Improving virtual representations of environments for usage in augmented reality

Adrian Hirt

Supervised by

Prof. Dr. Marc Pollefeys

Dr. Iro Armeni

Master thesis

Departement of Computer Science

ETH Zürich

Switzerland

August 2023

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computer Vision
and Geometry Lab

Abstract

In this thesis, I present a method to merge data captured from the real world in the form of a point cloud with a low-detail virtual model of the same environment. The method first extracts the relevant structure from the captured point cloud and then uses this structure to generate a low-detail version of the point cloud. This low-detail version of the point cloud will then be fitted into the virtual model, using a combination of a custom global registration and ICP. The fitted point clouds will then be merged into a single point cloud, resulting in a version of the virtual model with added data from the real world.

1. Introduction

1.1. Overview

In this thesis, I explore the possibility to augment a 3D representation of a building containing interior structure with data captured from the real world. The representation might have some inaccuracies in regard to scale, and lack details such as color, furniture and other more detailed structure, which could be added with the captured data.
I start with a simple, relatively rough model of the building, which is generated from the floor plan of the building, and first explore various methods of capturing data from the real world. I then explore some different approaches to process the captured data, such that it can be added to the 3D model. This includes a process to fit the captured data into the 3D model, needing little to no human interaction during the process.
I then show in more detail the final method I developed, and explain the various intermediate steps required for the whole pipeline. I also present some other approaches which I evaluated, and elaborate why I decided against using these approaches.
I will compare my method to other approaches, for example directly running the ICP algorithm on the data. I will also outline the shortcomings of my method and possible future work to improve it.

1.2. Motivation

In my semester project ”AR indoor navigation” [1], I developed a proof-of-concept for a system which navigates users through buildings using a head-mounted mixed reality device such as the Hololens 2.

This method uses a 3D model of the environment for the computation of the navigation, as well as for the visualization of the navigation (e.g. to correctly occlude the rendered navigation visualization when it disappears behind obstacles).

[1] https://github.com/Adrian-Hirt/SemesterProject

The 3D model of the building was generated from the floor plans of the structure, which simplifies setting up the model, as no extensive scanning of the real world (e.g. by creating a point cloud using LIDAR or structure from motion) is needed. However, the 3D model lacks details, and might contain some dimensional inaccuracies. This can be due to inaccuracies in the floor plan itself (e.g. wrong scale of the sides of a room), in the reconstruction process (e.g. by making a wall too thin) or using a wrong height of the floor (as this measurement is often absent in floor plans).

A floor plan also does not include many details, for example the height of a door, which therefore results in the doors being ”holes” in the wall, stretching from the floor all the way to the ceiling. This results in several issues with the navigation setup.

For example, if the virtual world (where the navigation is computed & rendered in) differs too much from the real world, the visualization of the navigation will be inaccurate or wrong, e.g. by having the visualization lead through a wall, the height of stairs being incorrect etc.

Using the 3D model to locate the device from one (or possibly several) 2D query images is also rather difficult, as many important features either are missing or don't match their real world counterpart.

In the application I developed for the project, the user has to select their start-point from a list of given start locations, which translates the virtual model to match the position of the user. This is extremely cumbersome and limits the usability of the application. Also, as the scale of the model is not exactly the same as the real world, the navigation often would clip through walls.

My main motivation for this thesis is now to develop an approach where I can still start with the 3D model generated from the floor plan, and then use the application a few times to capture real-world data, which then corrects the scale of the model and adds important details, such that localization is possible and the visualized navigation path matches the real world.

1.3. Applications in practice

As previously mentioned in the section about the motivation 1.2, this thesis is based on some issues I came across during development of my application for the *AR indoor navigation* semester project. One application in practice could definitely be to augment the virtual models in that project in such a way that the localization & tracking work better, while also fixing incorrect scales of the virtual model.

Another use could be to progressively enhance simple models of public buildings, such as train stations, air ports, schools etc. with data submitted from users, where the users only would need to submit sequences of photos, which then add more details to a simple initial model of that building.

2. Related work

2.1. Structure from motion

Structure from motion (SfM) was proposed by Ullman [10]. It is an approach to estimate 3D structure from 2D input images, as well as estimating the camera poses for each individual input image.

In my thesis, I use SfM to create the initial point cloud from the set of input images captured with a normal camera. The pipeline built to process and fit this point cloud into the 3D model of the building would also work with any other point cloud, e.g. generated from a lidar scanner. The advantage of using SfM is that it's possible to use my method with only a RGB camera, for example a smartphone camera, not requiring any specialized hardware to capture the data.

2.2. Iterative Closest Point

The *iterative closest point* (ICP) algorithm is an often used approach to register two sets of points, whether they are in a 2D curve or a 3D point cloud.

The original ICP algorithm was proposed by Besl and McKay [1] as well as by Zhang [11].

In ICP, the goal is to find a rigid transformation on the set of points $P = \{p_1, \dots, p_n\} \in \mathbb{R}^d$, which minimizes the distance to the set $Q = \{q_1, \dots, q_m\} \in \mathbb{R}^d$. The transformation can be represented as a rotation matrix $R \in \mathbb{R}^{d \times d}$ and a translation $t \in \mathbb{R}^d$.

The ICP algorithm is, as the name already suggests, an iterative algorithm which requires multiple iterations of two steps to find this transformation:

The first step is the *data association* step, where each point in P is associated to its nearest neighbour (i.e. point with smallest distance) in Q.

The second step is the *transformation* step, where the transformation is updated to minimize the euclidean distance between the points and their respective corresponding points. This transformation usually consists of shifting the centers of mass of the points and their corresponding points onto each other, and then applying a rotation which minimizes the distance between the points. The rotation is usually computed using SVD.

This process is then repeated until a convergence criteria is reached. In each step, the points are newly associated to their corresponding points, as the initial correspondence might not be optimal.

The ICP algorithm is widely used to align two point clouds with each other, usually to reconstruct surfaces from different scans, where the partial scans overlap in some way.

In this thesis, I evaluated directly running ICP on the two point clouds (one from the 3D model and the other from the captured data), to see whether this already yields an usable result. I also use ICP to refine the alignment I get from registering the point clouds on a larger scale, as ICP usually requires an already good initial alignment.

2.3. RANSAC

The *random sample consensus* (RANSAC) algorithm was first proposed by Fischler and Bolles [4].

It can be used to handle data with outliers / noise. It separates the data into inliers and outliers. In the 3D case, this usually consists of fitting a 3D plane into a collection of data points, for example a point cloud.

In the RANSAC algorithm, a certain number of points (in the case of a 3D plane at least 3) is randomly selected, and then the line or plane is fitted into these selected points. In the next step, the algorithm computes the number of inliers and outliers, which results in a "score" for this fitting. Usually, this just means selecting the points which lie within a certain distance to the plane as inliers, while treating all points which lie outside of this threshold as outliers.

This process is then repeated a certain number of times, and after all iterations are completed, the fitted plane with the best score is returned from the algorithm.

In its most basic form, the RANSAC algorithm is very simple, and can easily be implemented and adapted. In my thesis, I use the RANSAC algorithm, both in its normal form, where an arbitrary plane can be returned from the algorithm, as well as in a modified version, where the search space for planes can be restricted to a certain axis in 3D space. I will elaborate more on this modified version in the section about the *method* 4.

2.4. Manhattan world assumption

The *Manhattan world assumption* was first proposed by Coughlan and Yuille [2]. It states that all surfaces in the world are aligned with three main directions, usually called the X, Y and Z directions. While this assumption does not always work for structures found in the real world, as there might be diagonal walls, round shapes or other irregularities, it still holds true for the surfaces I work with in my thesis.

Using this assumption simplifies many tasks in my thesis, as I can for example assume that all planes identified as walls need to be aligned within a certain degree of one of the three main axis to be a valid wall. Furthermore, registration of the point cloud from the 3D model and the point cloud from the captured data is simplified, as I do not need to consider arbitrary rotations in the transformation. The problem simplifies to first align the floor planes of the two point clouds, and then rotating one point cloud in a way that the X and Y principal axis of the two clouds align. Now, the rotation still might be wrong, as the point clouds can have different principal directions, but now I only need to consider 4 possible rotations: 0, 90, 180 and 270 degrees.

3. Research Questions

The main question that the thesis is concerned with is the following:

Is there a way a noisy point cloud generated from 2D input images can be processed in such a way that it can be fit to a 3D model without needing human interaction?

As explained in the section 1.2, the main goal of this process is to augment the CAD model point cloud with data captured from the real world, such that augmented reality applications can use this virtual model to navigate through the real world. Of course, one could process the data manually, fitting the newly captured data in the CAD model by hand, however this task is time-consuming, especially if there are many point clouds to fit into various models.

In addition to the main question, two sub-questions also are relevant for the thesis:

1. *What steps can be taken to reduce the complexity & noise of the captured point cloud?*
2. *Are there any simplifications and assumptions that can be applied to simplify the overall task?*

As I'm working with an unprocessed point cloud as input, it will have some noise in it, as well as points which cannot be used for registration of the two point clouds (or rather negatively impact the accuracy, e.g. from points belonging to furniture in one point cloud which are not present in the other point cloud).

4. Method

4.1. General

The general idea of my approach is to receive a set of images taken by any normal RGB camera and a CAD model of the structure where we want to fit the captured data in (e.g. an apartment). It then processes the images, creating a 3D point cloud, which however still contains a lot of unusable points, in the form of noise as well as points which do not belong to a wall. These points usually make up the furniture and any other objects existing in a normal environment.

While the points belonging to furniture have the potential to add a lot of meaningful structure to the model, they also make if more difficult to actually fit the captured point cloud into the CAD model, as the difference between the two point clouds can be rather large, and as the arrangement of furniture often varies between types of rooms, it's rather difficult to come up with a way to also use these points to fit the point clouds onto each other.

4.1.1 Inputs

The first input is the CAD model of the structure. This CAD model is a point cloud of the whole structure, which might have some inaccuracies, e.g. a wrong wall height. The details on this model are rather low, i.e. only the most relevant structural elements are present, so no furniture or decoration.

For my thesis, I followed a similar approach as I did for my semester project, where I use a cleaned up version of the floor plan of the structure (i.e. no text, door symbols and other symbols), where only the walls are present.

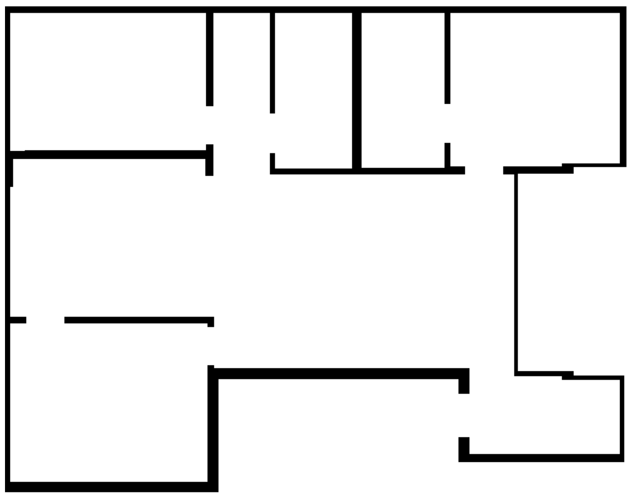


Figure 1. The simplified floor plan used to create the CAD model from

As one can see, this is basically a binary height map, where a black pixel means *full height* (i.e. a wall), and a white pixel is *zero height* (i.e. a floor point).

With this, I then use the *HMM* [2] program, which turns this 2D image into a 3D mesh. The initial height of the model is chosen in a way that it looks plausible. This is a rather inaccurate approach, however many floor plans available to normal users lack any details about the height of the rooms. Other than manually measuring, there isn't a way to get the correct height, but usually one can estimate which height input works for the given floor plan.

After processing with the HMM program, the resulting mesh is then turned into a point cloud using sampling, which results in the point cloud representing the environment I want to fit the captured data in. The sampling step was done with a large sample count, returning a dense point cloud, as downsampling (using uniform downsampling or voxel downsampling) can still be used later in the program if needed.

[2] https://github.com/fogleman/hmm

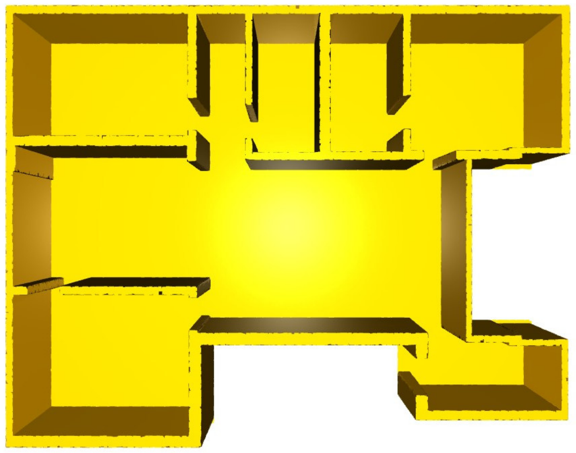


Figure 2. The resulting point cloud from the floor plan

Using *HMM* and the floor plan is of course only one of many possible ways to create the CAD model point cloud. Any other point cloud of the environment will also work.

