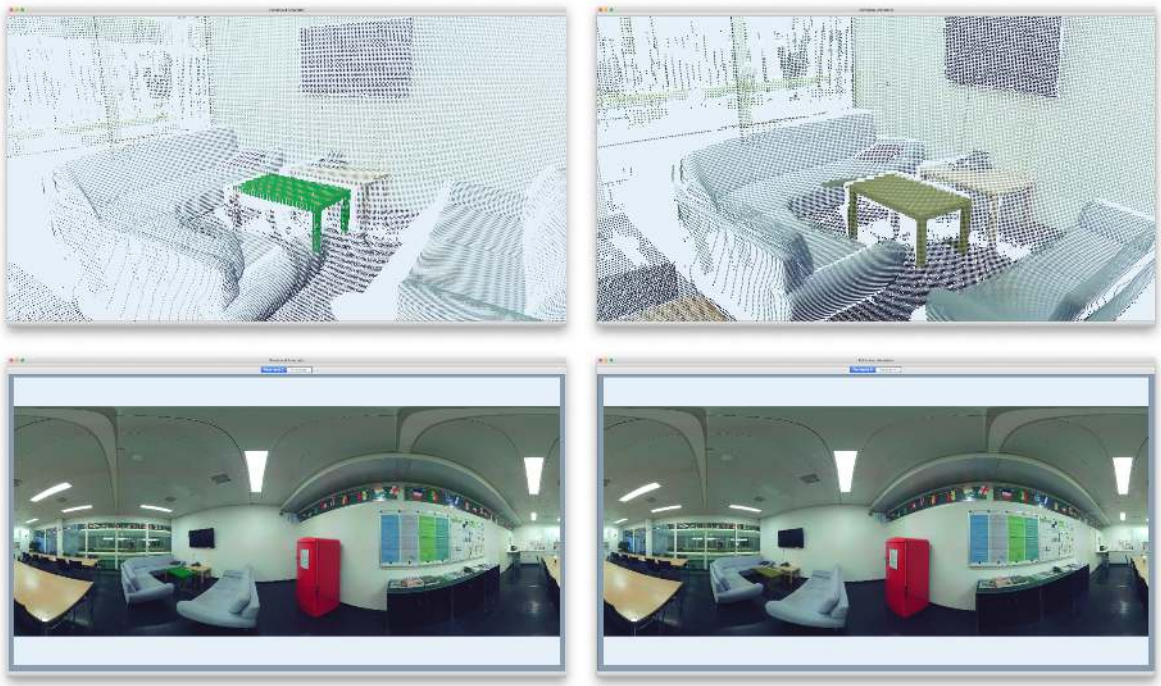


Figure 7.2.: The points belonging to a table have been selected once by means of region growing, on the left, and once as a result of multiple frustum intersections, on the right. The result of each intersection is shown both in 3D and 2D. With almost equal results, as a matter of time, the first method is surely the most efficient.

With respect to the usability, some mechanisms could be improved so that the users' demands are better met. To mention a concrete example, the *PointCloudAnnotator* lacks the possibility to undo an action just performed. Hence, "wrong" points can be removed from a *group* only by performing a local fitting or by intersecting it with a non-locked group. Such a functionality would be of great help in supporting the working flow within the annotation. In this particular case, the issue is partially solved, especially when the user performs an undesired insertion of a temporary *group*. The following situation can be taken as reference: the user just added a set of points resulting from one of the multiple selection techniques to *group* A. They immediately realise that the temporary *group* should have actually been inserted in *group* B. If the temporary *group* has not been changed or cleared (by moving seed or 3D shape, or leaving the selection mode), the same content can be added again but to the correct *group* this time. As a consequence, while being added to B, the points are automatically removed by A (that must be unlocked) in accordance with the "non-overlapping" rule.

Before discussing how the current thesis can be extended, a few words should be spent on the configuration of the fitting operations. The results of which strongly depend on the values chosen for the parameters. This is particularly evident for local fittings, where a careful calibration is needed for optimal results. A large analysis on the factors to consider in order to set the most appropriate values for high quality fittings, has not been conducted as it is out of the scope of this thesis. Nevertheless, some considerations can still be done. The segmentation operations in the *PointCloudAnnotator* are based on the methods provided by CGAL, which are further based on the RANSAC algorithm proposed by Schnabel et al. [34] (see Section 3.4.2). If user-defined parameters are not provided, default values are computed by the CGAL method, as explained in Section 6.5.1. These values already perform good for global fittings while local operations often require some adjustments. Since the input set of points is selected by the user, a collection of shapes too diversified is not to expect as a result of the local fitting. Therefore, the default value for the cluster-ε should be increased. The same should happen for the amount of minimum points, since 1% within an already limited input set is too restricting. The set of values reported in Figure 7.3 have proven to perform well with respect to the CGAL-default values. Here, the results are produced by tests conducted on reference shapes for each primitive type, also visible in the figure.

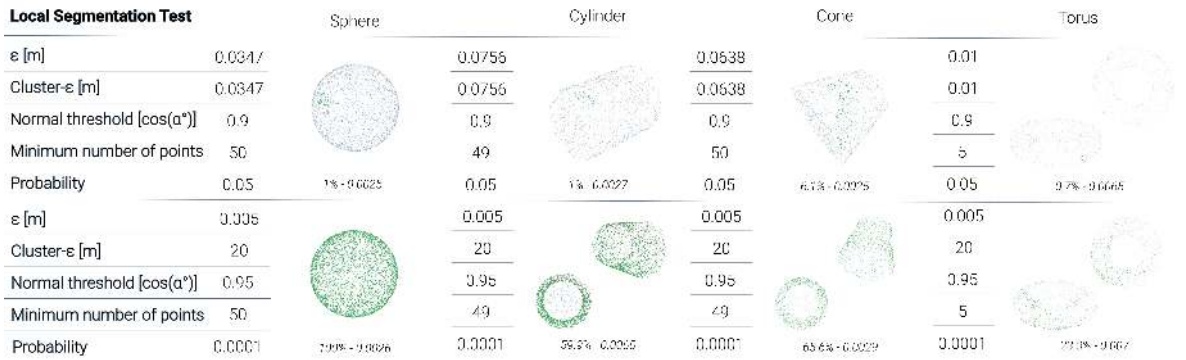


Figure 7.3.: A set of reference point clouds has been used to test the quality of the local fitting segmentation. On top default values computed by CGAL. On the bottom, values that have been found to provide good results.

