3D building planimetry from unaligned point clouds

Master Thesis

Dave Meier

University of Bern

July 2022

Abstract

In the past years, much research has been done in the field of reconstructing indoor scenes. This is not a trivial task because of the clutter and occlusion that may occur due to furniture. In this work, a new algorithm is presented for reconstructing rooms from point clouds. It is able to model diagonal ceilings and multiple ceiling-heights. The algorithm is split into two parts. In the first part, planes are detected using Principal Component Analysis (PCA) and a new clustering-algorithm. In the second part, the algorithm detects corners and edges and links them together. Finally, a closed triangle mesh of the room is returned. We show how our algorithm is able to work on synthetic data and on point clouds that were captured from a laser scanner.

Prof. Dr. David Bommes, Computer Graphics Group, University of Bern, Supervisor
Simone Raimondi, Computer Graphics Group, University of Bern, Assistant

Contents

1 Introduction

This thesis focuses on the reconstruction of non-empty rooms. The algorithm takes a point cloud as input and outputs a closed mesh of the reconstructed room. One key attribute of this thesis is that it is able to handle diagonal ceilings and multiple ceiling heights per room. Most research that has been done in the past years assumes horizontal ceilings and vertical walls. A second key attribute is that the room may contain furniture. Furniture is difficult to handle because it does not only hide parts of the walls and the floor, but it also generates points that are not wanted. We are able to reconstruct the contours of the room despite occlusions and generate a closed mesh as a result.
There exist current algorithms which are able to convert point clouds of rooms into a mesh or other formats. However, many of them require the room to be in a very specific structure or are not able to work with furniture inside.
The contributions of this thesis are:

1. A clustering-algorithm which separates the point cloud into sets of points that lie on the same plane.
2. An algorithm which is able to find edges and corners in the room.
3. An algorithm which is able to link the corners together to a closed room.

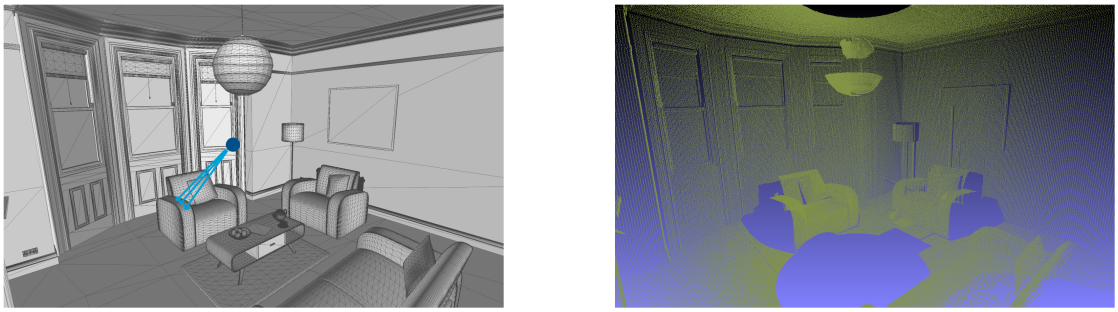
This thesis consists of 6 chapters. In the remaining part of this chapter (chapter 1), background information is given. Chapter 2 discusses related work. Chapters 3 and 4 describe the project itself. Here, the most important parts of the algorithm are explained. Chapter 5 shows how the algorithm performs on different point clouds. Generated results and also the runtime of the algorithm are shown. Chapter 6 contains a conclusion and elaborates on future work.

1.1 Point Cloud

A point cloud is a set of points within a vector space. Each point consists of its location and may also contain other attributes such as a normal or a color.
The point clouds used in this thesis contain only the coordinates of the points. All other information was either not generated or stripped away. This is on one hand very beneficial because it means that the algorithm does not require other information to work correctly. On the other hand, the algorithm could be improved by adding more information such as normals or colors. The chapter ”future work” (Chapter 6) elaborates more on this.
The point clouds were either generated synthetically or taken from real rooms using a laser scanner. The following two sections explain how the point clouds were generated with each method.

1.2 Synthetic Generated Point Clouds

For the synthetic generation of point clouds, a program was written which acts like a virtual laser scanner. It takes a mesh and the position of the laser scanner as input and outputs a point cloud. It works like raytracing, but stops at the first intersection and has a field of view of 360° x 300°. To model the noise of real laser scanners, the resulting intersection points are changed by a small random vector. This vector has a length between 0mm and 2mm.



(a) Input (mesh) (b) Output (point cloud)

Figure 1.1: A model of a room transformed into a point cloud

1.3 Laser Scanner

To acquire point clouds of real rooms, we used a laser scanner called ”Leica Cyclone FIELD 360”. Its accuracy is 1.9mm within 10m, 2.9mm within 20m and 5.3mm within 40m. The field of view is 360° x 300°.

1.4 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a method that computes new axes for a dataset. These axes are called ”principal components”. Principal components are unit vectors that fit the data the best and are ordered by importance. In machine learning, this method can be used to get a better understanding of the data and perform a dimension reduction by removing less important axes.

In this thesis, it is used to check whether a set of points is rather flat or voluminous. This is done by comparing the importances λ_i of the principal components to each other. There are 3 special cases the principal components can take in the 3-dimensional space:

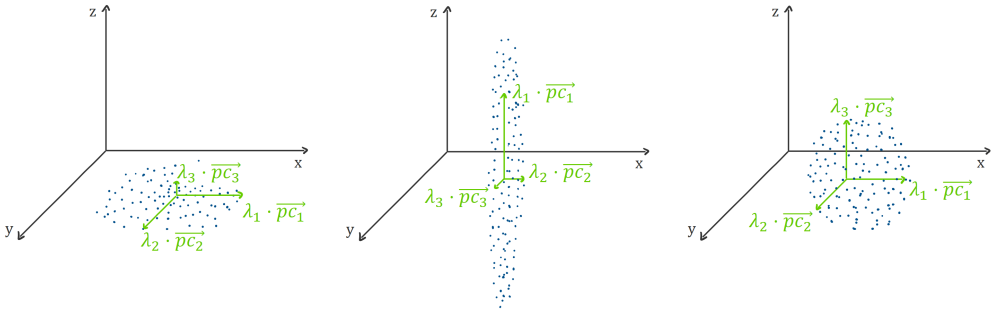


Figure 1.2: 3 cases of PCA

In Figure 1.2, the principal components $\vec{pc}_i$ are multiplied by their respective importance λ_i. In the first example, λ_3 is substantially smaller than the other two. In the second example, λ_2 and λ_3 are substantially smaller than λ_1. In the last example, all three vectors have about the same size. It is not hard to see that we are looking for the first case where λ_3 is substantially smaller than λ_1 and λ_2. In this case, the set of points expands only in two main directions and is rather flat. For a set of points, we compute the following two scores:

$$\text{planarScore} = 1 - \frac{\lambda_3}{\lambda_2}$$

$$\text{stringScore} = 1 - \frac{\lambda_2}{\lambda_1}$$

, for $\lambda_1 > \lambda_2 > \lambda_3$ the importances of the principal components sorted in decending order

PlanarScore is an indicator for the planarity of a set of points. It holds a value between [0,1], while numbers close to 1 indicate that the set of points is planar. It works by comparing the smallest importance λ_3 to the second smallest, λ_2. In case of a planar set of points, λ_3 is substantially smaller than λ_2. The division $\frac{\lambda_3}{\lambda_2}$ is very small and planarScore close to 1. On the other hand, if λ_3 gets close to λ_2, the division $\frac{\lambda_3}{\lambda_2}$ is close to 1 and planarScore close to 0.
StringScore was introduced to make sure that λ_2 and λ_1 have around the same size. This is necessary because we want to avoid cases where the set of points is flat, but also like a string. StringScore works the same way as planarScore, with the only difference that it compares λ_2 to λ_1.

The PCA is used in this thesis for the clustering algorithm (Chapter 3) at two places:

1. Computing the normals for the whole point cloud, and at the same time excluding points from the point cloud where the local neighborhood is not flat enough.

2. Computing the best-fitting plane for clusters.

2 Related Work

2.1 Related Work

In the past years, much work has been done in the field of reconstructing interiors or even complete buildings from point clouds. This section intends to give an overview of the past research and does not claim to be complete. The works differ from one another in many aspects. The most common and important ones are input format, occlusion handling, parts of the building, restrictions for the structure of the building and whether windows and doors are modelled. All works described in this section require a point cloud as input and are able to handle some form of occlusion. Some of the works require additional data such as acquisition path (in case of Mobile Laser Scanners), Scanner positions, number of rooms and other information.

2.1.1 Whole building with multiple storeys

There exist works that are able to process point clouds of whole buildings, for example [5], [3], [10], [13], [19]. All of them mention to use a projection of the points onto the z-axis. Floors and ceilings show up as peaks in this diagram because they are in most cases horizontal and stretch over the whole building. The detected floor- and ceiling-heights are then used to divide the point cloud into separate storeys. After the separation into storeys, they use algorithms for reconstructing single storeys. How they do it and also how other works do it is described in the next section.

Some works that are able to process whole buildings have features that other works only operating on single storeys are not able to provide. [3] is able to reconstruct staircases in a building. They are also able to model empty spaces across multiple storeys. [5] mentions the possibility to use facade information for window detection in future works.

2.1.2 Process single storeys and rooms

Besides the works mentioned in the previous section, there are other works that are able to reconstruct storeys or rooms of a building. Previous work differentiates between plane-based, segmentation-based and volumetric-based reconstruction methods. In plane-based reconstruction methods, surfaces are extracted and used for the reconstruction. Examples for plane-based reconstruction algorithms are [18], [2], [12], [11], [15], [17], [9]. Segmentation-based reconstruction algorithms segment the storey into rooms and are able to infer walls with the segmentation. Some works using this approach are [5], [3], [8], [6], [1], [7], [4]. Volumetric-based approaches divide the space into volumetric shapes. This was done for example in [14].

There are different ways how the works deal with occlusions and clutter generated from furniture. A common way is to assume that furniture doesn't reach the ceiling. Walls are then detected because they are vertical and contain points near the ceiling. This approach is used for example in [5], [1], [3]. Another approach is to add context to detected surfaces. Context could be for example that a surface bounded to the ceiling and the floor is more likely to be a wall. Some works that use context-based clutter recognition are [15], [8].