The second input is the captured data, in form of a sequence of RGB images. The images do not necessarily need to be RGB, as the chosen SfM pipeline (see the following section 4.2) also works with grayscale images. However, most cameras featured in portable devices such as smartphones, digital cameras, drones and often also augmented reality headsets are capable of capturing RGB images.

Using RGB images has the advantage that the resulting point cloud also features colors. This is additional useful information which is added to the colorless CAD model point cloud. This color information can then either be used for localization & navigation inside the environment, or the algorithm could also use color information of previously added data to better fit data in the CAD model point cloud. This is especially the case if the environment is rather large, e.g. a whole building instead of only a part of the building (e.g. an apartment).

The main point to be made here however is that the method works with RGB images, and does not require RGB-D images (i.e. images which also feature a depth dimension). To be able to capture RGB-D images, one needs special hardware supporting this, which limits the number of devices that can be used for this.

The image 3 is an example of an image from the sequence of input images. The images were captured with a Sony Alpha II camera, held on chest height, while walking around the apartment, using the rapid shutter to capture 5 images/sec. The images are not captured in a special way, the process of capturing the images can be described as ”simply walking around and looking around”.

Figure 3. Example input image

4.1.2 Outputs

After running the whole program (which is described in more detail in the following section 4.2), the program returns a new point cloud, where point cloud from the captured data is fitted into the CAD model point cloud, as well as the transformation matrix $T_{final} \in \mathbb{R}^{4\times4}$, which can be applied to the CAD model point cloud to align it with the captured data point cloud.

See the section 4.12 for more details about the resulting point cloud.

4.2. Overview of the steps

In the following sections, I will now show the steps of the method. The steps usually follow each other, for the following general structure:

4.3 Process image sequence into point cloud

4.4 Removing outlier points

4.5 Aligning point cloud with main axis

4.6 Finding relevant planes

4.7 Extracting the structure from the found planes

4.8 Creating a new point cloud from the structure

4.9 Computing the global alignment

4.10 Refining alignment locally

4.11 Merging point clouds

While the steps follow each other, their implementations are independent of the other steps, i.e. the implementation of a step could be changed, as long as the in- and outputs of the step remains the same. This way, individual steps can be changed if another implementation returns better results.

4.3. Process image sequence into point cloud

The very first step in my method is to turn the input images into a 3D point cloud, which then can be processed further.

For this step, a large number of well-working off-the-shelf software exists, and as such, I will use *OpenSfM* [3] for this step.

This step only involves installing OpenSfM on the computer, creating a new directory containing the sequence of images taken with the camera, and running the reconstruction of OpenSfM:

```
bin/opensfm_run_all <image-sequence-directory>
```

This will start the OpenSfM pipeline, which computes a 3D reconstruction from the images.

Please note that OpenSfM can be exchanged by any other suitable SfM program which outputs a point cloud in the `.ply` format. OpenSfM was chosen due to the ease of use, as well because OpenSfM features python functions to process the data after running the reconstruction. A number of these functions were used in some of the other approaches described in section 4.13, however in the final method, these python functions are not used, and as such, OpenSfM could indeed be easily replaced by another SfM software.

After running the OpenSfM reconstruction on the dataset, I end up a point cloud in the `.ply` format, shown in figure 4, with which the method can continue.

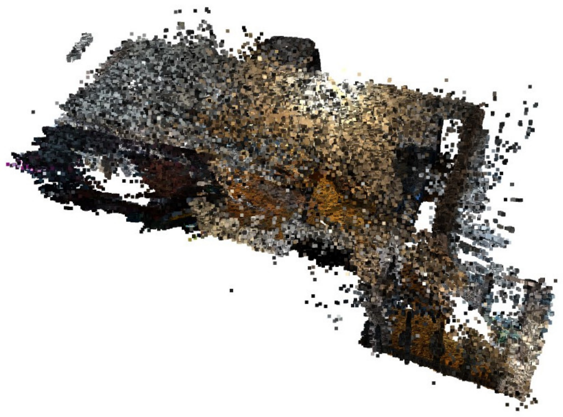


Figure 4. OpenSfM output

When inspecting the point cloud, one can see the issue that I described earlier: The point cloud contains many points which do not contain any meaningful information about the structure of the building itself, e.g. furniture (red in figure 5). Also, the walls have large holes, especially in regions where the SfM algorithm failed to track the movement between frames. One example is on the right side of image 5, where the wall is almost completely absent, as there is only a plain white wall without any structure.

In comparison to the wall, the floor (blue in figure 5) does not contain large holes, this is due to the floor having enough structure for the SfM algorithm to track movement. However, if the floor were to have less structure, e.g. because it's a plain white surface, the SfM algorithm might struggle as well and leave holes in the floor.

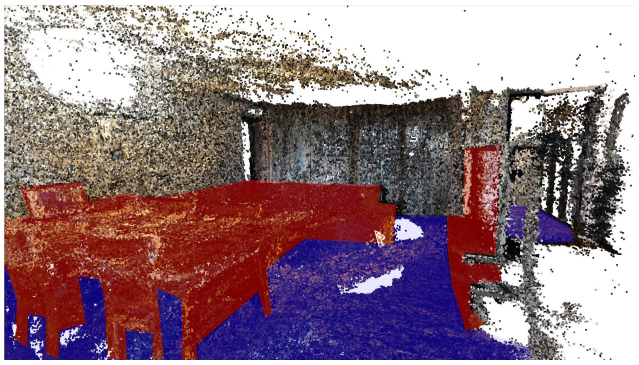


Figure 5. OpenSfM output, detail view with color overlay

Due to this, the point cloud first needs to be processed, removing any points which do not correspond to a point in the structure of the room, and filling in any holes in the walls and floor, resulting in a point cloud that only contains points corresponding to the structure of the building.

4.4. Removing outlier points

As one can see in figure 4, the point cloud has many outlier / noise points, which do not correspond to any real points in the 3D world, and are artifacts from the SfM process. In a first step, I remove these points, such that the point cloud has less noise.

For this, I use the method `remove_statistical_outlier` from Open3d. Another option would be to use `cluster_dbscan` from Open3D[4], which implements the DBSCAN algorithm, which first was proposed by Ester et al [3]. `cluster_dbscan` however does not work for point clouds with many points, as it uses too much memory and the python process crashes.

Both algorithms however are able to identify outlier points, which mainly are noise points, added to the point cloud as artifacts of the SfM process.

[3] https://opensfm.org/

[4] https://www.open3d.org/

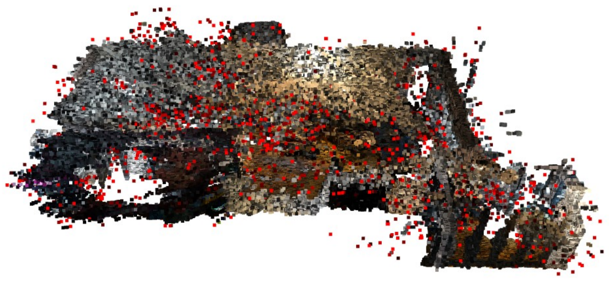


Figure 6. Outliers in the point cloud, marked in red

As one can see in figure 6, mostly points on the outside of the significant point cloud are marked as outliers and therefore removed.

The point cloud in figure 6 does not feature too much noise, however there can be significant more noise. One source of unwanted points are windows, where the SfM approach manages to track points on the outside of the building (i.e. surfaces which are visible from the location of the user).

While of course the simple way to avoid such points being tracked is not capturing these surfaces in the first place, (e.g. by closing curtains), it's not always possible, and as such these points need to be removed as well.

Here, the same approach as for removing noise proved to work well, as these points usually are less dense compared to the points near the camera, and as either of the two previously mentioned algorithms can be used, as shown in 7. In that example, the same apartment was captured, however this point cloud contains many artifacts outside the apartment structure (clusters on the top left and bottom right of the image). However, they were all identified as outliers, and subsequently removed.

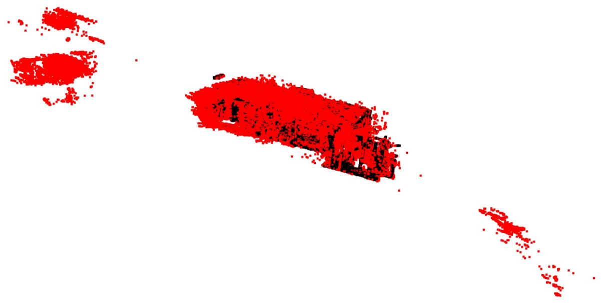


Figure 7. Running DBSCAN on the point cloud containing outside points

4.5. Aligning point cloud with main axis

Right now, the point cloud can have an orientation which does not align with the X, Y and Z axis in 3D space. While the orientation usually is not be completely arbitrary, it still makes the following steps more complicated, and as such, this step aligns the point cloud with the main axis (up to a certain degree, the alignment will not be perfect, but good enough for the following steps).

For this, the first step is to compute the principal components from the point cloud, which will be explained in further detail in 4.6.1.4. This results in a set of 3 vectors, which describe the local main X, Y and Z directions for the point cloud.

To align them with the unit direction vectors in world coordinates, I use the method `align_vectors` from the `scipy` package, which takes two sets of vectors as input and outputs a rotation matrix to align these two sets.

After computing this rotation, it can be simply applied to the point cloud, which is now better aligned with the XYZ vectors in world coordinates, as shown in figure 8.

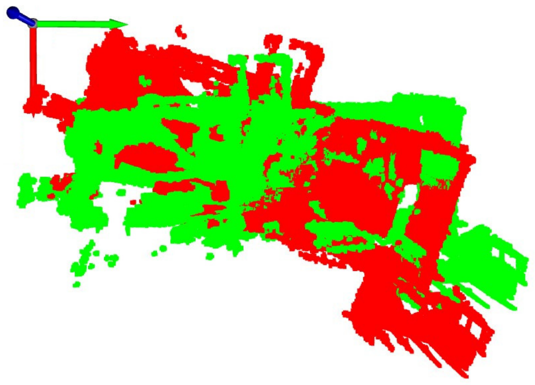


Figure 8. Point cloud before (red) and after (green) aligning

4.6. Finding relevant planes

The next step in the algorithm is to detect relevant planes, which means planar surfaces in the point cloud which either belong to a wall or the floor. As only using RANSAC for finding planes yielded mixed results, I divided this step into three steps: First, I use the RANSAC algorithm to find planar surfaces, and decide whether they can be kept as floor or wall planes. This step is further explained in the subsection 4.6.1.

The second step, which is independent from the first, is finding all planes which make up the outer boundary of the captured space. The RANSAC approach might not find all the planes making up the boundary, and as such I added an additional step to make sure these planes are also kept. This step is explained in 4.6.2.

Lastly, I merge these two sets of planes, as some planes might be found by both approaches (or rather planes very close to each other), which I want to merge into a single plane. I show how this is done in 4.6.3.

The output of this step is a collection of planes, which should lie along the walls and floor of the point cloud.

4.6.1 Find (inner) planes with RANSAC

In the first version, I tried using the RANSAC algorithm to directly find the planes. When running the algorithm on the point cloud from the previous step explained in 4.4, the algorithm usually finds the floor plane. As the RANSAC algorithm is a randomized algorithm, it might find planes which don't belong to any relevant surfaces, e.g. by going diagonally across the point cloud, as shown in figure 9.

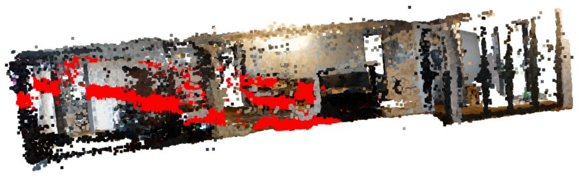


Figure 9. Unusable plane found by the RANSAC algorithm

Another issue with this approach is that running the algorithm multiple times usually finds some planes (as the plane found depends on the choice of the n anchor points), however these usually are either very similar, or only the largest few planes that have a similar count of inliers. For small walls, the likelihood that they are found with this approach is very small (and decreases with increased number of iterations for the RANSAC algorithm), as the chance of always picking n points along these small walls is relatively low.