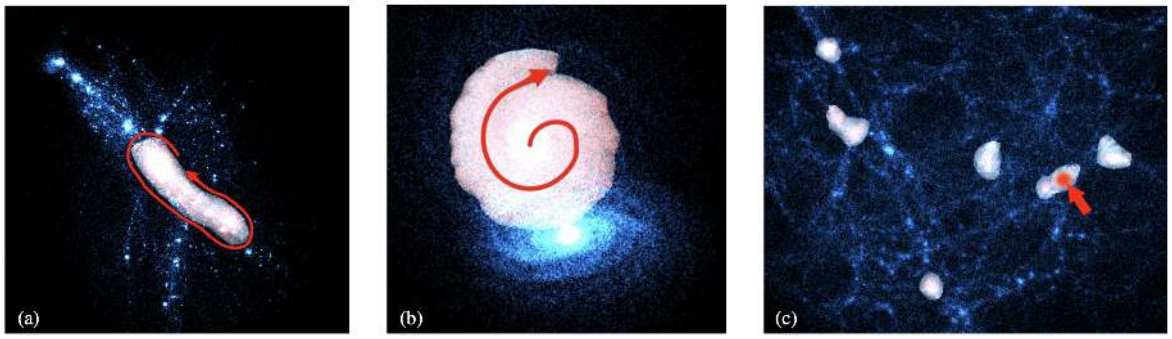


Figure 7.4.: Applications of the three interactive CAST techniques proposed by Yu et al. [40]. a) SpaceCast selects points within a closed lasso, b) TraceCast derives a lasso shape by interpreting the input path an d c) PointCast selects small set of points by a single click. Image courtesy of [40].

7.2. Future Work

The *PointCloudAnnotator* lays the foundations for a number of extensions having interesting applications. The following sections aim to briefly discuss some of these features.

7.2.1. General Improvements

Currently the *PointCloudAnnotator* only supports the elaboration of two standard formats, PLY and PTX. The spectrum of applications could already be widened by extending the set of supported formats, including for example LAS, PCD or TLS. As a consequence the target audience will automatically expand. A further advancement is represented by providing the functionalities – discussed in the previous section – to improve daily usability as the above-mentioned "Undo" or functions as assigning the same label to multiple *groups*, which can speed up the annotation process.

Pas and Platt [30] propose a technique for primitive fitting using general coefficients. In particular, they adopt a Taubin quadric fitting technique which describes the least-squares fit of a quadratic surface in terms of a fixed set of coefficients c_i having $i \in [1, 10]$:

$$f(c, x) = c_1x^2 + c_2y^2 + c_3z^2 + c_4xy + c_5yz + c_6xz + c_7x + c_8y + c_9z + c_{10} = 0$$

where (x, y, z) are the cartesian coordinates of a point on the considered surface. The advantage is that the suggested method of quadric fitting is not limited by the type of primitive and can be generally applied. This approach could represent a good step towards generalisation and could be implemented within the context of segmenting operations. This way, instead of being represented by a specific set of parameters – as radius, axis etc. – the detected fitting primitives will all be described by the same set of general coefficients.

7.2.2. Efficient Lasso-Selection Methods in 3D Point Clouds

A selection technique to increment the efficiency with user interaction with large 3D point clouds is developed by Yu et al. [40]. The proposed context-aware selection is the evolution of a prior work presented a few years before by the same research group [39]. Actually, Yu et al. [40] introduce a set of three interactive methods under the name of Context-Aware Selection Techniques (CAST), which

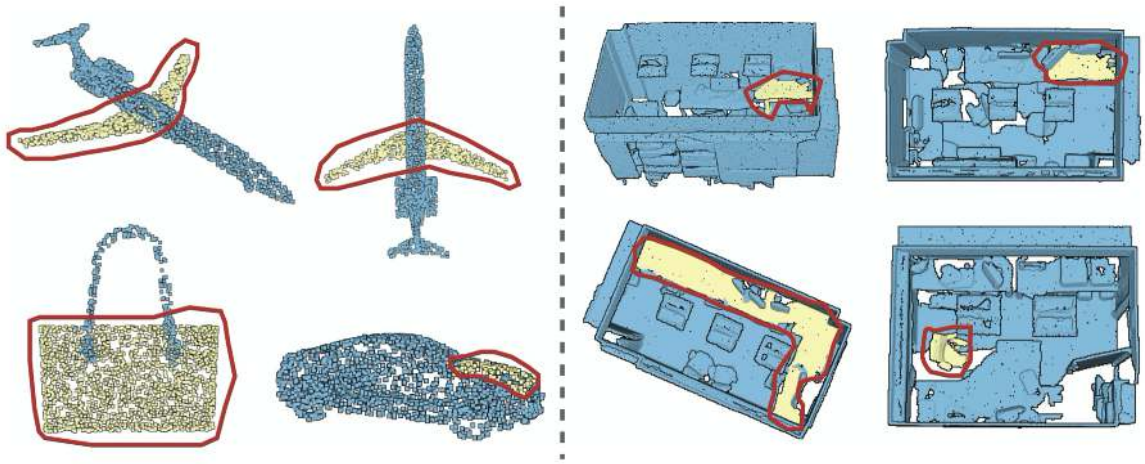


Figure 7.5.: The points in yellow represents some of the selection records used to train the LassoNet. Image courtesy of [11].

are illustrated in Figure 7.4. The spatial selection methods share the ability to derive the intention of the user using context-aware algorithms to define the selection volume from the input gesture. This is accomplished by computing a density field representing the density of each point in the cloud. Despite a common foundation, each method is better suited to different use cases. SpaceCast works with overlapping regions while TraceCast recognises drawn input paths. These methods are adequate when dealing with large, complicated structures whereas smaller structures can be selected using the third technique, PointCasts which implements a more traditional approach of picking along the line of sight, in principle similar to the technique introduced in Section 3.1. An internal user study demonstrated that the proposed interactive methods provide efficiency and effectiveness without sacrificing accuracy, other than being positively valued for their intuitiveness by the users.