2.1.3 Detect doors and windows

There exist many works that are able to detect and model doors and windows. This is not trivial because a missing part in the wall may either be due to occlusion, or it may simply be an opening (window or door). In order to reliably differentiate between "empty" and "occluded" parts of the wall, the algorithm must know if there is an object that occludes the part. This can be done by implementing a raytracing algorithm as done in [2], [4], [12], [15], [13], [7], [20].

2.1.4 Special room layouts

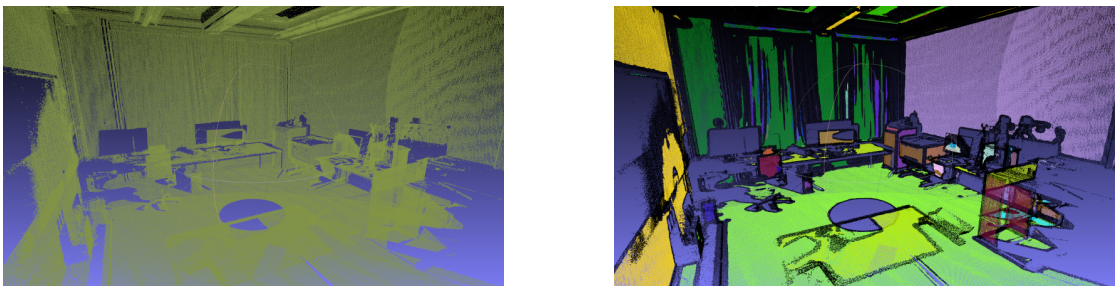
Most works described above work only with rooms that have horizontal ceilings and floors and planar vertical walls. However, there are some exceptions that are also able to work in more complex room layouts. [1] is able to reconstruct rooms with curved walls. Another work, [8], is able to reconstruct rooms with diagonal ceilings.

3 Clustering

The first part of the algorithm is clustering. In this step, points that lie in the same 3-dimensional plane are put together in a single cluster. A cluster is defined by a set of points, a center and a normal. The center and the normal together define the cluster-plane.

3.1 Cluster-Algorithm

The Cluster-Algorithm is an algorithm which takes a point cloud as input and assigns points that lie in the same 3-dimensional plane to the same cluster. Points for which the local neighborhood is not flat are excluded because they might belong to multiple clusters. Figure 3.1 shows a point cloud before and after clustering:



(a) Before Clustering

(b) After Clustering

Figure 3.1: Room 104 before and after clustering

The clustering-algorithm is divided into three parts:

1. The PCA, planarScore and stringScore are computed for each point in the point cloud.
2. An iteration over each point yields the clusters. Points whose planarScore or stringScore do not exceed a certain threshold are excluded.

3. Similar clusters are merged together.

The following subsections describe each step in detail.

3.1.1 PCA computation

The PCA computation is done for each point separately. For this, the K closest points are taken and the PCA is computed as described in Chapter 1. If this subset of points is flat enough (i.e. $planarScore$ exceeds a certain threshold), the smallest principal component $\vec{pc}_3$ can be seen as the normal of the point. The most crucial part is to decide the value of K. If K is set too small on a high density point cloud, wall-irregularities are captured instead of the "real" wall-normal. Consider the following wall:

Figure 3.2: A rough wall. On the left side, K is set too small and because of that, the normal belongs to the wall-irregularity. On the right side, the normal belongs to the actual wall.

If the amount of neighbors, K, is set too small, the radius of the PCA is also too small. We might end up calculating the normal(s) of each bump instead of the more general normal of the wall. As a consequence, the points on the wall get all very different normals and recognizing that these points belong to the same wall is difficult and error-prone. If K is set high enough, the PCA looks at a bigger part of the wall and the resulting normal of the point belongs to the actual wall instead of the bump in its local neighborhood.
The radius of the PCA and the maximum extend of the wall-irregularities have to be set by the user. By default, they are set to 0.1m and 0.005m respectively. With those two values, the algorithm is able to determine the required neighbors (K) and the threshold for the planarScore, $planarScoreTH$.

The number of required neighbors K is computed using the average spacing. First, the average spacing of the points is computed using the two-ring neighborhood (20 neighbors). Second, the required number of neighbors K is computed by dividing the area of the PCA by the average area of a point:

$$K = \frac{\pi \times \text{PCAradius}^2}{\pi \times (\frac{\text{avgspacing}}{2})^2} = \frac{\text{PCAradius}^2}{(\frac{\text{avgspacing}}{2})^2}$$

The point cloud may be less dense at some points and that is why we might end up taking too much neighbors. To avoid this, the PCA-computation also looks at the distance of each neighbor and excludes it if it is beyond the PCA-radius.

The $planarScore$ is described in Chapter 1 in more detail. Basically, it describes the ratio between the smallest and the second smallest principal component. The following image shows a wall from the side:

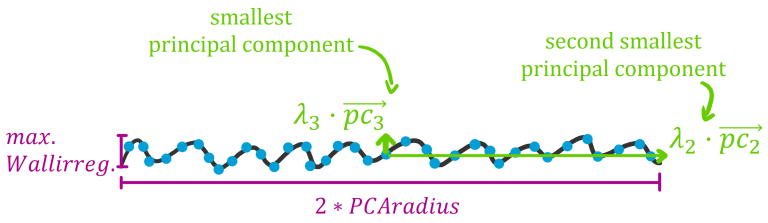


Figure 3.3: A wall from the side in black. Possible captured points in blue. The principal components of the PCA in green. The predefined values $maxWallIrregularities$ and $PCAradius$ in violet.

To classify a set of points as planar, we require the computed $planarScore$ of the PCA to be bigger than a certain threshold value. This threshold value is computed as follows:

$$planarScoreTH = 1 - \frac{\text{maxWallIrregularities}}{2 * \text{PCAradius}}$$

In general, this should return numbers between 0.9 and 1. The meaning of 0.9 is that we require λ_3 to be at least 10 times smaller than λ_2. In this case, the points stretch mainly in only two directions and don't vary much along the third principal component.
If $planarScoreTH$ is under 0.9, the computed normal may deviate too much from the "real" surface-normal. In the extreme case of $2 * PCAradius == maxWallirregularities$, we tell the algorithm that within the PCA-radius, the points may scatter all they want and should still be considered as flat. As a consequence, $planarScoreTH$ will be equal to 0 and we classify all points as flat. We get normals, but they do certainly not represent the actual surface-normals in general.

3.1.2 Cluster generation

After the PCA computation, we use the computed values to generate clusters. The basic idea is that points with the same normal that lie on the same plane are put in the same cluster. In particular, a point has to fulfil three requirements in order to belong to a certain cluster:

1. The point has to be close to the plane defined by the cluster
2. The point-normal has to be similar to the cluster-normal
3. The point has to be close to at least one point that is already in the cluster.

The reason for the first and second requirement is visualized in Figure 3.4. The cluster c_1 should only contain points belonging to the horizontal black wall close to c_1. The points p_1 and p_2 should not be added to it. Requirement 1 ensures that p_1 is not added to the cluster because p_1 is too far away from the cluster-plane. Requirement 2 ensures that p_2 is not

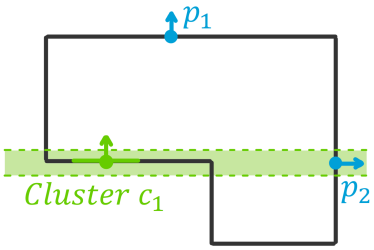


Figure 3.4: Room from above. Points p_1 and p_2 should not be added to cluster c_1

added, because it's normal is not similar to the cluster-normal. The reason for the third requirement is because the computed normals are not exact. A small inaccuracy in the normal makes a big difference for the plane at high distances away from the center.

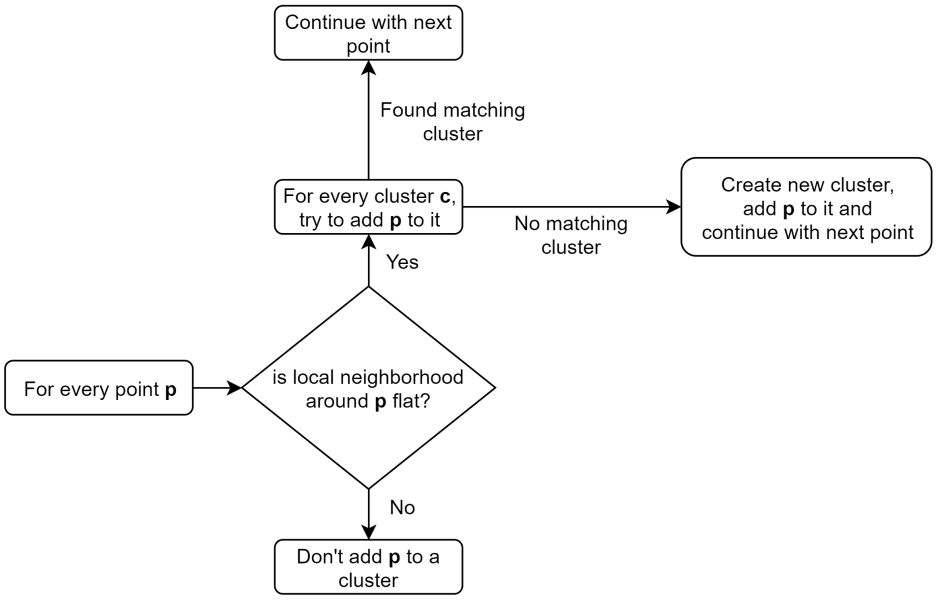


Figure 3.5: An overview of the cluster generation.