During testing of the RANSAC method, I also found out that, next to planes being oriented completely arbitrary as shown in figure 9, they also might have the correct alignment in respect to vertical alignment (i.e. have the correct direction for being a wall), but go diagonally through the point cloud, as shown in figure 10. While *in theory*, it might be possible for a structure to have such diagonal walls, many structures do not, and if I apply the Manhattan world assumption 2.4 to this plane, it would also mean that the other planes need to be oriented parallel or perpendicular to this plane.

To resolve this, the process also has an additional step which filters out such implausible planes, keeping only planes which are either horizontal or vertical and align with the walls of the point cloud.

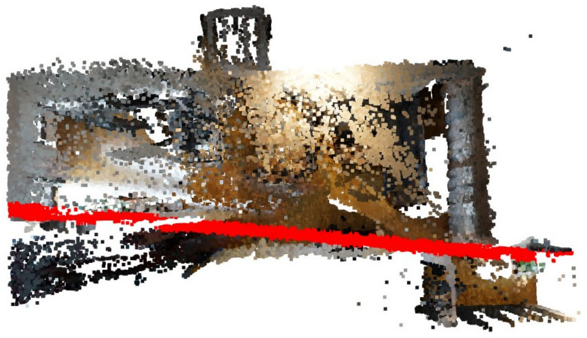


Figure 10. Vertical plane which crosses the point cloud diagonally

4.6.1.1 Fitting RANSAC planes iteratively

As mentioned in the previous section, one issue with RANSAC is that running the algorithm multiple times on a point cloud usually returns very similar planes, which means that most planar surfaces are not found. This is especially true the larger the number of iterations of the RANSAC algorithm is, as the chance of the algorithm picking points which lie on the optimal plane for that point cloud (i.e. the plane with the largest number of inliers) is increased.

For example, running RANSAC 5 times on the point cloud used in the above examples shows that 4 of these planes are almost identical, with one plane being a bit off, as shown in figure 11.

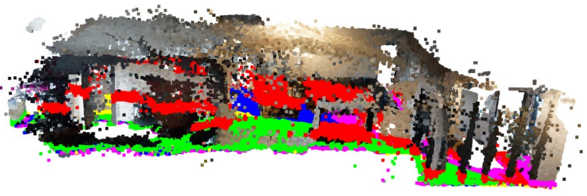


Figure 11. Result of running RANSAC multiple times

This is clearly not an usable approach of finding all relevant planes, especially as there are no wall planes being found, only various horizontal planes which might belong to the floor.

The underlying problem with this is that the points that belong to a RANSAC plane are still available for the next run of the RANSAC algorithm, which means the RANSAC algorithm can pick similar points again in the next run. Therefore, to resolve this issue, the solution is to remove all points which are within a certain threshold of the found plane. With this, the next run of the RANSAC algorithm runs on a point cloud without the points that belong to the previous plane, which means it cannot find the same plane again.

Running the RANSAC algorithm multiple times, now with the additional step of removing the points in between runs, actually results in more distinct planes (i.e. planes which actually differ enough from each other to be considered different planes), with multiple of the planes now being vertical wall planes. The result of this change can be seen in figure 12. While there still are multiple horizontal planes, due to the point cloud having many points on that surface because of the furniture, they are now distinct planes that lie on different heights.

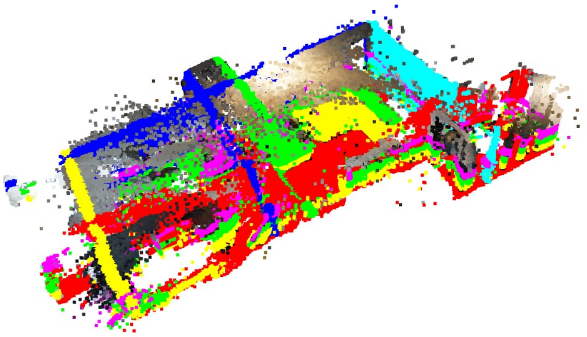


Figure 12. Result of running RANSAC multiple times, removing points between runs

With this change, I'm now able to run RANSAC a configurable number of times n, finding n distinct planes. However, there are now too many planes, which either have an arbitrary orientation or don't align with the direction of the planes.

4.6.1.2 Group planes into vertical, horizontal and other planes

As shown in figure 12, the iterative RANSAC approach is able to find many different planes, many of which are valid, as they lie along the floor or one of the walls. However, as shown in figure 9, the algorithm can also find planes with some other orientation, which are not valid surfaces in the point cloud and therefore should be discarded.

In addition to this, only one floor plane should be kept, all other horizontal planes are usually unnecessary and should be discarded.

With this in mind, I categorize the planes into 3 categories, with different characteristics:

- *Horizontal* planes: Planes which might belong to the floor. Here, I only want to keep 1 plane, ideally the plane really belonging to the floor, discarding other horizontal planes (the process for this is elaborated in the next paragraph 4.6.1.3).
- *Vertical* planes: Planes which might belong to a wall. In this group, I want to keep as many valid planes as possible. Some planes might not align with walls, the process to remove these is shown in paragraph 4.6.1.5.
- *Other* planes: All planes which have an orientation which is not horizontal or vertical, e.g. the plane shown in fig 9. All of these planes should be discarded without any further action, as they cannot contribute anything to the process of finding the structure of the point cloud.

Grouping these planes into these three categories is rather simple, especially as the point cloud was already rotated in such a way that its principal component vectors align with the three main axis, as explained in section 4.5.

This also means that the planes which do not fall into the *other* category have normal vectors which align within a certain threshold with one of the three XYZ normal vectors $[1, 0, 0]$, $[0, 1, 0]$ or $[0, 0, 1]$.

Computing the normal vector of a plane is simple, especially as the planes are in the format of the plane equation $ax + by + cz + d = 0$ (i.e. the planes in the program are a list of the parameters $[a, b, c, d]$). For a plane in this format, the normal vector is simply the vector $[a, b, c]$.

The process of grouping the planes now simply involves computing the normal vector for each plane, and then comparing whether certain coefficients lie near 0 (within a certain threshold):

- If the first (x) and second (y) coefficient are close enough to zero (i.e. $-threshold <= x <= threshold$ and $-threshold <= y <= threshold$ are both satisfied), we know the plane is a *horizontal* plane (the z component must be large in this case, as the vector has unit length).
- If the previous condition does not hold, I compare the third coefficient to the threshold (i.e. whether $-threshold <= z <= threshold$ is satisfied). If this holds, the plane is a *vertical* plane. Please note that this only checks that the plane is vertical, it does not check whether it aligns with the walls themselves.
- Finally, if neither of the previous two checks succeed, the plane is in the *other* category, and can be ignored. The points still are removed, to avoid getting the same plane again in the next run of the RANSAC algorithm.

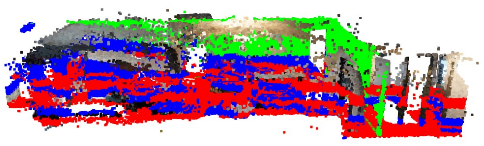


Figure 13. Categories of planes: *horizontal* in red, *vertical* in green and *other* in blue

As one can see in figure 13, the blue planes can be discarded without any further consideration, as they all run diagonally through the point cloud. However, there still are too many red *horizontal* planes, ideally we only keep one plane, which will be shown in the paragraph 4.6.1.3.

4.6.1.3 Discarding invalid floor planes

As described in previous paragraphs and visible in figure 12 and 13, the iterative RANSAC approach still results in multiple horizontal planes being found. As one can see in the figure 14, these horizontal planes contain the floor plane, but also other planes that have an horizontal orientation.

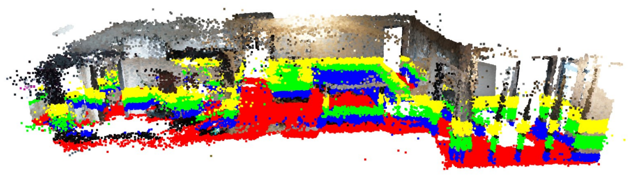


Figure 14. Horizontal planes

This can especially happen if the captured point cloud contains furniture or other structures which add many points to horizontal planes trough the point cloud. In empty rooms, it's less likely to happen, but if the overall size of the footprint of the point cloud is large compared to the surface of the walls in the point cloud, it still may happen.

To remedy this, I added an additional step to check whether a horizontal plane actually is the floor plane of the point cloud:

If a plane is found to be in the *horizontal* group, I count the number of points below the plane. If the fraction of points below the plane compared to the total number of points in the point cloud is lower than a certain threshold, the plane is considered to be the floor plane. In my tests, a number of 1% was shown to be a good number for this threshold.

If a plane satisfying this condition is found, it's marked as the floor plane, and any subsequent horizontal planes are ignored (with their points still being removed for the next run of the RANSAC algorithm).

With this additional step, only 1 floor plane is now left, and n possible wall planes, which might or might not align with an actual wall in the point cloud.

4.6.1.4 Computing PCAs of the point cloud

As mentioned in the previous sections and shown in figure 9, some of the found vertical planes might not align with the walls. Such planes can be removed if they differ too much from the principal components of the point cloud, or rather the principal components of the walls in the point cloud.

For this to work however, I need a way to compute the principal components of the walls in the point cloud.

To find the principal component vectors, I used the method `compute_mean_and_covariance` of Open3D to compute the covariance matrix of the point cloud, from which I then can extract the principal component vectors using eigenvalue decomposition.

One issue with this however is that if I pass the whole point cloud to this method, the returned principal components do not align with the direction of the walls (see figure 15).

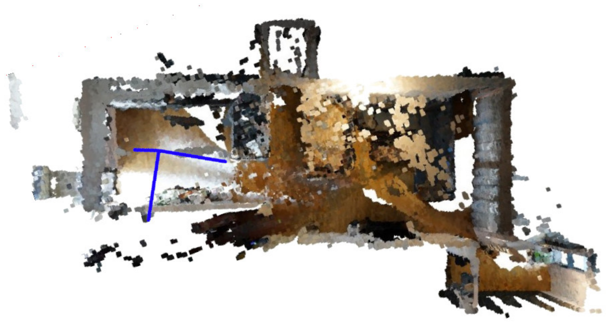


Figure 15. Principal components from running the algorithm on the whole point cloud

The vectors are correct, however the issue is that the points belonging to the small room on the lower right skew the distribution of the points, which means that these vectors are not usable to check the alignment of the planes. For some other point clouds, where the general shape of the point cloud is even more different than the direction of the walls, this issue was even more striking, as shown in figure 16.

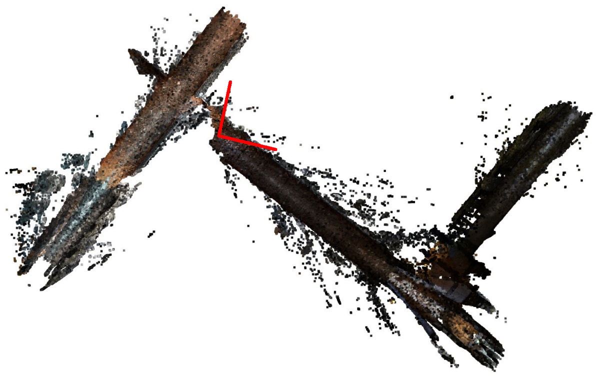


Figure 16. Principal components differing much from the direction of the walls

One approach that I tested as well was to only compute the principal components from the points belonging to the floor plane (which usually is found as the first plane with normal RANSAC), however, the distribution of these points usually also follows the same shape as the whole point cloud, and as such this did not improve the quality of the found principal component vectors.

During testing of the previous approach, I noticed that the first planes found by the RANSAC algorithm usually do correspond to actual walls in the point cloud (see the section 4.6.1.1 for an explanation how multiple planes can be found), and as such these might be used to compute the principal components.

What still was not possible was to only include the points making up these planes, especially if they were perpendicular to each other, this again yielded unusable vectors, which again was no surprise, as the shape of this "filtered" point cloud still was skewed in respect to the direction of the planes, as shown in figure 17.

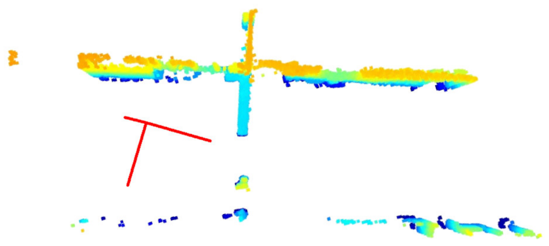


Figure 17. Principal components computed from points on walls

The next approach however then finally yielded better results: Instead of combining all the points of the first n planes into one point cloud, I compute the principal vectors for each plane point cloud individually, and then merge these together.