A recent very effective technique to select points from a point cloud is proposed by Chen et al. [11] as a deep lasso-selection. Figure 7.5 illustrates the principle of lasso-selection on various point data sets. The learning technique is based on a deep neural network, LassoNet, which is optimised to find the best mapping between the environment, in which the selection happens, and the actual intention of the user. Starting from the assumption that lasso-selection techniques generally express their full potential on 3D point clouds projected on 2D, for example on a screen, Chen et al. [11] considers a list of unordered points, a 2D surface on which they are visualised and a lasso on this surface, as selection environment. In order to achieve that, the LassoNet has been trained with a data set composed of approximately 30 thousands selection entries. A comparative study with two other lasso-selection methods, one of which consisted of a method proposed by Yu et al. [40], showed that the LassoNet-based selection method demonstrates robustness, effectiveness and efficiency in correctly predicting the user's intentions in many scenarios.

At the moment, the *PointCloudAnnotator* does not feature a deep-learning based selection technique that automatically exclude non-relevant points from the user selection. Nevertheless, taking inspiration from one of the efficient methods just explained to implement a smart-selection will represent a valid extension for the spatial selection technique to be explored. The selection mode based on the intersection with a 3D frustum is a very approximate initial step towards this direction. However much interesting work is left for the future.

7.2.3. Selection Using Gesture Recognition

The techniques proposed by Yu et al. [40] are not limited by the type of inputs, which instead of coming from the classical mouse and pen may also consist of gestures. In this regard Burgess et al. [9] developed an interface to automatically define selection volumes for large-scale point clouds from hand gestures. The motion of the hands is tracked using a Leap Motion[1] device to achieve interactions such as in Figure 7.6. By exploiting the tridimensionality of gestures, the researchers aim to bypass the implicit limitations resulting from the dimensional reduction embedded in 2D support. Any interaction with the tool is performed by hand gestures, from the use of menu functionalities to the creation of the actual selection volume, which is built between the palms of the hands taken as reference.

In spite of possible hardware limitations, the nature of a technique of selecting point directly in 3D holds a considerable advantage over the 2D counterpart and is worth being further researched. The recent increase of applications and consecutive progress in the fields of Virtual and Augmented Reality clearly demonstrates the current trend towards such technologies. In this respect it is easy to imagine that maybe not-so-far-future selection techniques for large 3D point clouds will be based on gestures recognition. Therefore, expanding towards this domain would surely be worthwhile for the *PointCloudAnnotator*.

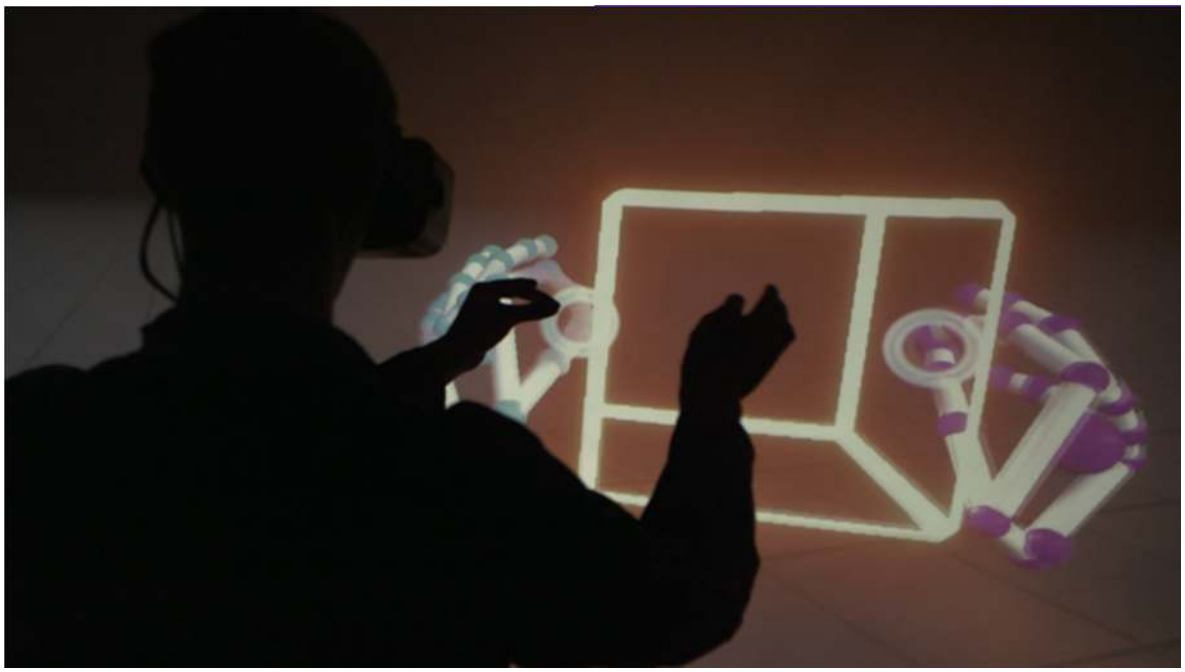


Figure 7.6.: Leap Motion devices enable the user to interact with the virtual environment directly in 3D[2].