Figure 3.5 shows how the clusters are generated. The algorithm starts by iterating over all points in the point cloud. For every point p, it is then checked if it's neighborhood is flat. This is the case when the $planarScore$ of p is greater than $planarScoreTH$ defined in the previous section. The $stringScore$ of p is also used to avoid cases where the local neighborhood is like a string and doesn't span an area. If the local neighborhood is flat, the algorithm tries to add p to an existing cluster. If there is no cluster that fits p, a new one is created.
To increase the speed of the algorithm, the iteration over all points uses a queue within the standard iteration. This queue contains neighbors of already processed points. They are added after processing a point. With this setup, the algorithm becomes a region-growing algorithm. This is necessary because the third requirement requires points to be close to the cluster. If the iteration wasn't region-growing, the algorithm would be very slow because too many clusters would be created.
When trying to add a point to a cluster, we need to check for the three requirements described at the beginning of this section. For requirement 1, the required distance to the plane is set to 0.05m. It can be set by the user. For requirement 2, the maximum allowed angle-difference $reqAngle$ between the normal of the point and the normal of the cluster is calculated from the user-defined variables $PCAradius$ and $maxWallIrregularities$. For this, we compute the maximum possible angle between two normals belonging to the same surface. Consider the following image:

Figure 3.6: The same wall captured at two different points

Figure 3.6 shows a wall in black and two captured points with normals on the wall. The angle between those two normals is the maximum possible angle-difference between two points on a wall. This is because the wall is as irregular as possible at both points.
We can use this angle as the maximum allowed deviation from the cluster-normal when trying to add a point to the cluster. It is calculated as follows:

$$reqAngle = 2 * acos(\frac{\vec{vp} \cdot \vec{vi}}{\|\vec{vp}\| * \|\vec{vi}\|}), \text{ for } \vec{vp} = \begin{pmatrix} 2 * PCAradius \\ 0 \end{pmatrix} \quad \vec{vi} = \begin{pmatrix} 2 * PCAradius \\ maxWallIrregularities \end{pmatrix} \tag{3.1}$$

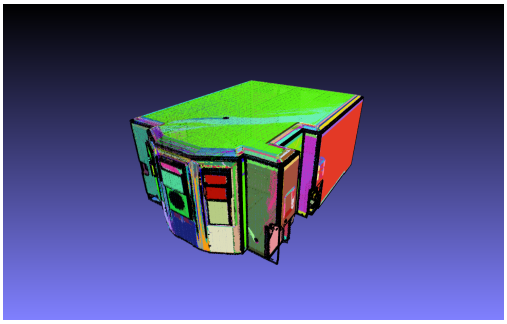
The more points we add to a cluster, the more precise becomes its normal. At the beginning however, it is not very precise and we may put points that do not belong to the same wall in the same cluster, especially if they are far away from each other. To overcome this problem, we introduced requirement 3 which requires points to be close to the cluster when adding them. For this, a method is needed to compute the distance from a point to the cluster. A very simple approach would be to compute the distance to every point in the cluster and take the smallest one. However, this is not very efficient and the distance is more precise than needed. The algorithm uses a list called ”markerPoints”. Every time a point is added to the cluster and it is more than 0.1m away from all marker-points, it is added to the marker-point list. The distance from a point to the cluster is then computed by computing the distance to all marker-points and take the smallest one.

3.1.3 Cluster merging

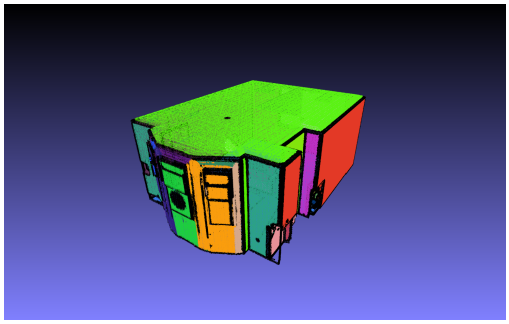
After the clusters are generated, they need to be merged because of three reasons:

1. Points on the same surface may have created multiple clusters.
2. A surface may have bigger irregularities like a curtain.
3. Clusters that are separated by space are merged into one cluster.

The merging-algorithm merges the clusters in the same order as mentioned above. This is important because merging two clusters results in a bigger cluster with a possibly changed normal. The resulting cluster may then not be able anymore to merge with more important/similar clusters. For example, close similar clusters should be merged first before merging spatially separated clusters.
The following figures show a room before and after merging:



(a) Before merging



(b) After merging

There are two different ways two clusters can get merged. In the first way, two clusters are merged if the center of the smaller cluster fits into the bigger cluster. In the second way, two clusters are merged if a certain percentage of the points of the smaller cluster fit into the bigger cluster. The algorithm uses merging-tickets to specify which of the two methods is used.

A merging-ticket contains the following information:

- The required angle for a point to belong to a certain cluster
- The required distance for a point to belong to a certain cluster
- The required percentage of points of the smaller cluster that should also fit into the bigger cluster. If set to 0, only the center of the smaller cluster needs to fit into the bigger cluster.
- The maximum allowed distance between the two clusters.

For each of the mentioned merging reasons in the beginning of this section, a merging ticket is created. They are displayed in the table below:

	reqAngle	reqDist	reqPercentage	reqCloseness
Merge close similar clusters	$\pi/32$	0.04	0	0.5m
Merge close irregularities (e.g. curtains)	$\pi/4$	0.05	0.8	0.5m
Merge distant similar clusters	$\pi/32$	0.1	0	-1 (infinity)

Most of the values in the table above are heuristic values and may be changed by the user. They were only tested empirically and changing them might improve the clustering result.
For merging reason 1 and 3, the algorithm uses the faster merging-approach where only the cluster-center of the smaller cluster is tried to fit into the bigger cluster. In the table, this is depicted in the ”reqPercentage” column. For merging reason 2, this is not possible because the normals of the two clusters might be very different. This is the reason why we use the approach with the required percentage. Because of this, the normals of the two clusters may be very different. The algorithm accepts clusters that have different normals up to 45°.

4
Clustering to Mesh

After the clustering, it is necessary to combine the clusters to generate a closed mesh. This is not trivial because of three reasons. First, the clusters are still just a set of points with an associated plane. We need to find out which clusters need to be linked to which other clusters. Second, the furniture inside the room was not removed and does also contain clusters which should not be part of the resulting mesh. Third, the furniture also hides parts of the required clusters. The algorithm should reconstruct the missing parts to output a closed mesh.
Two algorithms were developed to address this challenge. The first one is much simpler and works only on a small set of rooms. It assumes that there is just one flat floor and one flat ceiling. Furthermore, it only works with vertical walls. The second algorithm is able to reconstruct rooms with diagonal walls and ceilings. It also works with multiple ceiling-heights. It is based on triangles, which represent corners between 3 clusters. Both algorithms are not able to detect doors and windows. This component was not implemented since it is already well studied. Works on this subject are for example [2], [4], [12], [15], [13], [7], [20].

4.1 Algorithm 1 - Projecting walls to 2 dimensions

This section describes a simple algorithm for transforming the clusters into a closed mesh. The final algorithm uses another approach described in the next section. Nevertheless, this simple algorithm is also described to show the capabilities and limitations of requiring walls to be vertical.
This algorithm works by finding the walls, ceiling and floor once the clustering is done. For this, it assumes that the point cloud is aligned such that the floor and ceiling clusters have vertical normals. The floor is then the cluster whose center has the smallest z-value, and the ceiling is the cluster whose center has the highest z-value. The walls need to have a horizontal normal and must reach from the ceiling to the floor.

4.1.1 Motivation for Algorithm 1

The following scenario shows the inspiration for the idea that walls must reach from the bottom to the top. Consider two different rooms, one with a wardrobe and one with a chimney covered by a wall:

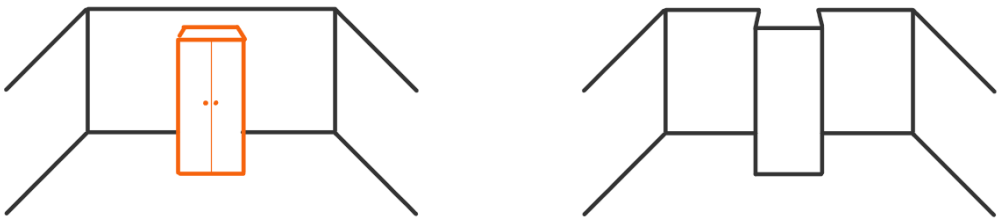


Figure 4.1: Two rooms, one with a wardrobe and one with a chimney

Once the point cloud is taken from those two rooms, the only part where the clusters differentiate from each other is that the clusters on the side of the wardrobe don't reach the ceiling.

4.1.2 First part of Algorithm 1, Wallfinding

The wallfinding-algorithm finds wall-clusters and projects them onto the xy-plane. A cluster is considered to be a wall-cluster if it's vertical and if it reaches from the floor to the ceiling at some locations. There may be cases where furniture hides the whole lower part of a wall. To address such cases, the user is able to set two variables which define how close a wall must be to the floor and ceiling.

4.1.3 Second part of Algorithm 1, Separation of walls

Since the clustering puts all points that lie on the same plane with the same normal in one cluster, there may be cases where one cluster contains multiple walls. Figure 4.2 shows an example of such a case:

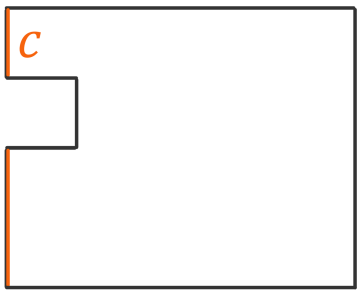


Figure 4.2: A cluster that contains 2 walls

In order to separate the walls, the algorithm walks through the projected wall from one end to the other. If there is a gap of 1cm where no points are present in the whole vertical axis, the walls are separated. The algorithm then continues with the remaining part.
This step is necessary for the last part of the algorithm, where walls are combined in order to get a closed shape. It is also necessary because it may even be the case that a wall is in the same cluster as furniture.

4.1.4 Third part of Algorithm 1, Reconstruction

The reconstruction of the room takes 2-dimensional walls as input and combines them to a closed polygon. In the end, the floor and ceiling levels are used to create a 3-dimensional room.
The algorithm first intersects each wall with all other walls and stores the result in $wallIntersections$. After that, it combines the walls by starting at the biggest wall and then iterating over $wallIntersections$. The wall-intersection for which the intersection point is closest to both walls is chosen and the two walls are combined. The algorithm continues at the new chosen wall and looks for an intersection at the other end of the wall. This continues until we reach the starting wall.

4.2 Algorithm 2 - Triangle finding and linking

The idea of this algorithm is to find every possible corner in the room and then linking them together to a closed room. For this, it intersects every combination of 3 clusters that are not parallel to each other. For each intersection point, we also get 3 intersection lines. If the intersection point is a corner of the room, the intersection lines have only close points on one side. The other side of the line behind the intersection point should be empty.