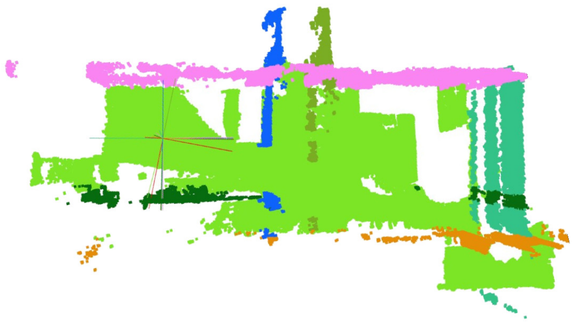


Figure 18. Principal components computed from individual walls

Shown in figure 18, the red vectors are the principal components computed from the whole point cloud (inserted for comparison), and the other vectors are colored the same color as the point cloud they were computed from.

As one can see, the only of these vectors that is off is the light green one, which belongs to the floor point cloud, the others all align relatively well with each other, as well as the walls.

Only keeping the principal components from the walls, discarding the set from the floor, these vectors are averaged, which then resulted in better principal components vectors, as shown in figure 19.

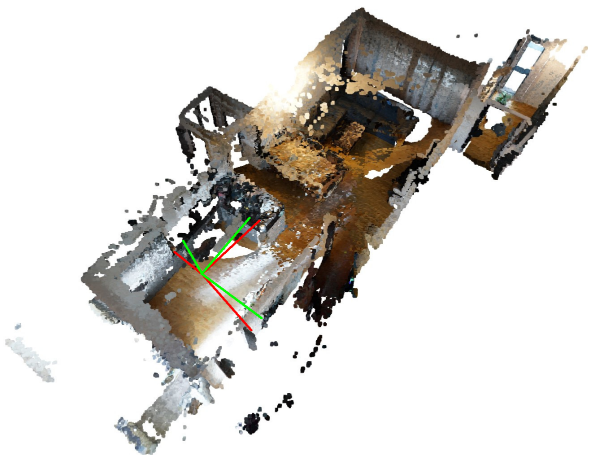


Figure 19. Comparison between principal components computed from the whole point cloud (red) and the merged components from the first n walls (green)

4.6.1.5 Discarding planes differing too much from PCAs

With the principal components of the walls that were computed previously, I can compare the orientation of the vertical planes with these principal components, and discard the plane if it differs too much from the principal directions.

As in the paragraph 4.6.1.2 for grouping planes, I can use the normal vector of the planes to check this alignment. For this, I simply check if the dot product of the normal vector with the X or Y principal component is close enough to 1 (normal vector is parallel to the principal component) or -1 (normal vector is anti parallel to the principal component).

For the notion of "close enough", I again use a configurable threshold, where a number of 0.99 has shown to work reasonably well.

After running this additional check, planes which do not align with the principal components well enough are discarded, and a collection of vertical planes which align well enough with the walls in the point cloud is returned. A visualization of this step is shown in figure 20.

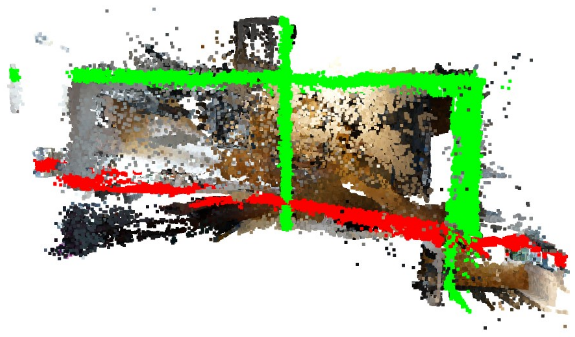


Figure 20. Planes aligned with the principal components (green) and differing too much (red)

Please note, that after this step, there still might be planes that do not correspond to a wall in the point cloud, e.g. when there are many points from furniture or other objects along a plane in the point cloud. These planes however are removed in a later step.

4.6.1.6 Final result of finding (inner) planes with RANSAC

After running all the steps shown in the previous paragraphs, the result is a collection of planes, which include the horizontal floor plane, as well as a set of n vertical wall planes.

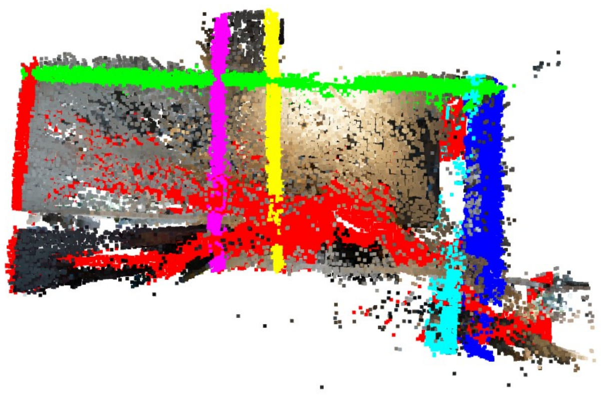


Figure 21. Final result of running iterative RANSAC with cleanup steps

Using this approach, it's possible to find many different planes, especially also larger planes in the inside of the point cloud, however there might be holes in the outer boundary of the point cloud, as some of these walls are rather small. In the figure 21, the absence of a plane at the very top of the point cloud, as well as on the bottom of the larger room and the boundaries of the smaller room on the bottom right are visible, and in this state, the result cannot be used, as the planes do not form a closed boundary around the rooms in the point cloud.

4.6.2 Find outer planes

As explained in 4.6 and visible in figure 21, only using the RANSAC approach results in a set of planes which might be missing some of the planes making up the boundary of the point cloud.

To find these planes as well, I added a second step, which computes the outer planes for a point cloud, making sure the point cloud is captured entirely by the computed planes.

4.6.2.1 Constructing a FlatPointcloudGrid

First, a `FlatPointcloudGrid` is constructed from the point cloud. This is a custom class, which divides the space that the point cloud occupies into bins. The number of bins is a configurable parameter of this class, where the longer side of the grid contains this number of bins.

For each point in the point cloud, the grid then computes to which bin it belongs to by projecting the point onto a plane, increasing the counter of elements contained in that bin by 1. With this, I get a heat map for the point cloud, where one can query the number of points in a bin, as shown in 22.

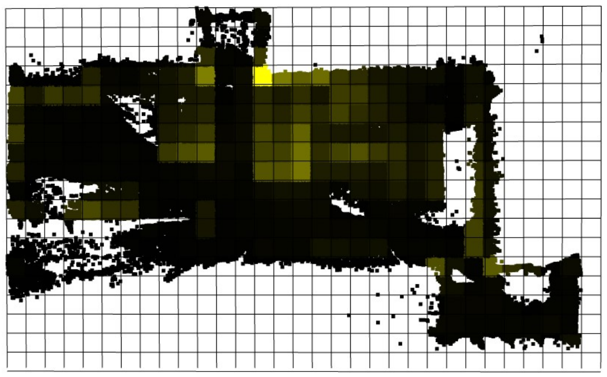


Figure 22. Heat map of the distribution of points, yellow bins containing the most points

The `FlatPointcloudGrid` also has methods where the user can query the bin of a 3D point and the number of points within that bin.

4.6.2.2 Finding outermost bins & consecutive filled bins

With the `FlatPointcloudGrid` from the previous step, the program now steps through the grid and finds for each of the 4 directions (coming from positive X and Y as well as from negative X and Y) the first bin that contains more points than a certain threshold. Stepping trough the grid is done in a nested loop, e.g. when stepping trough the positive X direction, for each X value, all the bins along the Y line for this X value are considered. If a bin with a large enough threshold is found, the other bins along this (inner loop) index are then not considered anymore, as there already is a bin further out in the grid that has enough points.

The threshold is required, as some outlier points, or rather groups of outlier points, still exist in this point cloud, and they should be ignored for finding the bounding bins. One example of applying this filter is shown in figure 23.

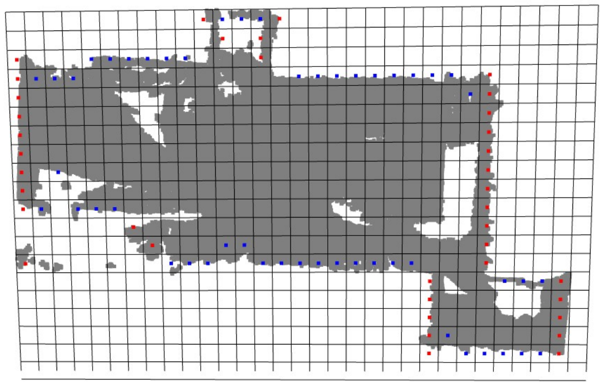


Figure 23. Outermost bins, red = positive and negative X direction, blue = positive and negative Y direction

During this process, I keep track whether the previous bins were marked as containing enough points or not. If enough consecutive bins contain the required number of points, the current slice is marked as containing a plane.

After the algorithm stepped through the grid in all 4 directions, I have a list of X rows and Y columns, where the algorithm found enough consecutive bins for the column or row to contain a plane.

With these two lists, I can then proceed and compute the planes that can be fit into these rows and columns.

4.6.2.3 Fitting planes into bin point clouds

With the rows and columns computed in 4.6.2.2, I now gather the points that belong to the columns or rows, creating several point clouds which each need a plane fitted in (see figure 24).

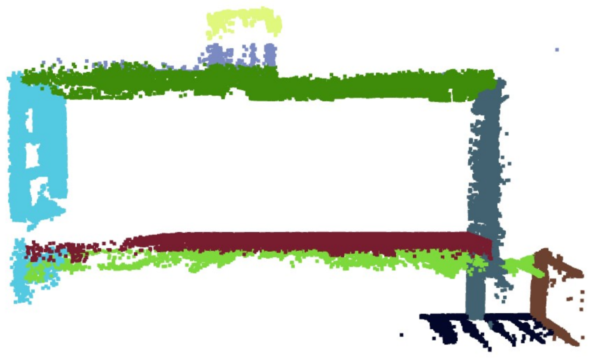


Figure 24. Point clouds found by keeping only relevant points

With these point clouds, I can now compute the planes that fit into them. However, I noticed during testing that using normal RANSAC (even with the PCA constraint shown in 4.6.1.5) still results in incorrect planes, as some of these point clouds are rather small and therefore fitting a plane inside might not yield the desired result.

However, as I know for each of these point clouds whether it belongs to a plane oriented in the X or Y direction, I can use an approach I call *constrained RANSAC*.

Usually, in normal RANSAC, the n points to fit the plane in are picked randomly. In this version, I only pick one point from the point cloud at random, and then construct a plane from this point and the normal vector (which needs to be perpendicular to the heading of the plane, i.e. to the principal component for this direction).

With this plane, I then apply the normal logic for RANSAC: Compute the number of inliers, and compare if this plane has a larger number of inliers compared to the previous iteration. This approach forces all planes to be parallel to each other (following the direction given by the principal component for this direction), only moving the planes along their normal vector.

The constrained RANSAC approach is further explained in 4.13.6, as I also tried to find planes with this approach in the whole point cloud. The issues with this approach are shown in the previously mentioned section.

Running this constrained RANSAC approach on all the slice point clouds then results in a collection of planes, which consist of all planes that were found in the step explained in 4.6.2.2.

4.6.2.4 Final result of finding outer planes

Finally, after running this step, I end up with a set of planes, which contain most of the bounding planes for the point cloud, shown in figure 25.

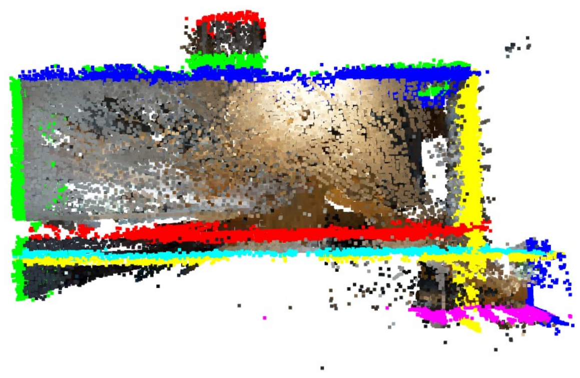


Figure 25. Final result of running the steps for finding outer planes

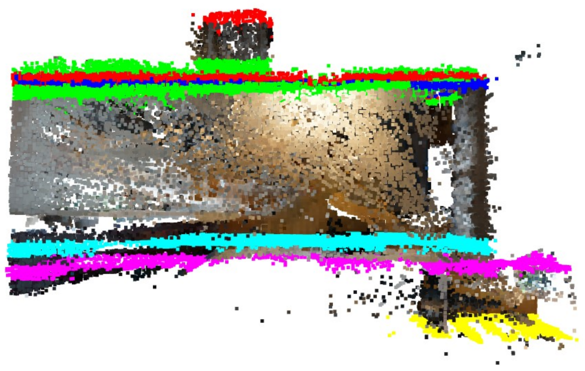


Figure 26. Set of X planes

When compared to the result from using RANSAC in figure 21, some planes are missing from this result, while some other planes that are missing in the RANSAC result are present here.