[1] The technology used is provided by Leap Motion: `https://www.ultraleap.com`. Accessed December 23rd, 2020.

[2] Ibidem.

8. Conclusions

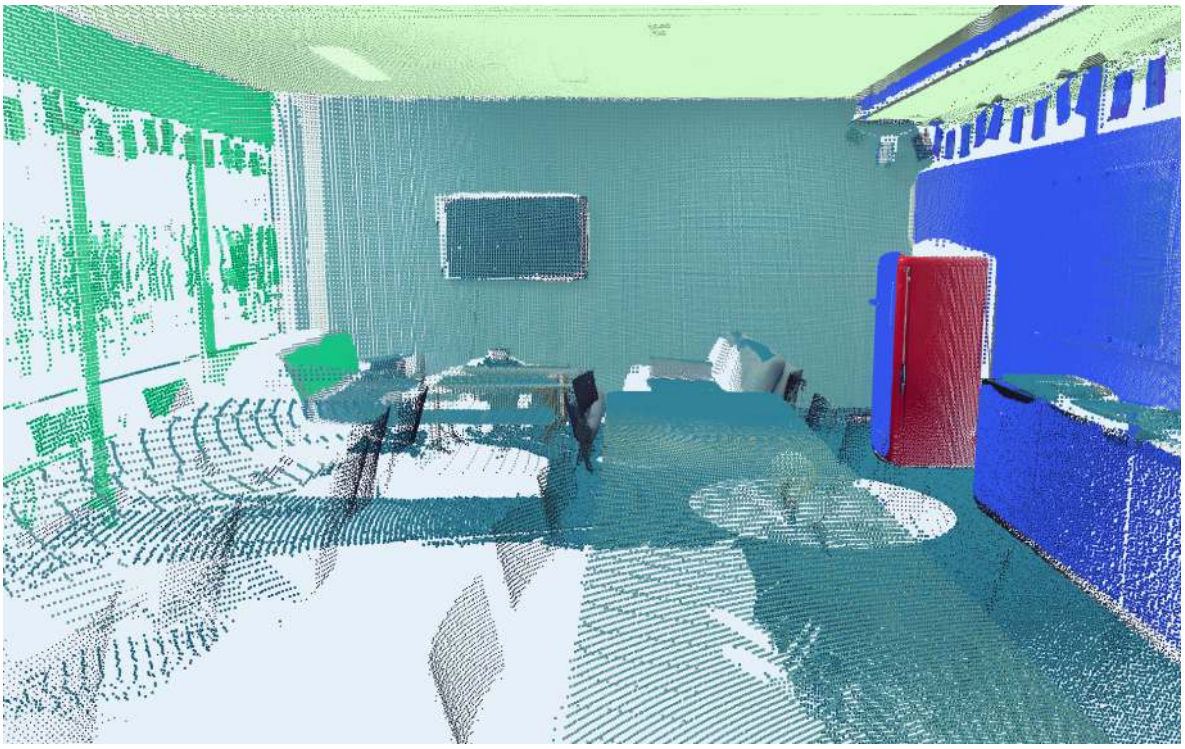


Figure 8.1.: Internal view of a room represented by point cloud data after the annotation by means of planar segmentation in the *PointCloudAnnotator*.

The increasing number of application scenarios exploiting the possibilities provided by large point cloud data is reflected by the development of many tools for the annotation of such data. These tools are usually based on deep-learning techniques, which further require consistent amount of annotated data for training.

In Chapter 2, three categories of annotation tools were presented, namely *(i)* general-purpose tools for data processing, *(ii)* specific-purpose tools developed as part of a research in the field of deep-learning techniques to analyse point cloud data set, and *(iii)* general-purpose tools specifically designed to perform annotations of large point clouds. The result of this thesis is a general-purpose tool to simplify and accelerate such complex and repetitive process, otherwise done manually.

The *PointCloudAnnotator* is a toolkit which allows the user to load a point cloud model from scanned data, either as an unstructured list of 3D points or as a collection of registered panoramic depth maps. In the first case, the application renders a 3D model of the point cloud, while in the second case, an image for each panoramic map is supplemented in the view. Users can interact with both components to create groups of points by individual or multiple selections. The created groups can be semantically labelled and associated with a geometric primitive that best approximates the shape of the group. This association can be automatically achieved via planar segmentation on the entire point cloud. Local fitting of non-planar shapes is another option to geometrically annotate a group.

A significant contribution provided by the *PointCloudAnnotator* consists in the flexibility of the annotation process. Most of the annotation tools, as the ones presented in Chapter 2, allow manipulating point clouds as 3D models, exclusively. In contrast, the *PointCloudAnnotator* combines 3D and 2D visualisations, allowing the user to tailor the selection of specific objects based on the most convenient method, namely from a 3D model or from a 2D image. Such an advantage can motivate the use of the *PointCloudAnnotator*, which is still open to further extensions – for example benefits could result from lasso-based mechanisms for the selection inspired by the works of Chen et al. [11] or Yu et al. [40]. The potential of the *PointCloudAnnotator* can thus be exploited by frequent use and by technical progress in the domain of annotation techniques.

9. Acknowledgement

Professor Renato Pajarola first deserves my gratitude for granting me the opportunity to pursue this work within his research group VMML. His constructive suggestions greatly contributed to improve the final design of the developed toolkit.

This thesis could have not been accomplished without the guidance of my supervisor, Dr. Claudio Mura, who must be thanked for the many valuable discussions and technical insights he provided. I value our collaboration even more, as the working environment was challenged – in terms of working mode and mostly of communication – by the extraordinary conditions of these difficult times.

Finally, a great thank to my family and friends, who were able to support me throughout these months. Despite the forced *distance*, they were ready to stand by me when I needed the most.