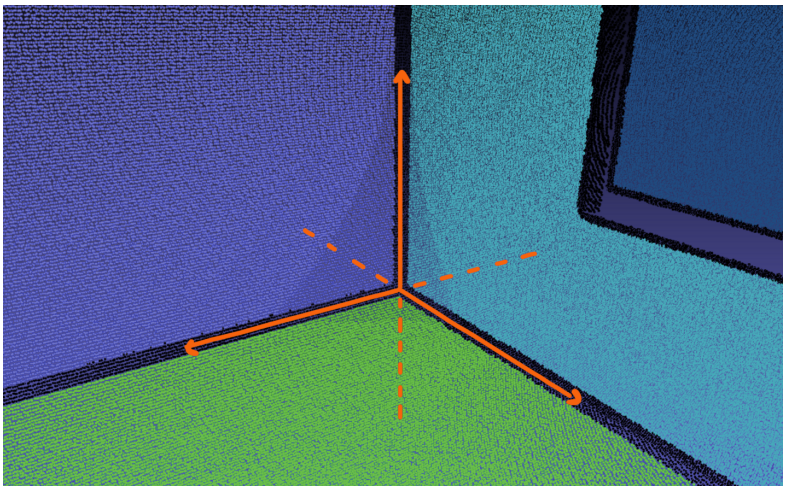


Figure 4.3: An example of a corner and the 3 intersection lines

In the algorithm, an intersection line restricted to only one side of the intersection point is called an arrow. It can be seen as a vector describing the extension of the wall. A corner with 3 arrows is called a triangle.

4.2.1 Motivation for Algorithm 2

The motivation for this algorithm were the limitations of algorithm 1. Because algorithm 1 works in the 2-dimensional space after finding the walls, it is not possible to reconstruct diagonal walls and ceilings. This required a new approach which works in 3 dimensions. By allowing diagonal walls and ceilings, it gets much harder to check whether something is a wall or furniture. It is not possible to check whether a cluster reaches from the floor to the ceiling because there may be different ceiling heights at different positions in the room.

4.2.2 First part of Algorithm 2, triangle finding

The triangle finding algorithm gets 3 clusters as input and has to determine the 3 vectors describing the extension of the wall (arrows). Because the vectors of the arrows are the same as the ones from the intersection lines, the algorithm just needs to determine the sign of each intersection vector.
If one arrow can't be determined, the intersection is not a corner, marked as invalid and not processed further. The algorithm also computes a score between [-1, 1] of how likely it is a corner and suggests lengths for the arrows.

In the beginning, the cluster-points of all 3 clusters are projected to a 2-dimensional plane such that the x- and y-axis of the 2-dimensional plane correspond to the intersection lines in 3 dimension.

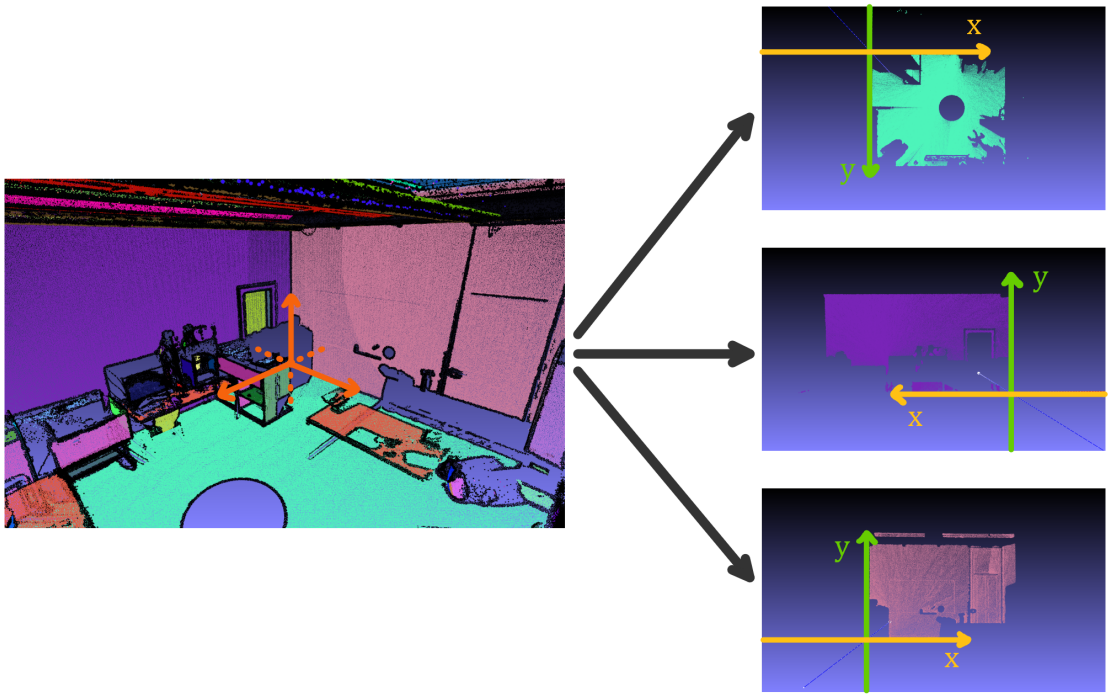


Figure 4.4: The 3 clusters of the intersection showed on the left side are projected to 2 dimensions.

For a corner which has not 90 degrees between two clusters, it is necessary to shear map the projection to align the axes of the 2-dimensional space to the intersection lines.

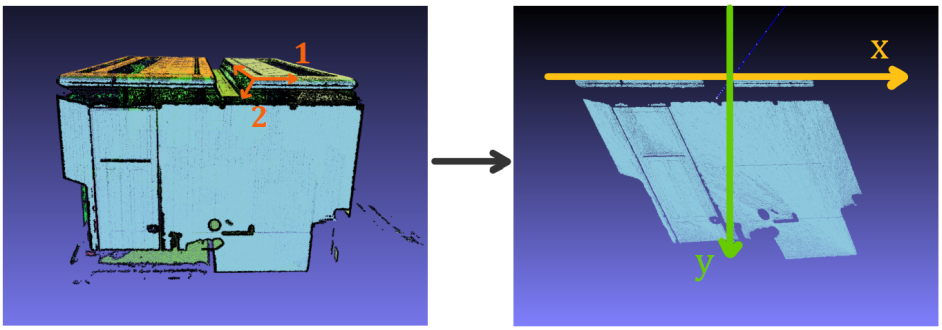


Figure 4.5: The blue cluster is shear mapped during the transformation to 2 dimensions to align vectors 1 and 2 with the x- and y-axis of the projection.

The projection to 2 dimensions enables us to check whether only one quadrant contains points in a local environment around the center. In the case of a corner, all 3 projections should locally only contain points in one quadrant. Note that every resulting arrow in the 3 dimensional space is determined by 2 projections. The algorithm uses the following convention for the x- and y-axis of each projection.

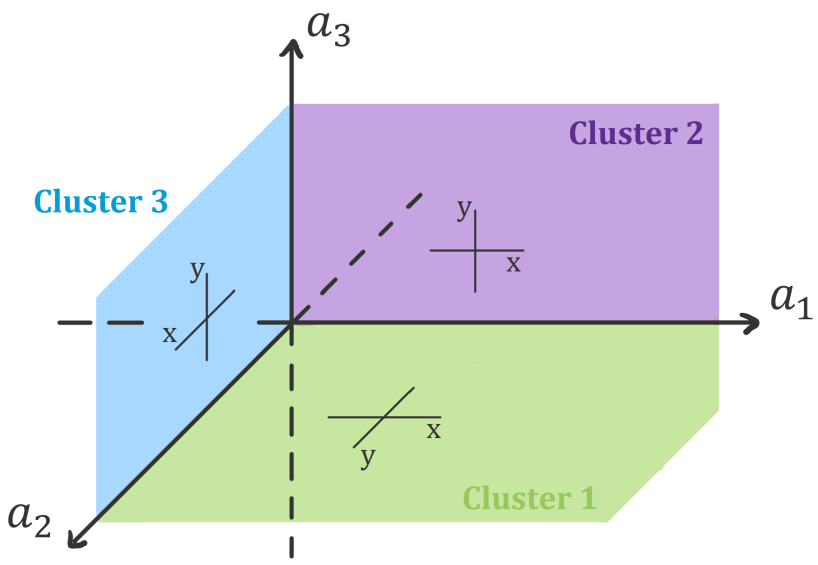


Figure 4.6: The convention the algorithm uses for the arrows and their respective axis in the projections. Please note that the center of each projected coordinate system is in reality at the intersection point. They were moved away for better visibility

Multiple approaches were tested to get the 3 arrows of an intersection. The approach that is used now yields the best results, but is the most computing intensive. The algorithm consists of the following steps:

Algorithm 1: Triangle finding

```
1. The 3 Clusters are projected and possibly shear mapped to 2 dimensions.
2. The 2-dimensional space of all 3 projections is divided
   into squares of 0.1m².
3. The points are used to mark the squares either as 'occupied' or 'empty',
   depending on whether they contain points or not.
4. Every possible combination of 3 arrows is generated.
   For each combination:
     4.1 For all 3 projections, the occupied space is
         calculated. It needs to be greater than 30%.
     4.2 The occupied space in all other quadrants 0.2m away
         from the axis is calculated. It needs to be empty,
         because we should have no points behind a wall.
     4.3 If both requirements are met, the intersection is
         likely to be a corner. Its score is computed using
         the lowest occupation we get from the 3 projections.
```

The occupation requirement of 30% can be set as a parameter. It is the amount in percent a wall needs to be visible. If set too high, not every corner is detected because furniture might hide most parts of the wall. If set too low, much more invalid triangles are generated because of furniture.

One might think that it would be enough to just increase the length of the arrows in a spherical way instead of trying all possible combinations. This is not possible because some walls are much higher than wide. The occupation may be 100% if the arrows are perfectly set, but under 30% if the arrows all have the same length.