Also, as one can see, both algorithms do find some very similar planes, which is why the next step shown in 4.6.3 is needed.

4.6.3 Merging the found planes

After running the steps outlined in the previous sections 4.6.1 and 4.6.2, I end up with two sets of planes, the RANSAC set from 4.6.1 and the boundary plane set from 4.6.2.

The goal of this step is now to merge these two sets into a single set of planes, such that the planes capture as much of the structure of the point cloud as possible.

4.6.3.1 Grouping planes into X and Y planes respectively

First, I group the planes in both sets into a set of X and Y planes. Visible in figure 25 for example, some planes might be very close to each other and should be merged.

Grouping these planes is simply a matter of determining the main direction the plane is heading, and creating an X and Y set of planes, where each set contains the planes from the RANSAC step, as well as from the bounding planes step.

After this, I have 2 sets of planes, with an additional floor plane from the RANSAC step. For example, see the figure 26 for the X set of planes.

4.6.3.2 Merging planes

Next, for each of the sets, the planes are merged into each other. This involves iterating over each pair of planes, and checking whether they are close enough to each other to be merged into each other. However, one can not simply merge these pairs into each other, as there might be a constellation where multiple planes need to be merged. Consider the following example:

Plane B is in the threshold for merging with plane A, but plane C is too far from A, however it is close enough to B to be merged. This means that we need to collect the set of ”planes to merge with this” for each plane, and then check if there’s an overlap with another set. If there is, these two sets need to be merged into one, and all the planes in this set need to be merged into one plane.

For each of these sets, I then keep the plane which contains the most points, and discard the other planes, which only keeps one plane per set.

4.6.3.3 Final result of merging planes

Finally, after merging the X and Y planes into each other respectively, I concatenate these two sets into one, and add the floor plane from the RANSAC step, which results in a single set of relevant planes.

For each point cloud belonging to a plane (i.e. all points which lie within a certain distance threshold to the plane), I also compute the bounding box, as this is required for the next step. An example of these bounding boxes are shown in figure 27.

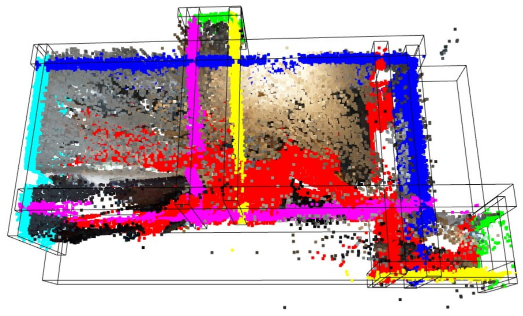


Figure 27. Merged planes with bounding boxes

These sets of planes and bounding boxes can now be used for the next step in the pipeline, which is computing the intersections of these planes and deriving a structure from these intersections.

4.7. Extracting the structure from the found planes

In this section, I will show the approach to extract the structure from the set of planes computed in the previous step. This part of the program is divided into various segments, which continuously process the data, discarding any data which is not part of the relevant structure of the point cloud.

4.7.1 Computing plane intersections

The first step is to compute all relevant intersections of the planes received from the previous step 4.6.

For each pair of planes from this set, I compute the intersection of them. Usually, an intersection can be found for every pair of planes, even for planes heading in the same direction, as the planes are usually not perfectly parallel to each other.

If there is the rather unlikely case that there are two planes that actually do run parallel, the algorithm simply skips this pair and moves onto the next pair.

If the intersection does exist, the algorithm to computing the intersections returns a line, represented by a point p_i and a direction vector d_i.

As planes in 3D space are indefinitely large, this means that many intersections lie far outside the point cloud, which is visualized in figure 28. The point cloud is the small colored spot in the denser region of lines. As one can see, many intersections are far outside the point cloud, which means they are irrelevant for this step.

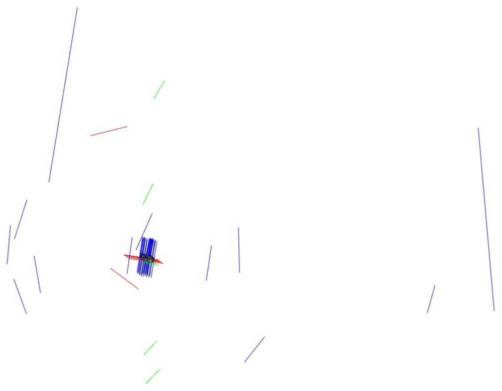


Figure 28. Visualization of parts of the intersection lines

To solve this issue, I use the bounding boxes of the planes computed in the previous step, and only consider segments of the lines that are within at least one bounding box of the two planes in a pair of planes.

During this step, the intersection lines are also divided into X, Y and Z lines, based on the maximum coefficient of their direction vector d_i, as the horizontal lines need to be processed different from the vertical lines.

As the bounding boxes for walls might not stretch all the way down to the floor (e.g. if the wall only has points above a certain height because tracking below this height failed), this step also ensures that the lines along the Z axis reach at least the floor, such that they can be used for processing.

The result of this step are three sets of lines, visualized in figure 29.

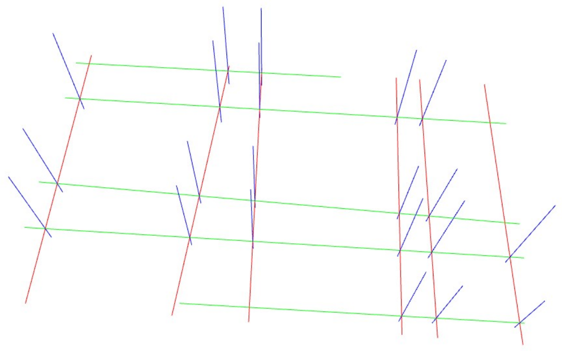


Figure 29. X, Y and Z line sets

At the start of this section, right after computing the intersections between the planes, each line was indefinitely long, parameterized as a point $p_i \in \mathbb{R}^3$ and a direction vector $d_i \in \mathbb{R}^3$. After this step, the lines are finite lines between two points, referred to in the following sections as *line startpoint* and *line endpoint*.

4.7.2 Computing intersection points of lines

With the intersections now computed and grouped into X, Y and Z lines, the next step is to compute the intersections between the lines on the floor, referred to as X and Y lines respectively.

For each pair of lines in X and Y, I compute the closest distance between lines, which also returns the point on each line that is the closest point to the other line.

If the closest computed distance is larger than a certain threshold, the lines are considered to not intersect, and the two points on the line are discarded. If the distance is close enough to 0, the lines are considered to intersect, and therefore the points are added as intersection points to a list of such points. As floating-point operations introduce some inaccuracies, the distance usually is larger than 0, however the computed distance for intersecting lines usually lies within the $1e - 14$ range.

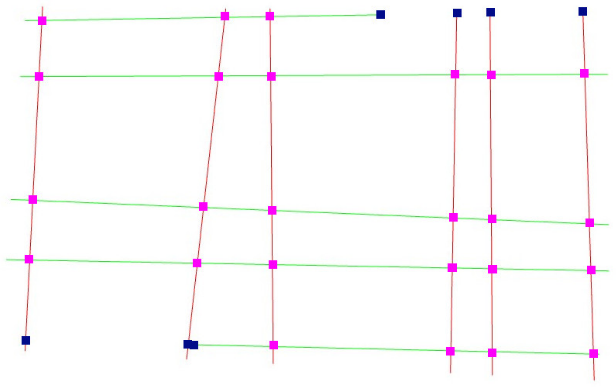


Figure 30. Real intersection points (purple) and points too far apart (blue)

As one can see in figure 30, the only points which are too far apart (blue) are on pairs of lines which do not intersect.

Going forward, only the real intersection points (purple) are considered, with the others being discarded after this step.

4.7.3 Discarding z lines outside the point cloud

In the next step, the Z lines (upwards lines, colored blue in 29) need to be processed. For each of these lines, the program checks whether there are any points near the line, using the `FlatPointcloudGrid` first introduced in 4.6.2.1. If there are no points in the same bin as the line, the line is considered to be outside of the relevant point cloud, and can be discarded.

If there are any points nearby, the line is kept for now, which results in a cleaned up set of lines, shown in figure 31.

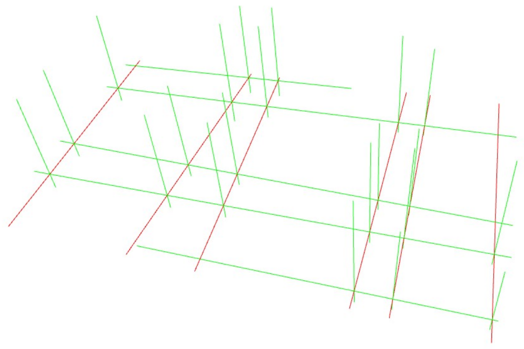


Figure 31. Visualization of valid (green) Z lines

4.7.4 Remove intersection points without corresponding z line

After filtering the Z lines in the previous step, the intersection points are filtered again, this time only considering the real intersection points found in the previous step 4.7.2.

For each intersection point I_{ij}, which is the intersection of the two intersection lines between the plane X_i with the floor and the plane Y_j with the floor respectively, we know that there should also be a corresponding Z line Z_{ij} of the intersection between the planes X_i and Y_i. If Z_{ij} is absent, the intersection point lies outside any bounding boxes of the point cloud, and therefore outside of the relevant space, which means it should be discarded.

As before, due to floating point imprecision, the distance usually is not exactly 0, but in the range of $1 - e14$ if Z_{ij} actually does intersect the point I_{ij}.

After running this step, all intersection points that are still present should be within the point cloud. As visible in fig 32, all points I_{ij} which do have a corresponding Z_{ij} line were kept, with only points on the outside removed.

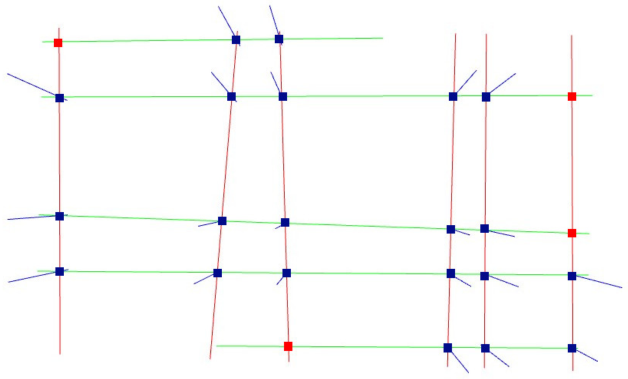


Figure 32. Removed intersection points (red) and kept intersection points (blue)

4.7.5 Clamp lines to their endpoints

With the intersection points now only existing inside the point cloud, the X and Y lines can now be processed in such a way that they only exist between their outermost intersection points, as parts of the line extending beyond these points are not relevant.

This step is rather simple, and simply involves finding all intersection points which lie along a line, and then compute the intersection point closest to the start of the line, as well as the intersection point closest to the end of the line. This results in 2 points, which now are the new start and end of the line.

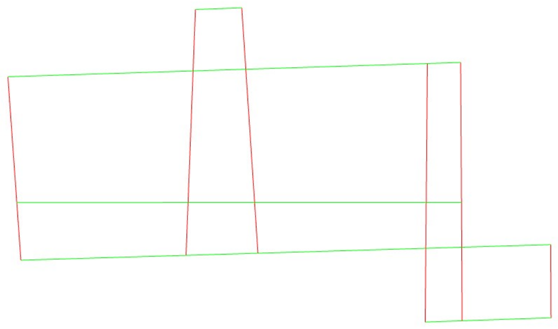


Figure 33. Lines clamped to their new endpoints

After this step, the structure of the point cloud now appears for the first time, with the lines enclosing the point cloud rather well. However, as one can see in figure 33, there are still many lines inside the shape, which do not correspond to an actual wall in the point cloud.

4.7.6 Filter z lines by neighbourhood

As there are still many lines inside the point cloud which do not correspond to an actual wall, the program needs to filter these. For this, the Z lines are again considered.

First, the set of Z lines is divided into a *boundary* set, which contains all Z lines which intersect a start- or endpoint of a line, as well as an *inside* set, which contains the other Z lines.

The boundary lines should be preserved at all times, as bounding lines are all considered to be valid, however the other lines can possibly be removed.