Bibliography

[1] *Description of ASCII .ptx format*. https://sites.google.com/site/matterformscanner/learning-references/ptx-format, October 9, 2012. Accessed December 6th, 2020.

[2] *Apple unveils new iPad Pro with breakthrough LiDAR Scanner and brings trackpad support to iPadOS*. Press release. https://www.apple.com/newsroom/2020/03/apple-unveils-new-ipad-pro-with-lidar-scanner-and-trackpad-support-in-ipados/, March 18th, 2020. Accessed December 4th, 2020.

[3] *Estimating Surface Normals in a PointCloud* in Point Cloud Library Documentation and Tutorials. https://pcl.readthedocs.io/en/latest/normal_estimation.html, 2020. Accessed December 2nd, 2020.

[4] Hasan Asy'ari Arief, Mansur Arief, Guilin Zhang, Zuxin Liu, Manoj Bhat, Ulf Geir Indahl, Håvard Tveite, and Ding Zhao. Sane: Smart annotation and evaluation tools for point cloud data. *IEEE Access*, 8:131848–131858, 2020.

[5] Gary Bishop, Greg Welch, et al. An introduction to the kalman filter. *Proc of SIGGRAPH, Course*, 8(27599-23175):41, 2001.

[6] Dorit Borrmann, Jan Elseberg, Kai Lingemann, and Andreas Nüchter. The 3d hough transform for plane detection in point clouds: A review and a new accumulator design. *3D Research*, 2(2):3, 2011.

[7] Aleksey S Boyko. *Efficient interfaces for accurate annotation of 3D point clouds*. PhD thesis, Princeton University, 2015.

[8] Nicolas Brodu and Dimitri Lague. 3d terrestrial lidar data classification of complex natural scenes using a multi-scale dimensionality criterion: Applications in geomorphology. *ISPRS Journal of Photogrammetry and Remote Sensing*, 68:121–134, 2012.

[9] Robin Burgess, António J Falcão, Tiago Fernandes, Rita A Ribeiro, Miguel Gomes, Alberto Krone-Martins, and André Moitinho de Almeida. Selection of large-scale 3d point cloud data using gesture recognition. In *Doctoral Conference on Computing, Electrical and Industrial Systems*, pages 188–195. Springer, 2015.

[10] Angel Chang, Angela Dai, Thomas Funkhouser, Maciej Halber, Matthias Niessner, Manolis Savva, Shuran Song, Andy Zeng, and Yinda Zhang. Matterport3d: Learning from rgb-d data in indoor environments. *arXiv preprint arXiv:1709.06158*, 2017.

[11] Zhutian Chen, Wei Zeng, Zhiguang Yang, Lingyun Yu, Chi-Wing Fu, and Huamin Qu. Lassonet: Deep lasso-selection of 3d point clouds. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):195–204, 2019.

[12] Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. Meshlab: an open-source mesh processing tool. In *Eurographics Italian chapter conference*, volume 2008, pages 129–136. Salerno, 2008.

[13] David Eberly. Dynamic collision detection using oriented bounding boxes, 1999.

[14] Esri. *Point Cloud Segmentation using PointCNN*. `https://developers.arcgis.com/python/guide/point-cloud-segmentation-using-pointcnn/`. Accessed December 4th, 2020.

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[16] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of cgal a computational geometry algorithms library. *Software: Practice and Experience*, 30(11):1167–1202, 2000.

[17] Frank Fiedrich and Sisi Zlatanova. *Emergency Mapping*, pages 272–276. Springer Netherlands, Dordrecht, 2013.

[18] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3354–3361. IEEE, 2012.

[19] Daniel Girardeau-Montaut. Cloudcompare, 2016.

[20] Stefan Gottschalk, Dinesh Manocha, and Ming C Lin. *Collision queries using oriented bounding boxes*. PhD thesis, University of North Carolina at Chapel Hill, 2000.

[21] Magdalena M Grabowska, David J DeGraff, Xiuping Yu, Ren Jie Jin, Zhenbang Chen, Alexander D Borowsky, and Robert J Matusik. Mouse models of prostate cancer: picking the best model for the question. *Cancer and Metastasis Reviews*, 33(2-3):377–397, 2014.

[22] Timo Hackel, Jan D Wegner, and Konrad Schindler. Contour detection in unstructured 3d point clouds. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1610–1618, 2016.

[23] Johnny Huynh. Separating axis theorem for oriented bounding boxes. *URL: jkh. me/files/tutorials/Separating% 20Axis% 20Theorem% 20for% 20Oriented% 20Bounding% 20Boxes. pdf*, 2009.

[24] Claudio Mura. *Room-aware Architectural 3D Modeling of Building Interiors from Point Clouds*. PhD thesis, Institut für Informatik, Universität Zürich, 2017.

[25] Claudio Mura, Gregory Wyss, and Renato Pajarola. Robust normal estimation in unstructured 3d point clouds by selective normal space exploration. *The Visual Computer*, 34(6-8):961–971, 2018.

[26] Ananya Narain. *Apple's LiDAR Scanner a game-changer in scanning technology?* `https://www.geospatialworld.net/blogs/apples-lidar-scanner/`, March 31st, 2020. Accessed December 4th, 2020.

[27] Sven Oesau, Yannick Verdie, Clément Jamin, Pierre Alliez, Florent Lafarge, Simon Giraudot, Thien Hoang, and Dmitry Anisimov. Shape detection. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.1.1 edition, 2020.

[28] Federal Office of Topography swisstopo. *LiDAR data acquisition*. `https://www.swisstopo.admin.ch/en/knowledge-facts/geoinformation/lidar-data.html`. Accessed December 4th, 2020.

[29] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *IEEE transactions on systems, man, and cybernetics*, 9(1):62–66, 1979.