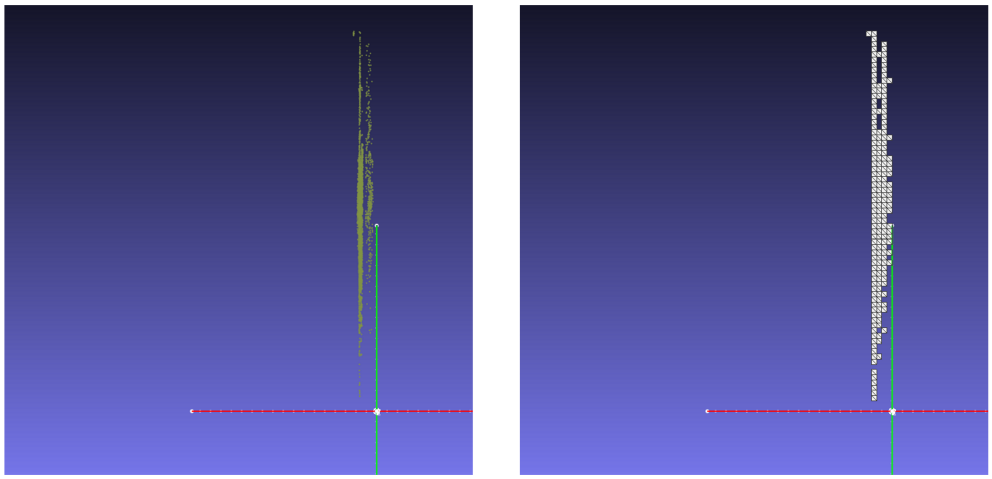


Figure 4.7: An example where the occupation is much higher than wide. On the left side are the projected points, on the right side the occupied squares

The image above shows such a case. If we just iterate from the center in a square or in a circular way, the occupation is too small at every iteration to reach 30%.

It may be the case that we get multiple possible arrow-combinations for one intersection. Consider the following wall:

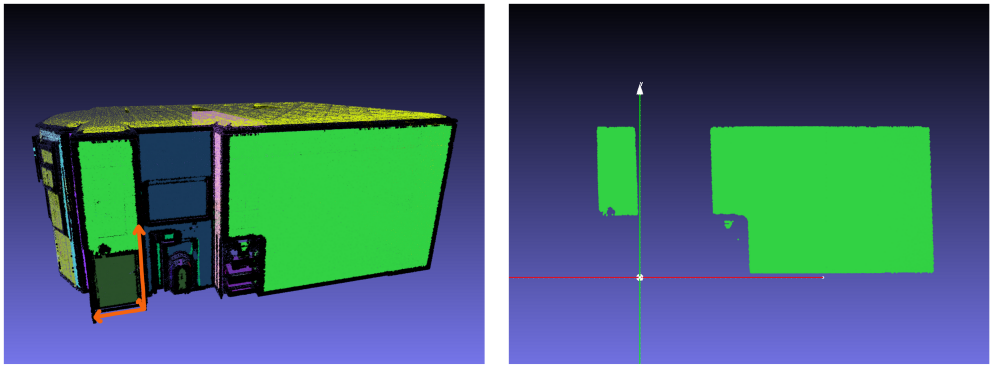


Figure 4.8: An example where two triangles are possible for one intersection

Here, the top-left quadrant should be selected for the given intersection. However, the quadrant on the top right has a better score if we just compute the occupations. To overcome this issue, we must select the first quadrant that matches the requirements. For this, the arrow-combinations are generated in a way that they

grow from the center and as soon as a combination fulfils the requirements, the algorithm quits.

Until now, we assumed that all corners are 'inwards', which means that the space spanned by the three arrows is air. This would restrict the possible rooms to convex rooms. In order to support more types of rooms, the algorithm is extended to detect also outward-triangles. The following image shows the difference between outward- and inward-triangles:

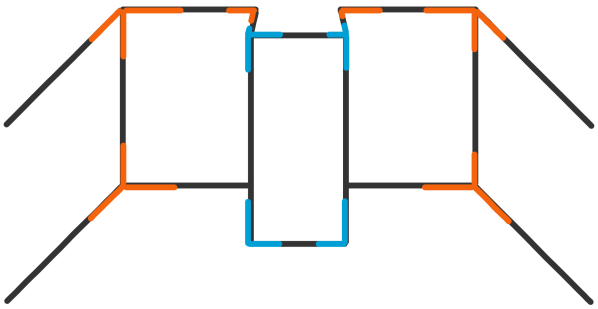


Figure 4.9: A room with inward-triangles in orange and outward-triangles in blue.

Outward-triangles have the unique property that one cluster is inverted. Instead of having points between two arrows, it has points on the opposing side. In the example above, the inverted cluster is the floor-cluster for the two outward-triangles in the middle.

The algorithm is very slow when executed as described above. It has to check every possible arrow-combination for a high amount of possible cluster-combinations. The following tweaks were made to increase the speed:

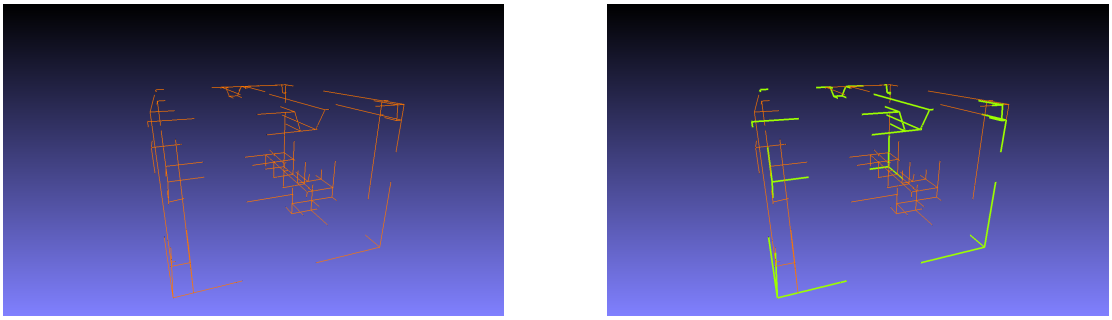
1. The three clusters that are tested for a triangle need to be close to each other. For this, each cluster has to be close to at least one of the other two. To achieve this, the marker-points are used to compute the distance from one cluster to another. The minimum required distance is set to 0.5m and may be increased up to infinity.

2. Large clusters result in substantially more arrow-combinations. To overcome this, the algorithm uses a limit for the maximum arrow-length. This also allows us to skip points during the projection to 2 dimensions that are beyond the limit. The maximum possible arrow-length is set to 5m and may also be increased up to infinity.

3. During the combination of arrows, the arrows are not increased linearly. The further away from the center, the less precision is needed for the arrows.

4.2.3 Second part of Algorithm 2, triangle linking

After the triangles are generated, they need to be linked to create a closed room. The following two images show the generated triangles in orange and the linked triangles in green.



(a) All generated triangles

(b) Found set of linked triangles

The main challenge is to find a set of triangles such that they all fit together. For this, the initial idea was to start at the floor cluster and find a closed circle of triangles. After a circle is found, every two adjacent triangles are part of a wall cluster. Those wall-clusters are then processed the same way as the floor cluster. The problem with this approach is that there may be multiple possible circles in the floor cluster. We may start with one circle, process some walls until we later find out that the floor circle was wrong. We then need to undo every calculations we made so far and start with the new floor-circle. The following image shows a possible wrong circle in orange and the correct one in green:

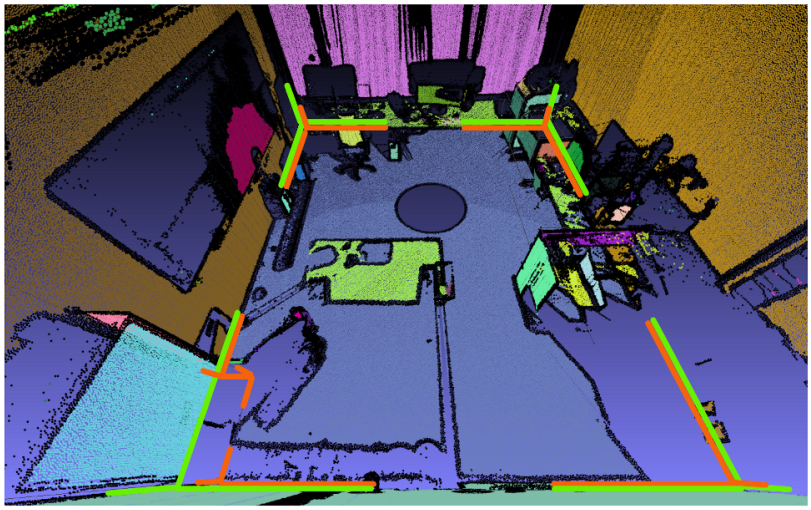


Figure 4.11: Two closed circles of triangles. The green one is preferred.

By just looking at the floor cluster, it is impossible to see which closed circle is the right one. The wardrobe could also be a wall going to the ceiling. In this case, the red circle would be correct. Of course, also in other clusters we might take a wrong circle and have to undo all calculations we made after choosing the wrong circle.

The solution to this problem is a new approach which operates on the triangles instead of the clusters. It uses the fact that every triangle in the result has 3 arrows that need to be linked. It works as follows:

Algorithm 2: Triangle linking

```
1. For every triangle, possible 'followers' are determined for each arrow.
   These are triangles that have 2 clusters in common and the two arrows
   resulting from the intersection line must be opposing.
2. Triangles that have no followers for one or more arrows are
   excluded from further processing.
2. Every triangle has 3 variables 'chosen' which define the
   chosen follower for each arrow. These are set to -1 (undefined)
   in the beginning.
3. The algorithm starts with the first triangle in the list and
   starts the recursion described in step 4.
4. The current triangle tries possible followers for each arrow.
   It starts with the first arrow and sets the first possible follower.
   It also sets the corresponding arrow of the follower to itself.
   It then starts the recursion (4.) with the follower-triangle.
   The recursion ends with 'false' when:
    - The follower triangle had already set another triangle
      to the corresponding arrow.
    - The follower triangle tried all possibilities and
      nothing worked.

   The recursion ends with 'true' when:
    - Every arrow has a follower.

   If the recursion ends with false, all arrows that were set during the process
   are reset to -1.
```

The sorting of the possible followers is very crucial for the success of this algorithm. It may be the case that multiple results are possible, but some are more accurate than others. The following image shows a simple case where two different outputs are possible:

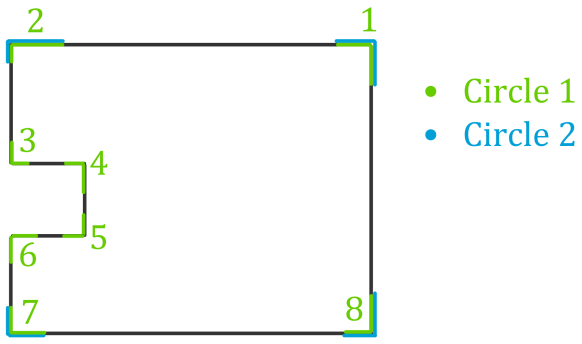


Figure 4.12: A Room from above, two possible closed circles

Triangle 2 may be linked to triangle 3 or triangle 7. Depending on the order of the followers ($\{3, 7\}$) of the downward arrow of triangle 2, the linking algorithm will output either circle 1 or circle 2. Triangle 3 should have a higher priority because it is closer to triangle 2. This applies in most cases and that is why

the algorithm uses mainly the distance to the followers to sort them. However, there are cases where this does not work. One example is showed in the image below.