For a Z line Z_{ij} in the inside set to be considered valid, the line needs to be near (or optimally on) the intersection between two walls X_i and Z_j, where both walls should contain some points near this intersection (i.e. both walls should exist at the intersection). If this does not hold and there are no points (or rather not enough, as there can still some outlier points in the point cloud) near this intersection Z_{ij}, it can be discarded. Please note that for this step, only the points belonging to the walls are considered (i.e. all points within a certain distance to a wall plane), as the floor usually has points everywhere within the point cloud, even if there is no wall.

All Z lines which have fewer points near themselves than the threshold are considered *low* Z lines, which will be discarded, all others are *high* Z lines, which should be kept.

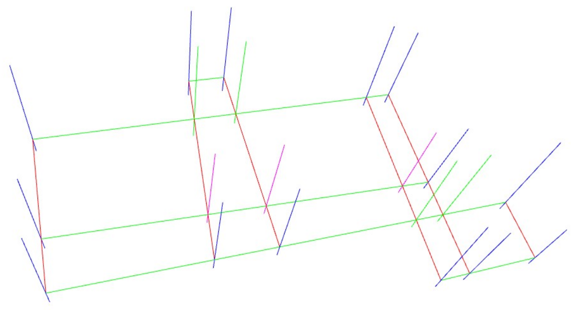


Figure 34. Boundary Z lines (blue), high Z lines (green) and low Z lines (purple)

As one can see in figure 34, some of the Z lines inside the point cloud are indeed marked as low Z lines, which results in them being discarded. Z lines on the boundary, but on an inner corner cannot be marked as boundary lines with this approach, however they are still kept, as there usually are enough points at these locations for the lines to be considered high segments.

4.7.7 Divide lines into segments and remove unneeded segments

Until now, the lines exist along the whole length of the line, i.e. between the start- and endpoint computed from the intersection points. Some of these lines however need to be divided, as they either might be part of an inner wall, which only exists along some segment of the line, or an outer line which might be divided by a hallway or door.

As a preparation for this filtering step, the lines are divided into segments by the intersection points laying along each line.

For this, the intersection points on each line are sorted by distance to the start point of the line, and then each consecutive pair of points in this sorted list makes up a new line segment, as shown in figure 35.

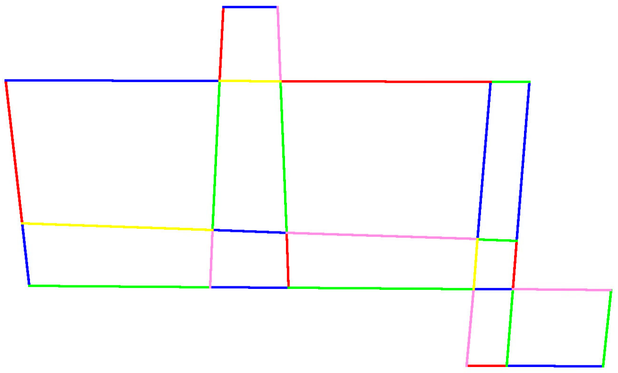


Figure 35. Lines divided into segments

In the previous step, some of the Z lines belonging to intersection points were removed, as they did not have enough points nearby, and as such were considered to be nonexistent intersection lines in the point cloud. After this step, only the set of kept Z lines was remaining, and this set is now used to remove invalid line segments.

For this, each line segment is checked whether the start- as well as the endpoint are still on one of the remaining Z lines. If the Z line Z_{ij} belonging to an intersection I_{ij} of two planes X_i and Y_j was previously removed, we know that there are not enough points near the intersection for it to exist in the point cloud. If either the start- or the endpoint (or both) are not on such a line, the segment is marked as to be discarded, as shown in figure 36.

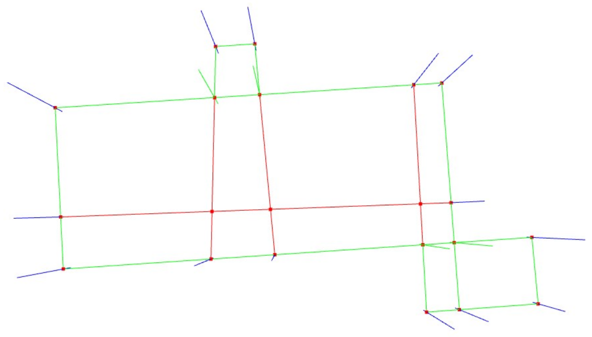


Figure 36. Kept segments (green) and discarded segments (red)

In the figure 36, one can easily verify that all segments existing between two Z lines are kept, while the other lines are removed. With this step, most of the unneeded inner line segments are removed, while the segments along the boundary are kept.

4.7.8 Compute boundary segments

This step is a preparation for the next step, where we're only interested in the line segments on the inside. This time however, not the Z lines are grouped, but the line segments. For this, we construct a new point cloud from the X lines by sampling along them, and then construct a `FlatPointcloudGrid` from this sampled point cloud. For each line, we now can check whether there are any lines above or below it in the `FlatPointcloudGrid` by simply checking the bins above and below a point on the line. If this approach finds that either above or below the line are no bins with points inside, we know there isn't any other line on that side, and therefore the line must be a boundary line.

If the program finds a line above and below, the line must be on the inside of the point cloud, and can be processed in the next step.

This process is then again repeated for the Y lines, which results in a set of boundary line segments and inner line segments, as shown in figure 37.

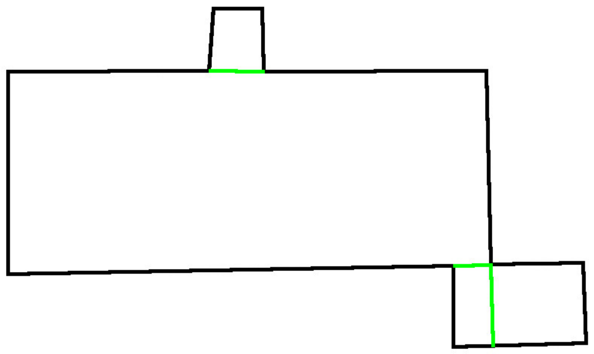


Figure 37. Boundary segments (black) and inner segments (green)

All the boundary segments that were found by this step are now definitely part of the final result, while the inner segments still need to be processed, as described in the next step.

4.7.9 Apply grid filter to inside line segments

Finally, as a last step in the process of finding the structure in the point cloud from the planes, the program needs to remove any leftover inside segments which don't have enough points along them.

Most planes that do not have any points along them were already removed in step 4.7.7, as at least one of their endpoints is not along one of the kept Z lines.

However, some segments do have both endpoints along one of the Z lines, but no points on the segment itself. One example of such a segment is a doorway, where there is a wall to the left and right.

To filter such segments, the `FlatPointcloudGrid` is again applied (again only considering wall points), and the bins (with the number of points inside them) along a line are counted. If most bins don't have any points, or the number of bins containing only a small number of points is too high compared to the number of bins containing many points, the segments is removed by the algorithm.

With this approach, the segments existing in spaces without enough points are removed, which means that structures such as doorways are now not blocked by a segment anymore. The result of this step is shown in figure 38.

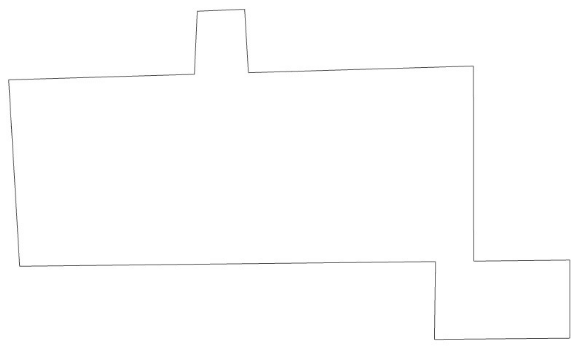


Figure 38. Lines being kept after filtering

4.7.10 Final result of extracting the structure

After running all the steps described in this section, I end up with a set of lines, which represents the structure extracted from the planes and their intersections in the point cloud. While some details might be missing (e.g. smaller walls on the inside) and some planes inside the point cloud might be useless, the general shape of the the point cloud is rather well represented. This set of lines is passed to the next step in the program, which processes them further.

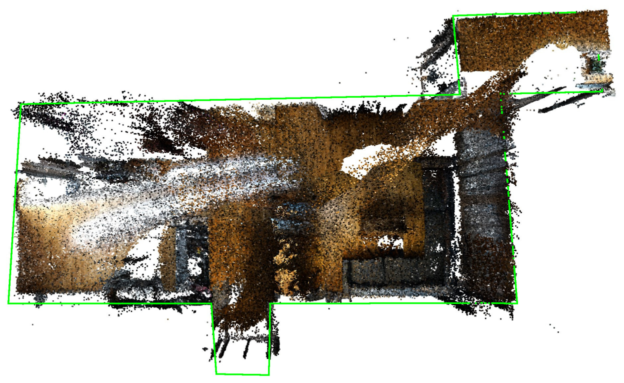


Figure 39. Final result, visualized with the point cloud

The output of this step is the total set of lines L_t, as well as the set of boundary lines L_b and inner lines that were kept L_i, for which we have $L_t = L_b \cup L_i$ and $L_i \cap L_b = \varnothing$.

4.8. Creating a new point cloud from the structure

The main point of extracting the structure of the point cloud as a set of lines in the previous step was to be able to only keep relevant points of the cloud to fit it into the CAD model.

Theoretically, one could use the points from the captured point cloud, keeping only the points on the extracted lines, and fit this in the CAD model. However, as already pointed out in the previous section, the point cloud has holes in the walls and the floor, which could impact accuracy of aligning the point clouds.

Therefore, as the next step in the program, the extracted structure is turned into a new point cloud, which does not have any holes in the walls and floor.

The input for this step are the point cloud from the captured data, the previously computed set of lines L_t as well as the set of boundary lines L_b. An optional argument is the number of points to sample, with the default set to $100'000$. This allows the user to sample more or fewer points, depending on the use for the sampled point cloud.

4.8.1 Sampling line segments

The first step is to sample the line segments themselves, turning them into walls for the newly created point cloud.

To evenly distribute the points across all segments, the total length of all segments l_s is computed. The number of desired samples is then divided by this number l_s, giving the number of samples per segment unit length.

Sampling the walls is then simply a process of iterating over each segment, picking a random point along this segment (i.e. between the start and end of the segment), using an uniform distribution. This results in a point cloud with all points lying on a single plane, as visible in figure 40.

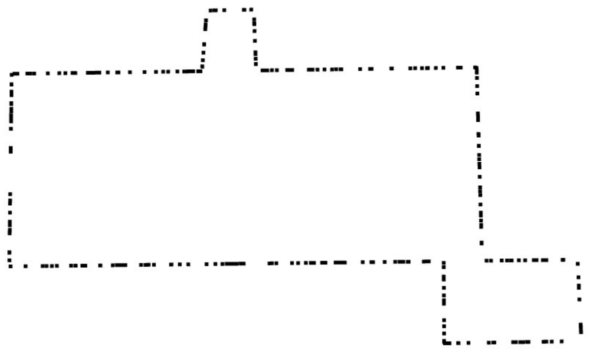


Figure 40. Sampled point cloud with all points on the same height

As one also can see in the figure 40, the points are spread out over the various segments, the short segments do not have significantly more or fewer points than the large segments. Due to the sampling using an uniform random distribution with a low number of samples (300 for this example), the points aren't evenly spread out. When the number of sampling points is large enough, the holes in the lines disappear. I don't use a fixed distance for sampling, as this would introduce a regular structure in the level of points which I want to avoid.

The points now still all have the same height, as they all lie on the line segments, which means I need to add the height of the point as an additional independent sampling variable. However, for this, I first need to compute the correct height of the point cloud, such that the sampled point cloud matches the captured point cloud in height as exactly as possible.

4.8.2 Computing Z height

One approach to compute the Z height would be to use the bounding box of the point cloud to compute the height from, however the two types of bounding boxes available in Open3D (`AxisAligned` and `Oriented`) usually are higher than the distance between the floor and the ceiling of the point cloud.

This can be due to the point cloud having some clusters of outliers (which were not removed by the outlier removal step shown before), or also because the point cloud might not be perfectly aligned with the main axis of the coordinate system. An example of this is visualized in figure 41.

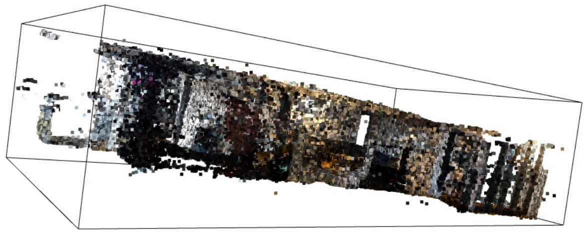


Figure 41. Axis aligned bounding box on point cloud

As one can see, the bounding box is higher than the point cloud, which means I can not simply use the height of the bounding box to compute the height of the point cloud.