[30] Andreas ten Pas and Robert Platt. Localizing grasp affordances in 3-d points clouds using taubin quadric fitting. *arXiv preprint arXiv:1311.3192*, 2013.

[31] Giovanni Pintore, Claudio Mura, Fabio Ganovelli, Lizeth Fuentes-Perez, Renato Pajarola, and Enrico Gobbetti. State-of-the-art in automatic 3d reconstruction of structured indoor environments. *STAR*, 39(2), 2020.

[32] François Pomerleau, Francis Colas, and Roland Siegwart. A review of point cloud registration algorithms for mobile robotics. 2015.

[33] Radu Bogdan Rusu and Steve Cousins. 3d is here: Point cloud library (pcl). In *2011 IEEE international conference on robotics and automation*, pages 1–4. IEEE, 2011.

[34] Ruwen Schnabel, Roland Wahl, and Reinhard Klein. Efficient ransac for point-cloud shape detection. In *Computer graphics forum*, volume 26, pages 214–226. Wiley Online Library, 2007.

[35] Filippo Stanco, Sebastiano Battiato, and Giovanni Gallo. *Digital imaging for cultural heritage preservation: Analysis, restoration, and reconstruction of ancient artworks*. CRC Press, 2011.

[36] Samuel T Thiele, Lachlan Grose, Anindita Samsu, Steven Micklethwaite, Stefan A Vollgger, and Alexander R Cruden. Rapid, semi-automatic fracture and contact mapping for point clouds, images and geophysical data. *Solid Earth*, 8(6):1241, 2017.

[37] Om Prakash Verma, Madasu Hanmandlu, Seba Susan, Muralidhar Kulkarni, and Puneet Kumar Jain. A simple single seeded region growing algorithm for color image segmentation using adaptive thresholding. In *2011 International Conference on Communication Systems and Network Technologies*, pages 500–503. IEEE, 2011.

[38] Bernie Wang, Virginia Wu, Bichen Wu, and Kurt Keutzer. Latte: accelerating lidar point cloud annotation via sensor fusion, one-click annotation, and tracking. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 265–272. IEEE, 2019.

[39] Lingyun Yu, Konstantinos Efstathiou, Petra Isenberg, and Tobias Isenberg. Efficient structure-aware selection techniques for 3d point cloud visualizations with 2dof input. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2245–2254, 2012.

[40] Lingyun Yu, Konstantinos Efstathiou, Petra Isenberg, and Tobias Isenberg. Cast: Effective and efficient user interaction for context-aware selection in 3d particle clouds. *IEEE transactions on visualization and computer graphics*, 22(1):886–895, 2015.

[41] Wuming Zhang, Jianbo Qi, Peng Wan, Hongtao Wang, Donghui Xie, Xiaoyan Wang, and Guangjian Yan. An easy-to-use airborne lidar data filtering method based on cloth simulation. *Remote Sensing*, 8(6):501, 2016.

[42] Xiao Zhang, Wenda Xu, Chiyu Dong, and John M Dolan. Efficient l-shape fitting for vehicle detection using laser scanners. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 54–59. IEEE, 2017.

Appendices

A. Class Diagram

The class diagram represented in Figure A.1 shows the main relations between the classes of the *PointCloudAnnotator*. To improve clarity the structures of the classes have been simplified to include only the members most relevant for the comprehension of the general relations.

The class having the main role of managing the resources is the `Core`. This class receives inputs from the UI via `PclAnnotator` and reacts accordingly. When necessary updated data is forwarded to the `Scene` instances, which are in charge for the rendering of correct items. `GeometryScene` and `PanoramicScene` derive from `Scene`. The first handles all the 3D objects, such as the main point cloud model or the *subclouds*, while the second manages the quad textured with the panoramic image. Each instance of `Scene` is associated with a `GLWidget`, which inherits from the the Qt class `QOpenGLWidget`. The `GLWidget` is responsible for drawing the associated scene at each frame and for handling mouse and keyboard inputs. If the user moves the camera, these inputs are forwarded to the `InputManager` which controls the `Camera` objects of the `Scene` instances.

The `Core` loads the point cloud data by using the methods of the virtual class `ScanDataManager` and stores it into a `PointCloudData` object. Additionally, data necessary for the rendering of the cloud is used to create the rendering structures of a `PointCloudDrawable` object. The actual rendering structures, such as buffers an material properties, are handled by the base class `Drawable`. Other classes representing 3D objects derive from `Drawable`. These are `Quad` for the panoramic image, and the shapes for the 3D-selection, `Sphere` and `Frustum`. `ShaderProgram` and `Texture` represent, respectively the shader programs and the textures needed for the panoramic images.

In order to properly handle the annotation process of the point cloud, the `Core` is supported by the `GroupsManager` which controls the current state of the *groups* entities.

Some of the *PointCloudAnnotator*'s functionalities require more mathematical operations, which are provided by the virtual class `MathHelper`.

Finally, the headers `Utilities` and `GLIncludes` provide shared constants and functions between multiple classes of the application.