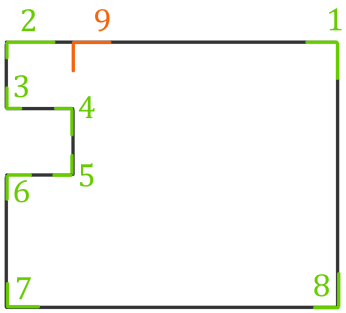


Figure 4.13: Room from above with an additional triangle

It is the same room, but we moved the chimney a little to the top. Because of that, an additional triangle is found by the triangle finding algorithm. It exists because the wall between triangle 4 and 5 might be longer and the missing part between triangle 4 and 9 is just occluded. When starting at triangle 1 and searching for a follower in the left direction, the triangle 9 is closer to triangle 1. This gives triangle 9 a higher priority and thus it is chosen instead of triangle 2. To overcome this issue, the algorithm uses two tweaks:

1. Followers close to each other are sorted according to their score instead of their distance. This ensures that bad triangles generated from walls further away are not selected first.

2. The algorithm uses a mechanism called 'arrow limitation'. The idea behind it is that triangles must not look for followers further away than the limitation. This limitation is set when another triangle is in the way and has the same arrow-direction (instead of being opposing, so basically an 'anti-follower'). In the example above, triangle 9 would be limited by triangle 4 and thus not link to triangle 5.

   But the limitation is not always correct because it could be from furniture or invalid walls. In the example above, the right direction of triangle 2 is incorrectly limited by triangle 9. This means the algorithm should still be able to go beyond the limitation if no other options worked. This is implemented in the sorting of followers. Followers that have for at least one arrow only followers behind the limitation get the lowest possible priority. In the example above, triangle 9 has for the downward arrow only followers past the limitation (triangle 5 is past the limiting triangle 4). Because of this, it gets the lowest possible priority for triangles that want to link to it (e.g. triangle 1).

The arrow-linking algorithm excludes most furniture by default. As soon as the furniture does not fully link to the contours of the room, it is automatically excluded. There are cases however where furniture may be included into the result. The following image shows such a case, it was already shown in Section 4.1 at the beginning of this chapter.

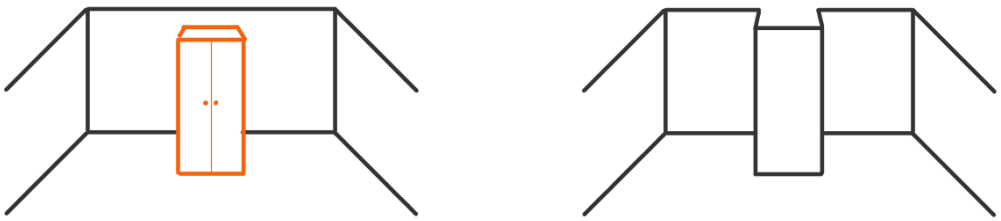


Figure 4.14

Both cases are very similar, except for the fact that the wardrobe doesn't go up to the ceiling. With the triangle-method, the algorithm is able to model the wardrobe and will include it into the result. To avoid this, we must study the difference between both cases and adapt the algorithm accordingly. Two observations and solutions are given below:

1. One observation is that the space above the wardrobe is empty. It would be possible to voxelize the space, recognize the air above the wardrobe and exclude the wardrobe-clusters.
2. Another observation is that the wardrobe links an outward-triangle at the bottom to an inward-triangle at the top, while the chimney links two outward-triangles. We can take advantage of this by limiting the followers of the arrow perpendicular to the inverted cluster of outward-triangles to only outward-triangles.

The algorithm uses the second option which only works on the triangles and doesn't require additional data structure. Basically, this approach excludes the remaining furniture by making sure that every part that consists of an outward-triangle goes straight through the whole room.

4.3 Ideas for Algorithms that did not work

A straightforward idea one might think of when it comes to determine whether a given point belongs to furniture or to a contour is to check whether the point is rather on the inside of the whole point cloud or on the edge. This approach did not work out because the furniture occludes parts of the walls. In the end, in a point cloud which is taken from a laser scanner, every point lies on the edge of the point cloud. This is because every point is at a location where the laser had to travel furthest w.r.t. a specific direction. So, the point cloud is like a highly deformed sphere.
Another idea would be to send virtual rays from the scanner location through each cluster. If the ray hits another cluster behind it, the current processed cluster would likely belong to furniture. If the ray does not hit another cluster within a reasonable distance, it would suggest that the current processed cluster is a wall, ceiling or floor. This works only in convex rooms and would thus highly limit the possible rooms. As soon as we have a concave room, we would potentially classify a wall as furniture.

5 Results

This chapter shows how the final algorithm performs on different point clouds. In chapter 4, two different algorithms were provided for the transformation of the clusters to a mesh. The final algorithm which is used for this results is the second one that works with triangles. The complete algorithm is shown below in a flowchart:

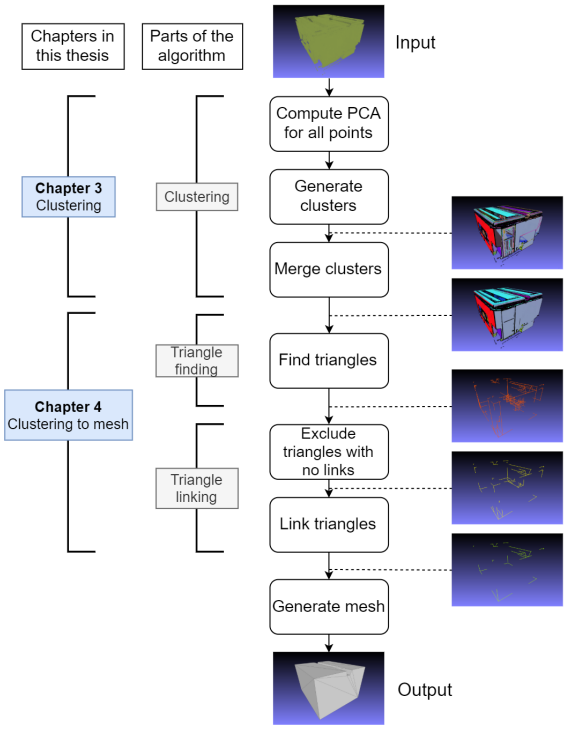


Figure 5.1: A flowchart of the complete process.

The quality of the output and the runtime are very dependent on the parameters that can be set by the user. The most important ones are:

1. PCA radius
2. Maximum wall-irregularities
3. Merging parameters
4. Minimum required covered area for clusters in triangle finding

The first three parameters are responsible for the clustering. A bad clustering may lead to missing triangles which leads to a bad output, or too many triangles which slows down the speed and may also lead to a bad output. The last parameter is responsible for the triangle finding. By default, it is set to 30% which means that a wall must be visible for at least 30% in order to get reconstructed. If set too high, some important triangles might not get detected. On the other hand, if set too low, too many triangles corresponding to furniture are generated which could lead to a bad result.

For the following experiments, the same parameters were used for all point clouds for better comparison. However, it would be possible to slightly increase the quality of the output and the speed by adapting the parameters.

The rest of this chapter contains two sections, one which shows the quality of the results and one which shows the runtime of the algorithm.

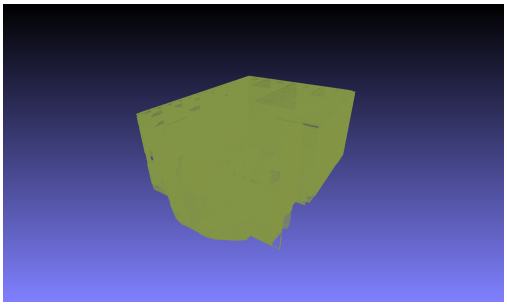
5.1 Quality of the results

5.1.1 Synthetic point cloud 1 - Living room

This room was downloaded from ”https://casual-effects.com/data/”. It is available under the ”CC BY 3.0” License and was created by Jay.

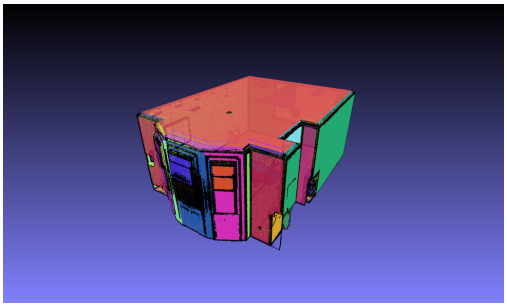
Figure 5.2: Living Room

After downloading, the room was converted into a .ply-file which contains only the mesh of the room. This mesh was then transformed into a point cloud using the algorithm mentioned in Chapter 1. The room needed to be captured from two scanner locations because there exists no scanner location such that all walls are present with only one scan. The reconstructing algorithm is not able to infer walls that are completely missing. The point cloud contains 2M points. The following images show the reconstruction results:

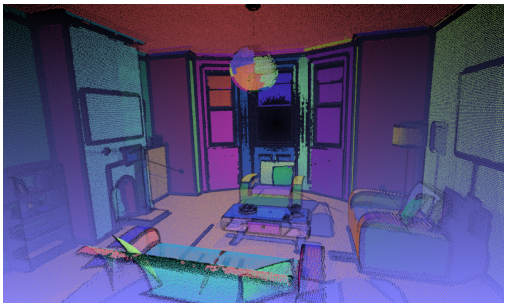


(a) Input point cloud

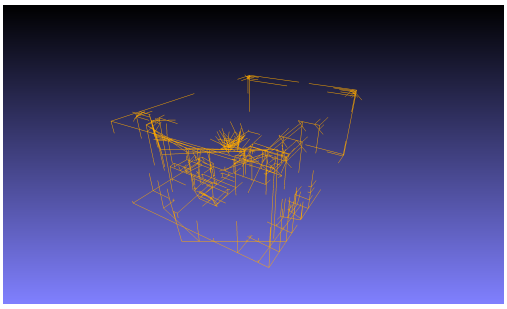
(b) Input point cloud (indoor)