From the previous steps, it is known that it's usually rather simple to find the floor plane, especially when using the constrained RANSAC approach shown earlier. In the current step, it's even easier to find the floor plane, as all the previously computed line segments lie on the floor plane. This means I can simply select three points from the set of start- and endpoints and compute the floor plane from these.

After computing this floor plane, I use the normal of this plane to shift the plane in the direction of its normal in the point cloud. After shifting the plane, I compute the percentage of points still above the plane. If this number is above a certain threshold (a threshold of 1% has shown to work well), the shifting step is repeated, until this threshold is reached.

With this, the ceiling of the point cloud can usually be approximated rather well. Of course this approach does not work if the point cloud does not reach up to the ceiling, but with this approach it's assumed that at least some parts of the point cloud do reach the ceiling.

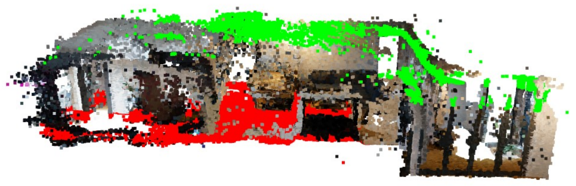


Figure 42. Top (green) and bottom (red) plane in the point cloud

After computing this upper plane, I simply compute the distance between the two planes, which results in a good approximation of the height of the walls in the point cloud.

With this height I can now also sample the Z value of a point randomly, which results in the walls actually reaching up to the ceiling of the point cloud, as shown in figure 43.

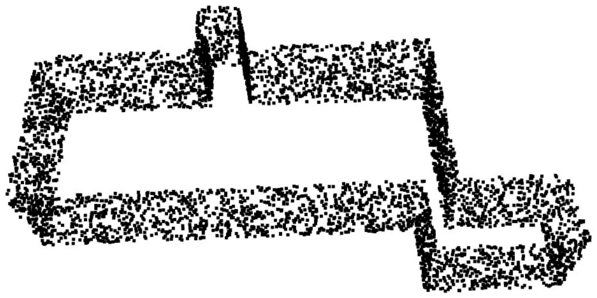


Figure 43. Sampled point cloud with walls

4.8.3 Scattering the points

For the next step explained in 4.8.4, I need the bounding boxes of the boundary wall segments, however, Open3D cannot compute a bounding box if all points lie on the same 2D plane.

To fix this problem, I introduce a small bit of imprecision when sampling the points, i.e. the x and y values of each points are shifted a random amount in positive or negative direction. The interval the random amount is chosen from is rather small, however it's already enough that the sampled points on a wall segment don't form a 2D surface anymore, but a 3D object, from which the bounding boxes can be computed.

4.8.4 Sampling the floor

The sampled point cloud has so far no points on the floor, as I only sampled points on the wall segments. The floor plane was already computed before, so I can use this plane to sample points.

One issue is that a plane is indefinitely large, which means simply sampling arbitrary points on this plane would not work, as the points might lie inside the point cloud, but could also lie anywhere else on this plane.

However, from the already sampled wall points, I can compute the bounding box of the whole point cloud, and constrain the points to lie within this box. To sample a point on the plane, I then simply pick an x and y value uniformly distributed within this bounding box and then compute the z value as:

```
x = random(bounding_box_min[0], bounding_box_max[0])
y = random(bounding_box_min[1], bounding_box_max[1])
z = (-a * x -b * y - d) / c
```

This results in a random point on this floor plane. As one can see when visualizing this plane (figure 44), many points still lie outside the walls, especially if the bounding walls do not form a rectangular shape.

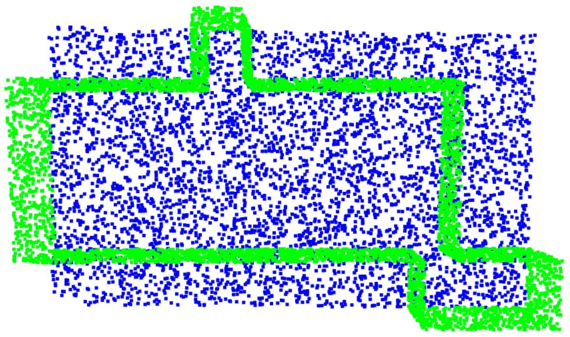


Figure 44. Floor points outside of the walls

To only keep points inside the walls, I copy each sampled point outside the space enclosed by the walls, and then trace a line between the sampled point and the corresponding point on the outside. I then compute the number of bounding boxes that intersect with this line (the bounding boxes only exist for the boundary walls). If the number of intersections is odd, the sampled point lies inside the walls and can be kept, if the number of intersections is even, it lies outside and can be discarded.

With this check, I now have a good result. There are some artifacts remaining, this is likely due to the bounding boxes slightly overlapping, but these few points could be removed in a post-processing step.

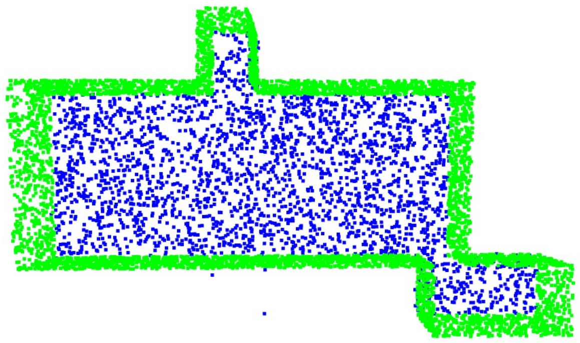


Figure 45. Sampled point cloud

When visualizing the sampled point cloud and the captured point cloud (figure 46), one can easily see that the sampled point cloud is a good approximation of the structure of the captured data.

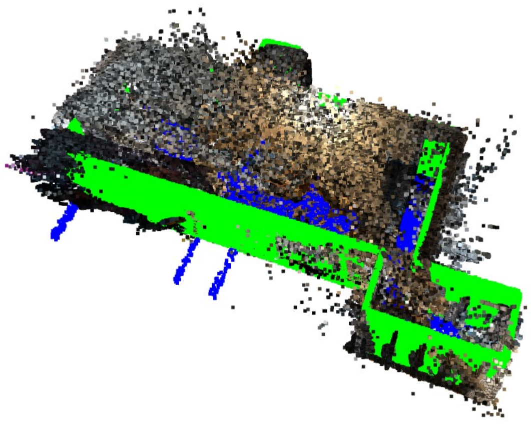


Figure 46. Sampled point cloud aligned with captured data

4.9. Computing the global alignment

The next step is computing the transformation needed to fit the captured data into the CAD model point cloud as well as possible.

4.9.1 Inputs

The inputs for this step are simply the sampled point cloud, which is an approximation of the structure of the captured data, as well as the CAD model point cloud. Optionally, one can additionally input a point cloud containing only the points from the sampled floor, this simplifies computing the floor plane for the next step.

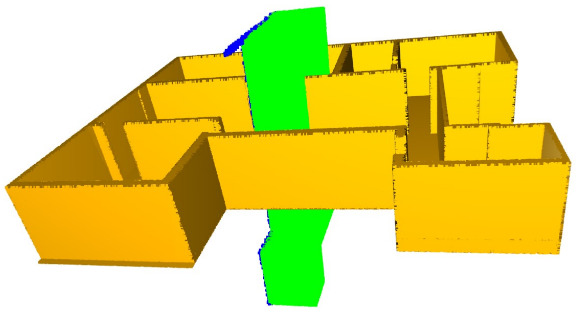


Figure 47. CAD model point cloud (yellow) and sampled point cloud (green & blue)

4.9.2 Aligning floor planes

As the orientation of both point clouds still might be completely arbitrary (as seen in figure 47), the first step is to align the floor planes, such that this orientation is already correct.

To do this, the program first computes the floor plane of both point clouds, which can easily be done as explained in the previous sections. From both planes, I compute the normal of the plane, and then use a method from the `scipy` package to compute the rotation needed to align these two vectors. Using this method proved to yield more accurate rotations compared to a version I implemented myself.

After computing this rotation, the program applies the rotation to the sampled point cloud, which now should align better with the CAD model point cloud. However, it might be oriented the wrong way, as the normals might now be anti parallel to each other. In this case, the sampled point cloud is flipped, which now also result in correctly oriented planes, seen in figure 48.

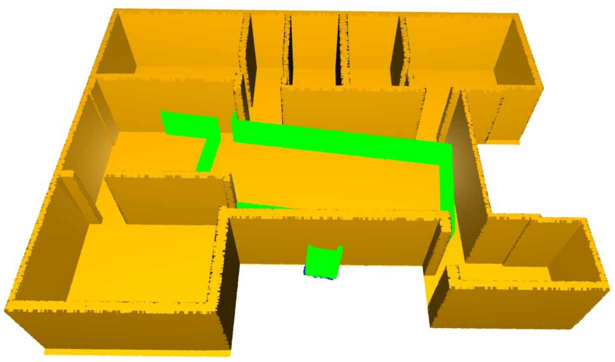


Figure 48. Point clouds with their floor planes aligned

As one can see in figure 48, the floor planes are aligned, however the walls still are not fully aligned, which will be done in the next step.

The result of this first step is a rotation matrix $R_{af} \in \mathbb{R}^{3\times3}$, which can be extended to a transformation matrix $T_{af} \in \mathbb{R}^{4\times4}$.

4.9.3 Align walls of both point clouds

After aligning the floor planes of both point clouds, the rotation about the Z axis might still be off. As the Manhattan world assumption shown in 2.4 still holds, I can simply compute the PCAs of both point clouds, and then align the X and Y principal vectors, as well as keeping the normal vectors from before to constrain the rotation.

As before, I use the `scipy` package to compute the rotation, and apply the resulting rotation matrix to the sampled point cloud. The result of this alignment step is now two point clouds, which are already aligned within the principal directions given by their floors and walls (figure 49).

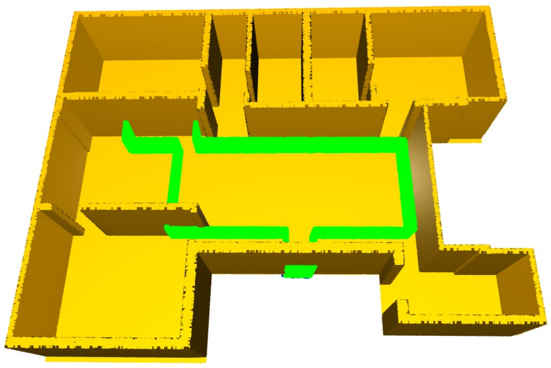


Figure 49. Point clouds with their floor planes and walls aligned

Multiplying the transformation matrix T_{af} from the previous step 4.9.2 with the transformation matrix $T_{aw} \in \mathbb{R}^{4\times4}$ (obtained by extending the rotation matrix from this step) then results in a transformation matrix $T_a \in \mathbb{R}^{4\times4}$. This matrix will be used later to compute the final transformation.

4.9.4 Initial scaling based on height

Now, with the point cloud walls aligned, the next step is to set an initial scaling based on the height of the point clouds. I use the method to compute the height of each of the two point clouds shown in 4.8.2. The height of the CAD model divided by the height of the sampled point cloud then results in the initial scale used for the following step.

Please also note that the height of the model will be fixed, as the height of the CAD model and the sampled model will need to be the same. This means for the next step, only two dimensions for scaling need to be considered instead of three dimensions.

4.9.5 Finding optimal transformation

Now finally, with the previous steps aligning the point cloud walls and computing the initial scale, the program moves to computing the optimal transformation.

The approach is rather simple, however it works reasonably well for finding a good transformation between the two point clouds. It consists of a few nested loops, which continuously change the transformation, apply it to the sampled point cloud and keep track which transformation yields the best fit:

```python
for rotation in rotations:
  for x_scale in x_scale_range:
    for y_scale in y_scale_range:
      for x_move_step in x_move_steps:
        for y_move_step in y_move_steps:
          # Compute transformation
          # Apply it to the point cloud
          # Compute fitness of current transformation
```

This might now seem inefficient, as the loops are nested relatively deeply, however in 4.9.5.1 I explain how the number of loop iterations (and as such the runtime) can be reduced by applying some sensible constraints.

This step of the program also features two live updated views 50, where one always shows the current alignment, while the other shows the optimal alignment which was found so far.