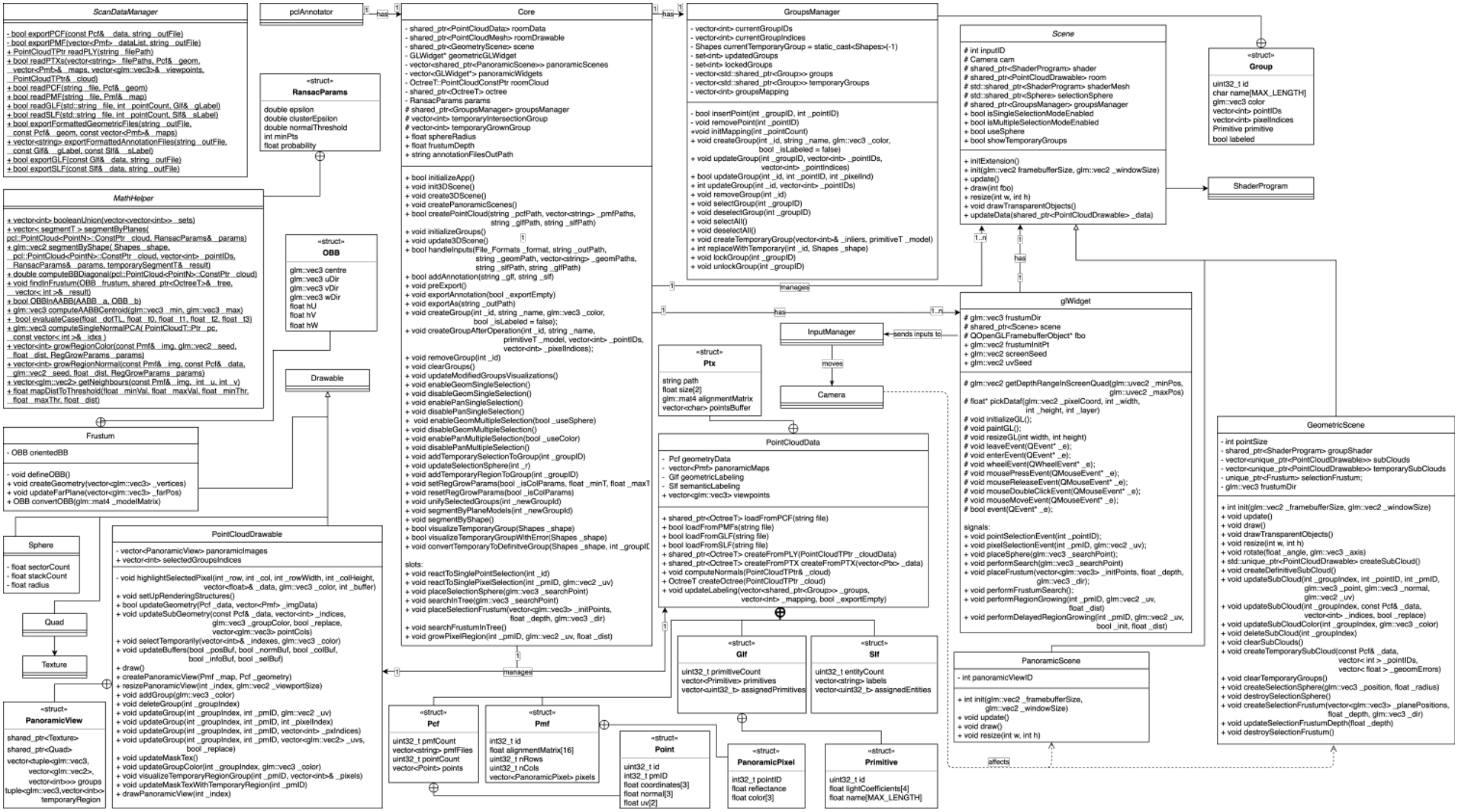


Figure A.1.: Class diagram of the *PointCloudAnnotator* application.

B. Loading of Formatted Point Cloud Files

Point cloud files formatted according to Section 4.3 can be loaded into the *PointCloudAnnotator* by means of the "Formatted" function within "Load Files" menu option in the top bar menu. Various combinations of inputs are possible: only PCF, the complete packet or only the PCF and PMF geometric files. These loading procedures are illustrated in Figure B.1.

The appearing dialog contains four input fields to hold the PCF file, possibly the PMF files and the annotations files, GLF and SLF if already available. The files' paths can either be browsed with the dedicated button or directly typed in the fields. The option a) in Figure B.1 shows that when a point cloud originally derives from a PLY file, the only available data will be the 3D model. In this case no annotation file is provided. Generally, after browsing the PCF file, which is the only file that must be always provided, the fields for the other files are auto-completed according to the input file just selected. The process of inputting the files can thus be reduced in time. Unless other panoramic files need to be added or the name of the annotation files differs from the PCF's, the user can already start the actual loading process.

Path b) in the same figure shows the process of loading a complete packet of formatted files: the user first browses the PCF file, then selects all the panoramic files and if the auto-completed pair of annotation files needs to be changed, the respective "Browse" buttons can be used. Eventually, the 3D point cloud with the associated panoramic maps and annotation is loaded in the toolkit.

The last path, c), illustrates the specular case to option a). Here, the user selects a PCF file and a group of PMF files but no annotation files.