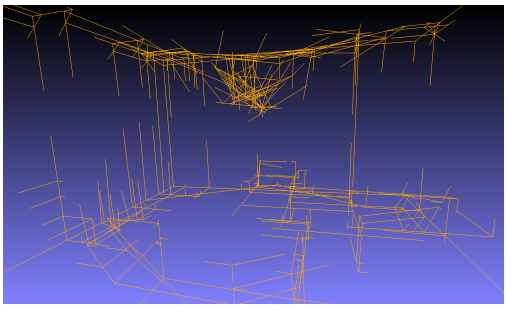
(c) Clustering



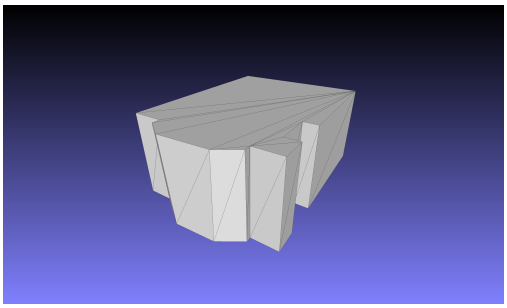
(d) Clustering (indoor)



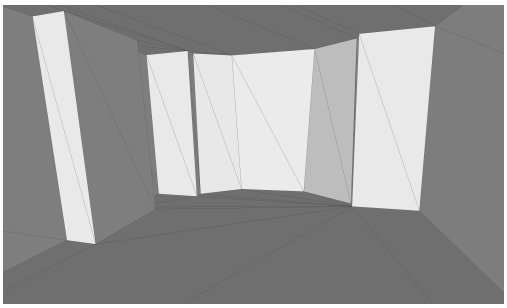
(e) Found triangles, before exclusion



(f) Found triangles (indoor)



(g) Reconstructed room



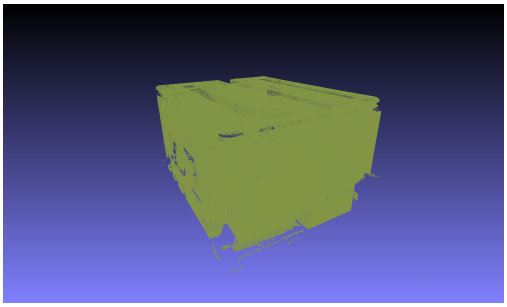
(h) Reconstructed room (indoor)

5.1.2 Real world point cloud 2 - CGG Room 104

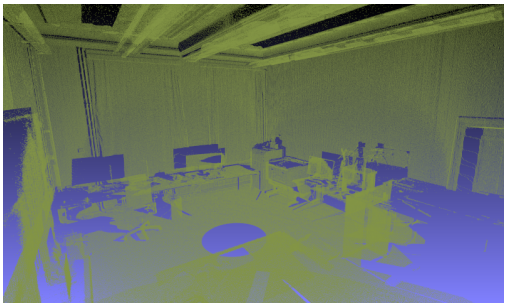
This room was captured using a real laser scanner called ”Leica Cyclone FIELD 360”. It is an office room at the university of Bern.

The original point cloud contains 11M points, but the computer that was used for the experiments was not able to handle the point cloud. This is why the point cloud was simplified to 8M points for the experiments.

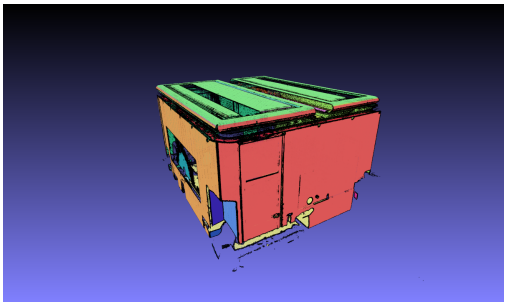
The following images show the reconstruction results of this room:



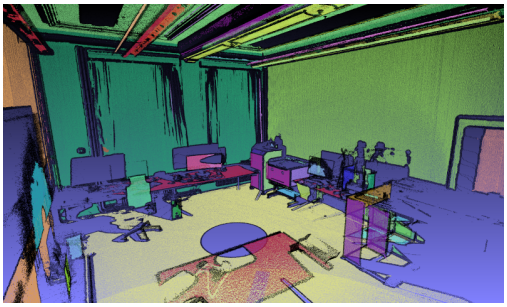
(a) Input point cloud



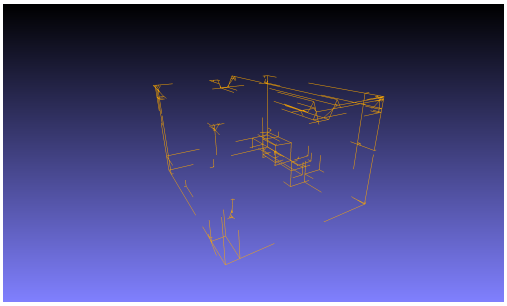
(b) Input point cloud (indoor)



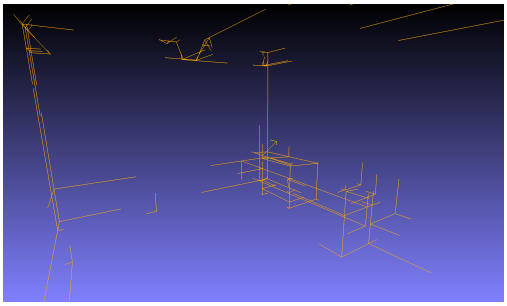
(c) Clustering



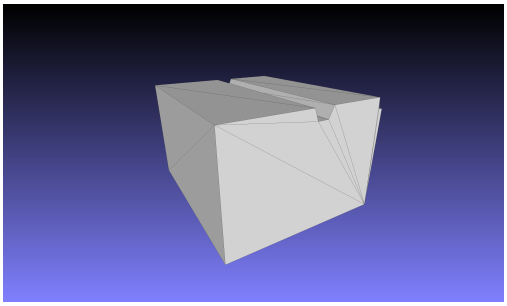
(d) Clustering (indoor)



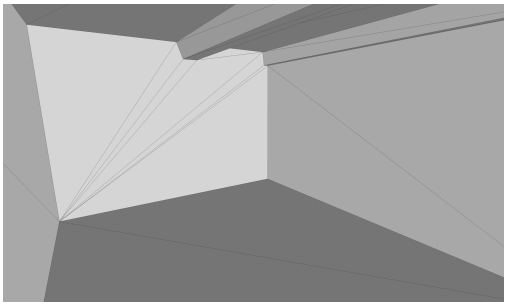
(e) Found triangles, before exclusion



(f) Found triangles (indoor)



(g) Reconstructed room



(h) Reconstructed room (indoor)

5.2 Runtime of the algorithm

This section shows the runtime of the algorithm. The experiments were done on a Windows 10 computer. Its main components are an ”Intel Core i7-6700K” processor running at 4GHz and 16GB Ram.
The runtime is shown for each time-consuming step of the algorithm in the chart below:

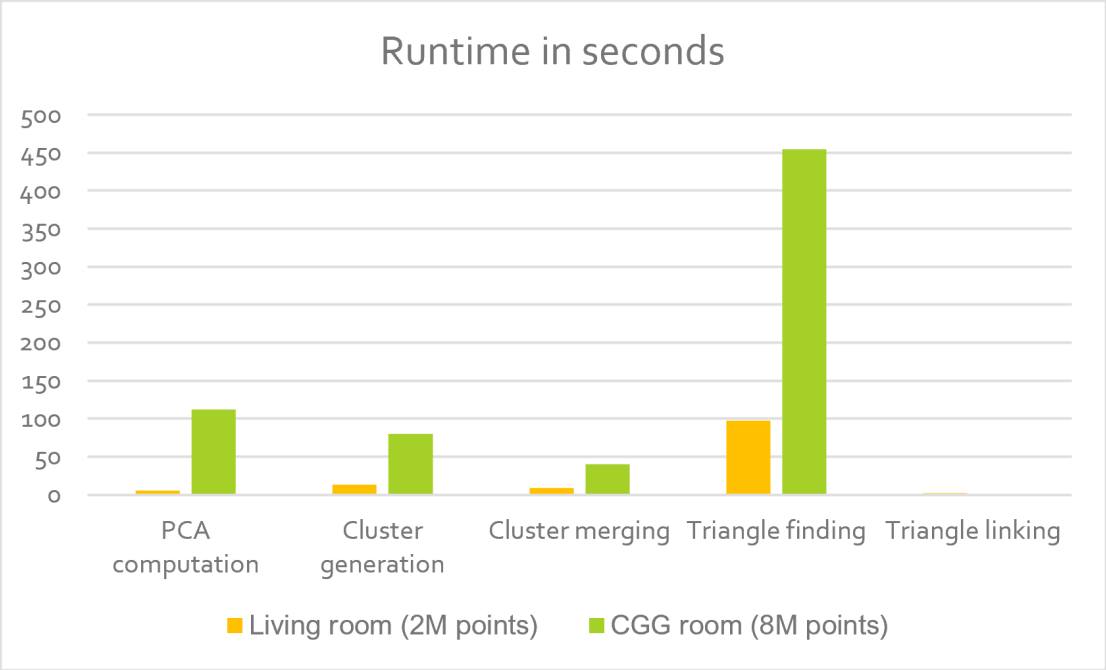


Figure 5.11: The runtime in seconds for each point cloud

The runtime of the algorithm depends mainly on the following two factors:

- The size of the point cloud
- The number of planar surfaces

The size of the point cloud has a huge impact on the runtime of the clustering and the runtime of the triangle finding. As for the clustering, the algorithm has to compute more PCA-values, has to consider more neighbors for the PCA and has to iterate over more points during the generation of clusters. For the triangle finding, it has to project more points to 2 dimensions. The number of planar surfaces in the room has an impact for the triangle finding algorithm. More planar surfaces will generate more clusters and because of this, the triangle finding algorithm has to try out substantially more cluster-combinations.
In the chart above, the difference in the runtime between the two rooms comes mostly from the difference in the size of the point clouds.