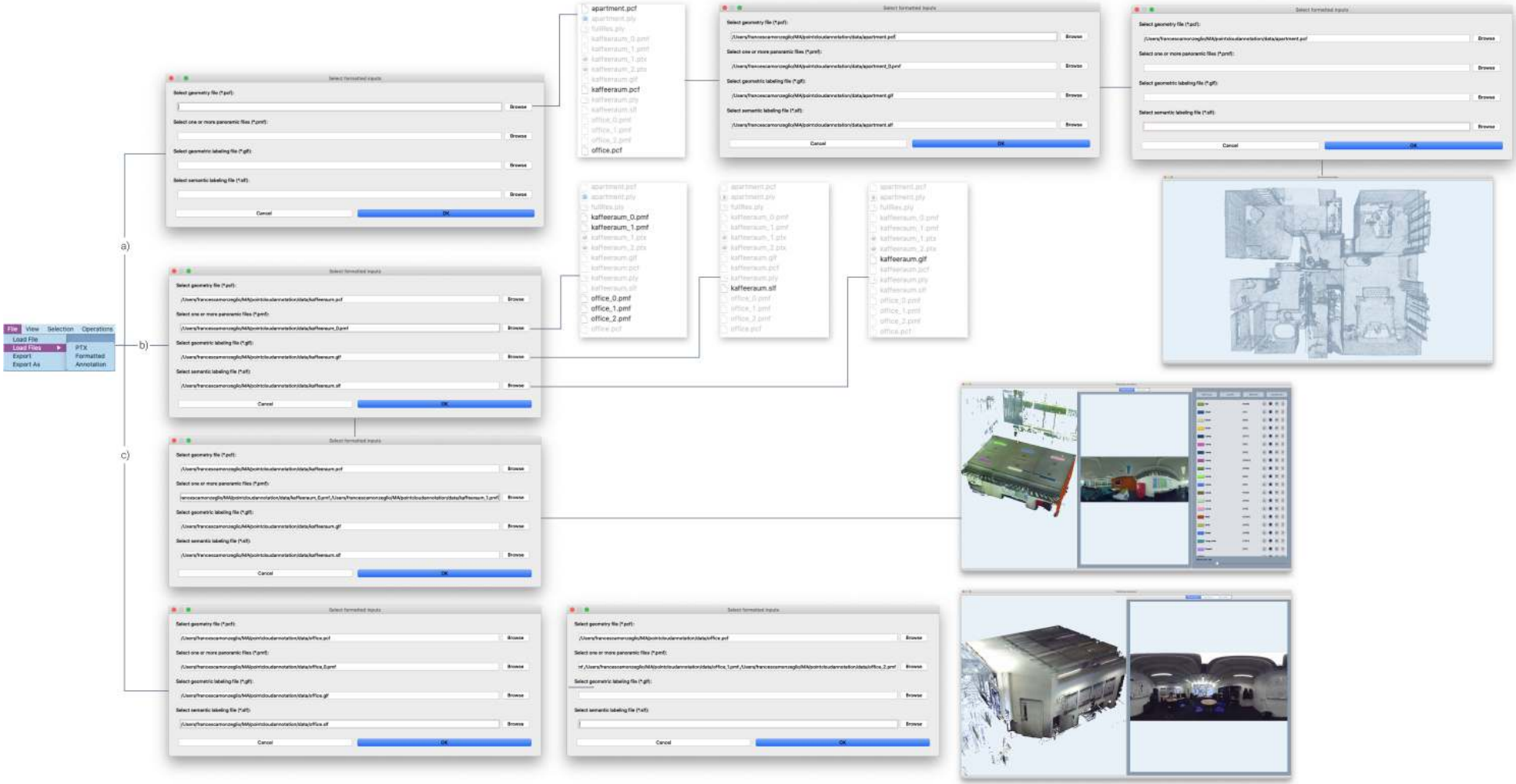


Figure B.1.: Different combinations of formatted files can be loaded in *PointCloudAnnotator*: a) only the 3D model is associated with a PCF file; b) geometric files (PCF and PMF) with annotation files (SLF and GLF) produce the most complete point cloud data; c) the visualisation of 3D model and related panoramic images is given by providing a PCF and a group of PMF files.

C. Configuration

This chapter aims to provide the necessary instruction for a proper configuration of the project. First, it will be explained how to use the preprocessing script to estimate the normals for a collection of PTX files. Afterwards, the instructions to setup the *PointCloudAnnotator* are provided.

C.1. Preprocessing Script for Normal Estimation

The script `createPLYWithNormals` is contained in the main project folder inside the *scripts* folder. It consists of a small CMake project that can be configured directly within the console as follows:

1. Create the folder for the executable with

   ```
   mkdir build-BUILD_TYPE
   ```

   where `BUILD_TYPE` can be release, debug or relwithdebinfo.

2. Enter the folder

   ```
   cd build-BUILD_TYPE
   ```

3. Generate the necessary files

   ```
   cmake ..  -DCMAKE_BUILD_TYPE=BUILD_TYPE -DCMAKE_PREFIX_PATH=QT_PATH
   ```

   where `QT_PATH` indicates the storing location of the Qt library.

4. Compile the project

   ```
   make
   ```

5. Now the script is ready to be used

   ```
   ./createPLYWithNormals outputPath inputFile1 inputFile2
   ```

   where `outputPath` is the name of the output PLY file (without .ply extension). After the output path, all the input PTX files are listed.

C.2. Setup of the *PointCloudAnnotator*

The main project is also based on CMake but it actually requires two steps in order to be correctly configured. First, the CMake project containing the external libraries is created as follows:

1. From the repository folder, enter the folder with the libraries

   ```
   cd external
   ```

2. Create the folder for the executables

   ```
   mkdir build-BUILD_TYPE
   ```

   where `BUILD_TYPE` can be release, debug or relwithdebinfo.

3. Enter the folder

   ```
   cd build-BUILD_TYPE
   ```

4. Generate the necessary files

   ```
   cmake ..  -DCMAKE_BUILD_TYPE=BUILD_TYPE -DCMAKE_PREFIX_PATH=QT_PATH
   ```

   where `QT_PATH` indicates the storing location of the Qt library.

5. Compile the project

   ```
   make
   ```

6. Now the script is ready to be used

   ```
   ./createPLYWithNormals outputPath inputFile1 inputFile2
   ```

   where `outputPath` is the name of the output PLY file (without .ply extension). After the output path, all the input PTX files are listed.

Now that the libraries are ready, the actual project can be generated by prompting the following terminal commands

1. In the main folder of the repository, create the folder for the executables

   ```
   mkdir build-BUILD_TYPE
   ```

   where `BUILD_TYPE` can be release, debug or relwithdebinfo.

2. Enter the folder

   ```
   cd build-BUILD_TYPE
   ```

3. Generate the necessary files

   ```
   cmake ..  -DCMAKE_BUILD_TYPE=BUILD _TYPE
   ```

4. Compile the project

   ```
   make
   ```

5. The program can be launched

   ```
   ./pointcloudannotation
   ```

It is worth stressing, that to avoid any linking problem, the *PointCloudAnnotator* must be launched from the parent folder of the executable. The generating procedure of the main program, additionally configures the project `tests_pointcloudannotation`, which contains the Google Tests used during the development.