6 Conclusion and Future Work

6.1 Conclusion

In this thesis, we explored the problem of reconstructing a (non-empty) room from a point cloud. In the related works, we saw that this subject is already well explored and that there exist multiple different approaches to this challenge. Although they vary in many aspects, they are all able to reconstruct interiors in some way. This work uses a new approach that operates on the corners of rooms. The corners are found by intersecting planes that are found using a new clustering algorithm. The clustering algorithm is very customizable and may be adapted to almost all possible rooms. However, this also makes it more prone to errors, because wrong parameters may lead to a bad clustering, which inevitably leads to a bad output.

6.2 Future Work

The current algorithm is limited to corners that can be represented by triangles. While this is the case for most corners in indoor environments, there still exist rooms that contain corners with more than 3 outgoing edges. One example would be a roof that is formed like a pyramid. There, the highest point is an intersection of 4 planes and has 4 outgoing edges. In the future, the algorithm could be extended to work with corners that have more than 3 outgoing edges. This task is not trivial because of possibly existing furniture. A corner with 4 edges may also be created by 3 wall-planes and one furniture plane.

Another extension for the algorithm would be a detection for windows and doors. This could be done by implementing a raytracing algorithm as done in [2], [4], [12], [15], [13], [7], [20]. Other works that do not use some kind of raytracing are either only able to detect doors and/or are less reliable. Note that a raytracing-algorithm requires knowledge of the scanner position(s).

The algorithm is only able to process a single room. The user has to manually capture and put multiple rooms together. The alignment of multiple point clouds from different scanner locations is already a well-studied topic and called ”registration”. In the future, the algorithm could be improved to accept a registered point cloud of a whole building and reconstruct it. However, this is not a trivial task in combination

with diagonal ceilings and multiple ceiling heights per room. The works described in Chapter 2 that are able to process whole buildings all assume horizontal floors and ceilings.

The algorithm reconstructs rooms from point clouds with no additional information. It was already mentioned in Chapter 1 that this is on one hand beneficial because the algorithm doesn't require more information to work properly. On the other hand, more information could be used to improve the output. Possible additional inputs could be:

- Point colors
- Point normals
- Scanner position(s)
- Point cloud of adjacent rooms

The point colors could be used to improve the clustering by assuming that points with the same colors are more likely to belong to the same cluster. The point normals could also be used to improve the clustering. The scanner position(s) could be used to reliably detect windows and doors. The point clouds of adjacent rooms could be used to detect walls that are more occluded than 70%. In the adjacent room, the wall may be less occluded and this information could be used.
For the point colors, point normals and point clouds of adjacent rooms, the algorithm could be build such that it accepts those inputs to improve the output, but doesn't require them.

Bibliography

[1] Fan Yang, Gang Zhou, Fei Su, Xinkai Zuo, Lei Tang, Yifan Liang, Haihong Zhu, and Lin Li. Automatic indoor reconstruction from point clouds in multi-room environments with curved walls. *Sensors (Basel, Switzerland)*, 19(17):3798, 09 2019.

[2] Mattia Previtali, Lucia Díaz Vilariño, and Marco Scaioni. Indoor building reconstruction from occluded point clouds using graph-cut and ray-tracing. *Applied Sciences*, 8(9):1529, 09 2018.

[3] Lin Li, Fei Su, Fan Yang, Haihong Zhu, Dalin Li, Zuo Xinkai, Feng Li, Yu Liu, and Shen Ying. Reconstruction of three-dimensional (3d) indoor interiors with multiple stories via comprehensive segmentation. *Remote Sensing*, 10(8), 08 2018.

[4] Wenzhong Shi, Wael Ahmed, Na Li, Wenzheng Fan, Haodong Xiang, and Muyang Wang. Semantic geometric modelling of unstructured indoor point cloud. *ISPRS International Journal of Geo-Information*, 8:9, 12 2018.

[5] Hélène Macher, Tania Landes, and Pierre Grussenmeyer. From point clouds to building information models: 3d semi-automatic reconstruction of indoors of existing buildings. *Applied Sciences*, 7:1030, 10 2017.

[6] Rareş Ambruş, Sebastian Claici, and Axel Wendt. Automatic room segmentation from unstructured 3-d data of indoor environments. *IEEE Robotics and Automation Letters*, 2(2):749–756, 04 2017.

[7] Sebastian Ochmann, Richard Vock, Raoul Wessel, and Reinhard Klein. Automatic reconstruction of parametric building models from indoor point clouds. *Computers & Graphics*, 54:94–103, 2016. Special Issue on CAD/Graphics 2015.

[8] C. Mura, O. Mattausch, and R. Pajarola. Piecewise-planar reconstruction of multi-room interiors with arbitrary wall arrangements. *Computer Graphics Forum*, 35(7):179–188, 2016.

[9] Charles Thomson and Jan Boehm. Automatic geometry generation from point clouds for bim. *Remote Sensing*, 7:11753–11775, 09 2015.

[10] Sven Oesau, Florent Lafarge, and Pierre Alliez. Indoor scene reconstruction using feature sensitive primitive extraction and graph-cut. *ISPRS Journal of Photogrammetry and Remote Sensing*, 90:68–82, 04 2014.

[11] Claudio Mura, Oliver Mattausch, Alberto Jaspe, Enrico Gobbetti, and Renato Pajarola. Automatic room detection and reconstruction in cluttered indoor environments with complex room layouts. *Computers & Graphics*, 11 2014.

[12] M. Previtali, Luigi Barazzetti, R. Brumana, and Marco Scaioni. Towards automatic indoor reconstruction of cluttered building rooms from point clouds. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, II-5, 05 2014.

[13] Adam Stambler and Daniel Huber. Building modeling through enclosure reasoning. In *2014 2nd International Conference on 3D Vision*, volume 2, pages 118–125, 12 2014.

[14] Kourosh Khoshelham and Lucia Díaz Vilariño. 3d modelling of interior spaces: Learning the language of indoor architecture. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XL-5:321–326, 06 2014.

[15] Xuehan Xiong, Antonio Adan, Burcu Akinci, and Daniel Huber. Automatic creation of semantically rich 3d building models from laser scanner data. *Automation in Construction*, 31:325–337, 2013.

[16] Victor Sanchez and Avideh Zakhor. Planar 3d modeling of building interiors from point cloud data. In *2012 19th IEEE International Conference on Image Processing*, pages 1777–1780, 09 2012.

[17] Enrique Valero, Antonio Adán, and Carlos Cerrada. Automatic method for building indoor boundary models from dense point clouds collected by laser scanners. *Sensors*, 12(12):16099–16115, 2012.

[18] Angela Budroni and Jan Boehm. Automated 3d reconstruction of interiors from point clouds. *International Journal of Architectural Computing*, 8:55–73, 01 2010.

[19] Eric Turner, Peter Cheng, and Avideh Zakhor. Fast, automated, scalable generation of textured 3d models of indoor environments. *IEEE Journal of Selected Topics in Signal Processing*, 9(3):409–421, 04 2015.

[20] Shayan Nikoohemat, Michael Peter, Sander Oude Elberink, and George Vosselman. Exploiting indoor mobile laser scanner trajectories for semantic interpretation of point clouds. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, pages 355–362, 2017.

[21] Ruisheng Wang, Lei Xie, and Chen Dong. Modeling indoor spaces using decomposition and reconstruction of structural elements. *Photogrammetric Engineering and Remote Sensing*, 83:827–841, 12 2017.

[22] Jaehoon Jung, Cyrill Stachniss, and Changjae Kim. Automatic room segmentation of 3d laser data using morphological processing. *ISPRS International Journal of Geo-Information*, 6:206, 07 2017.

[23] Daniel Huber, Burcu Akinci, Antonio Adan, Brian Okorn, and Xuehan Xiong. Methods for automatically modeling and representing as-built building information models. 01 2011.

[24] Ioannis Anagnostopoulos, Michael Belsky, and Ioannis Brilakis. Object boundaries and room detection in as-is bim models from point cloud data. 07 2016.

[25] Philipp Jenke and Benjamin Huhle. Statistical reconstruction of indoor scenes. 01 2009.

[26] Angela Budroni and Jan Boehm. Toward automatic reconstruction of interiors from laser data. 2009.

[27] Brian Okorn, Xuehan Xiong, and Burcu Akinci. Toward automated modeling of floor plans. 2010.

[28] Satoshi Ikehata, Hang Yang, and Yasutaka Furukawa. Structured indoor modeling. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1323–1331, 12 2015.

[29] Dorit Borrmann, Jan Elseberg, Kai Lingemann, and Andreas Nuchter. The 3d hough transform for plane detection in point clouds: A review and a new accumulator design. *3D Research*, 0202, 06 2011.

[30] Lucia Díaz Vilariño, Edward Verbree, Sisi Zlatanova, and Abdoulaye Diakité. Indoor modelling from slam-based laser scanner: Door detection to envelope reconstruction. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLII-2/W7:345–352, 09 2017.

[31] Angela Budroni and Jan Boehm. Automatic 3d modelling of indoor manhattan-world scenes from laser data. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 38:115–120, 01 2010.

[32] Ioannis Anagnostopoulos, Viorica Pătrăucean, Ioannis Brilakis, and Patricio Vela. Detection of walls, floors, and ceilings in point cloud data. pages 2302–2311, 05 2016.

[33] Alexandre Boulch, Martin de la gorce, and Renaud Marlet. Piecewise-planar 3d reconstruction with edge and corner regularization. *Computer Graphics Forum*, 33, 08 2014.

[34] Lucia Díaz Vilariño, Joaquin Martínez-Sánchez, Susana Lagüela, Julia Armesto, and Kourosh Khoshelham. Door recognition in cluttered building interiors using imagery and lidar data. volume XL-5, 06 2014.

[35] B. Quintana, S. A. Prieto, A. Adán, and F. Bosché. Door detection in 3d colored laser scans for autonomous indoor navigation. In *2016 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, pages 1–8, 10 2016.

[36] Fabio Dell'Acqua and Robert Fisher. Reconstruction of planar surfaces behind occlusions in range images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24:569–575, 05 2002.