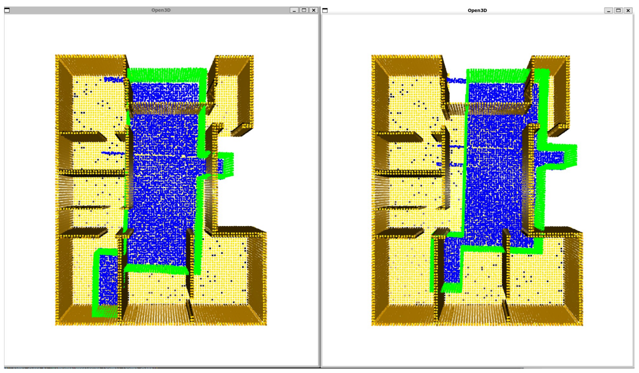


Figure 50. Live view (left) and current optimum view (right)

These views can be disabled, which speeds up the computation, however for debugging purposes, it might be useful to see what the algorithm is working on.

4.9.5.1 Constraints to reduce runtime

As mentioned before, the loop to compute the optimal transformation is deeply nested, which means the parameters to loop over need to be chosen very carefully, as not to increase the total number of loop iterations too much.

The following parameters all are used in the loops:

- `rotations`
- `x_scale_range`
- `y_scale_range`
- `x_move_steps`
- `y_move_steps`

The first parameter, `rotations`, is one where I can save many iterations. Usually, when aligning two elements in 3D space, one needs to consider all possible rotations, about all 3 axis (X, Y and Z). As the two point clouds are already aligned, with their floor planes matching alignments, and the walls being aligned with respect to each others principal components, this can be brought down to only needing 4 rotations: 0, 90, 180 and 270 degrees about the Z axis. Even though the point clouds are aligned, their respective principal components might not point into the same direction, and as such these 4 rotations are needed. This still brings the number of rotations down significantly, compared to the number of rotations needed if the point clouds were not aligned initially.

Next, the `x_scale_range` and `y_scale_range`: First off, scaling in z direction (i.e. the height of the point cloud) can be skipped, as the z scale is known from the height of both point clouds (as shown in 4.9.4. Second, the range where the x and y directions are scaled can also be constrained to a rather small range around the computed Z scale, as the scale of the CAD model might be slightly off, however it's assumed this imprecision is within a sensible range (i.e. if the scale is completely off, this approach does not work too well).

Also, if the current scale factors result in the sampled point cloud extending beyond the CAD point cloud, the move steps are skipped completely.

Lastly, the `x_move_steps` and `y_move_steps`: Again, moving in Z direction is not needed, as the floors are already aligned, and the height is set. For the steps in X and Y direction, a sensible number can be chosen, during testing 30 steps have been shown to work well.

With all these constraints, the number of loop iterations can be brought down significantly, which allows the algorithm to run all these steps to find an optimal fit in a sensible amount of time.

4.9.5.2 Keeping track of best result

To find the transformation (i.e. combination of rotation, scale and translation) which fits the sampled point cloud the best into the CAD model point cloud, some metric is required that enables comparing these registrations.

Open3D contains a method called `evaluate_registration`, which makes it possible to compute an evaluation of a registration, given two point clouds and a transformation. The results contains multiple fields, from which I use the `fitness`, which is the fraction of overlapping area (i.e. number of inlier correspondences / number of points). A higher number means a better registration, with a value of 1 being a perfect match (e.g. when two copies of the same point cloud are aligned perfectly with each other).

As the transformation in each step is applied to a copy of the sampled point cloud (because the transformation is built as a sequence of rotations, translations and scaling operations), the transformation passed in is the identity transformation.

For each computed transformation in the innermost loop body, a score is returned from this method. Getting the best registration is now simply a task of keeping track of the highest score and its corresponding transformation, updating both if a transformation with a better fitness is found.

After the loop terminates, the program returns a final best transformation, expressed as number of rotation steps, X and Y scale factors as well as the number of translation steps for both directions. This data now needs to be turned into a final transformation matrix.

4.9.6 Computing transformation matrix

Finally with the transformation matrix T_a from 4.9.3 for aligning the two point clouds as well as the data from the previous step, the final transformation can now be computed.

With the data from step 4.9.5.2, containing the number of 90 degree rotations, the translations in X and Y direction as well as the X and Y scale, I can now build individual matrices for this rotation, scaling & translation. Finding the final matrix for the global registration is then simply a matter of multiplying these matrices, which results in the final transformation matrix $T_g \in \mathbb{R}^{4\times4}$. When this matrix is applied to the (untransformed) sampled matrix that was passed in as an input, it will align it according to the best found alignment as shown in figure 51.

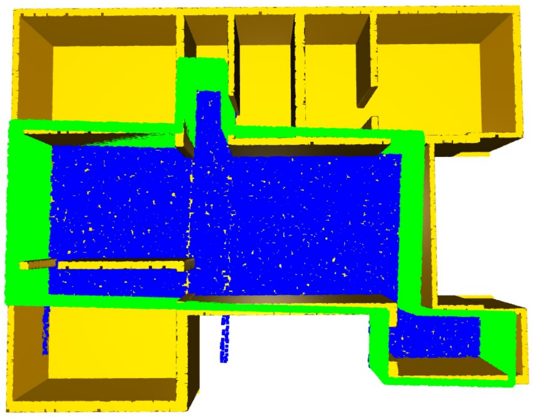


Figure 51. Alignment after computing the final transformation

4.9.7 Applying final transformation to CAD model

In the section before, the transformation steps (rotation, scale & translations) were always applied to the sampled point cloud. As the sampled point cloud is an approximation of the captured point cloud, which is the data from the real world with the correct scale, it's clear that the transformation should be applied to the CAD model point cloud, not the captured point cloud. This however is rather easy, as the transformation T_g can just be inverted as $T_{glob} = T_g^{-1}$, and then be applied to the CAD model point cloud, resulting in the correct transformation.

4.10. Refining alignment locally

The approach explained in the previous section yields a good global registration, however on the local scale, the registration still could be refined.

For this step, ICP 2.2 can be used, as ICP works well if the initial alignment is already rather good.

As a preparation, the previously computed transformation T_{glob} is applied to the CAD model point cloud, such that it and the sampled point cloud already fit into each other rather well.

Open3D features various ICP algorithms, from which I simply use the `PointToPoint` ICP algorithm. Again, I use the previously mentioned `evaluate_registration` method from Open3D to compute the fitness for a registration, which allows me to keep track of the best registration from a set of possible registrations.

As the scale still might be a little bit off, this step also again incorporates optimizing the scale, however this time in a much smaller range (i.e. $\pm5\%$ in X and Y directions each).

For each step in this XY loop, first the scaling is applied to the CAD model point cloud, and then ICP is executed on the now scaled point cloud and the captured point cloud. This yields a new registration, where the fitness might be better than in previous steps.

After running all iterations of the scale factors, the final transformation for local registration is computed as $T_{local} = T_{icp} \cdot T_{scale}$. As one can see in figure 52, this does result in a better alignment compared to the result of the global registration.

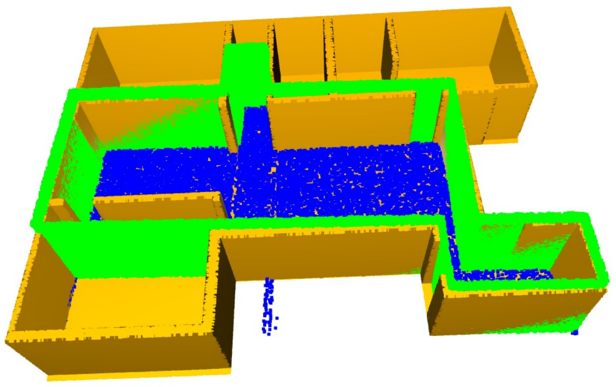


Figure 52. Alignment after running local optimization

The transformation matrix from this step can now be multiplied with the transformation from the global registration step, which results in the final transformation matrix $T_{final} = T_{local} \cdot T_{glob}$.

4.11. Merging point clouds

With the final transformation T_{final} for the CAD model point cloud aligning the two point clouds now rather well, the next step is to merge the two point clouds into a single one, which then can be used further.

For this, first the CAD model point cloud is aligned to the captured point cloud by applying the transformation T_{final} to it.

I then build a KDTree [5] on the transformed CAD model point cloud, and compute the average distance between each point and its 5 nearest neighbours by using this KDTree.

After this, I iterate over each point from the captured point cloud and use the position of the point as the input for the query on the KDTree, which returns the number of neighbouring points as well as their indices in the KDTree within a certain radius (which is a fraction of the previously computed average distance).

If this number is larger than zero (i.e. there are points close to the query point), the indices of the points inside this radius are added to a list l_{del}.

[5] http://www.open3d.org/docs/release/python_api/open3d.geometry.KDTreeFlann.html

After iterating over each point in the captured point cloud, the list l_{del} now contains the indices of all points from the CAD model point cloud that need to be removed. Removing these points is simply deleting the corresponding rows from the matrix of 3D points making up the point cloud, which results in a point cloud with holes in it where data of the captured point cloud can replace this data, as shown in figure 53.

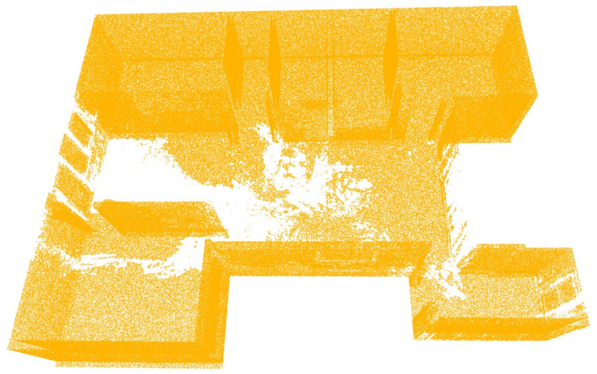


Figure 53. CAD model point cloud, points near captured data removed

Finally, the points and colors from the captured point cloud can then be added to the respective matrices of points and colors, resulting in the final result shown in 4.12.

4.12. Final result

After running all steps, the program outputs the final transformation T_{final}, as well as the merged point cloud, containing all points from the captured point cloud, as well as the kept points from the CAD model point cloud. This new point cloud (figure 54) can now again be used to fit another point cloud in it or in some other programs.

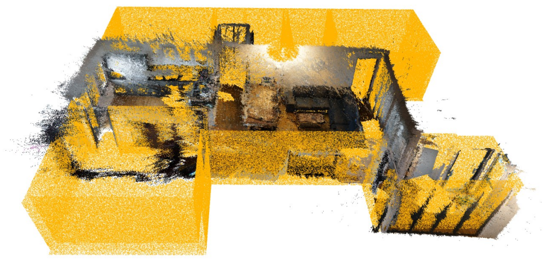


Figure 54. Output of the program, view from outside

Especially when looking from below (figure 55), one can also see that the captured data does replace parts of the CAD model.

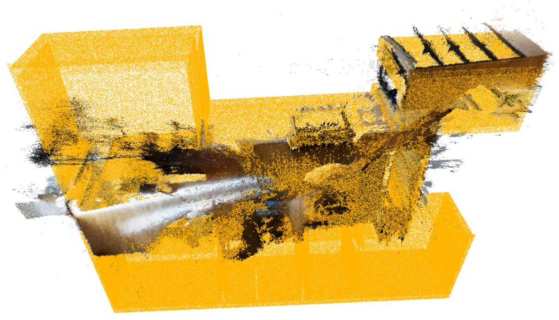


Figure 55. Output of the program, view from below

When zooming in, the added details from the captured point cloud become visible, with the CAD model point cloud being augmented with the data from the input images (figure 56).

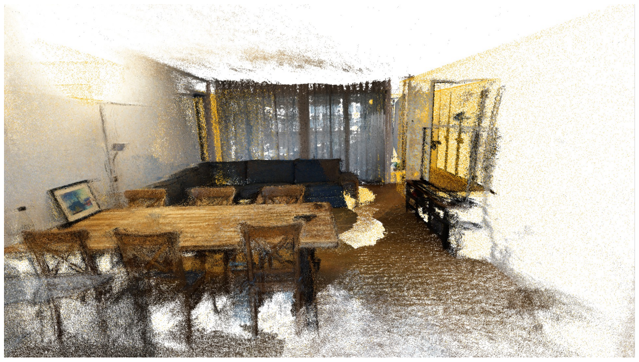


Figure 56. Output of the program, view from inside

4.13. Other evaluated approaches

In this section, I will show some of the other evaluated approaches which I think are worth showing and also elaborating on why I chose to not use that approach.

Some parts of these approaches eventually did find their way into the final method as part of the whole process, e.g. the 4.13.6 approach, while others were completely discarded.

The subtopics presented here are ordered by the time I evaluated them in the thesis, with some topics taking more time than others.

4.13.1 Using SOLD2 to reconstruct walls

One of the first approaches I tried was using the algorithm presented in the SOLD2 paper by Pautrat et al [9]. As the algorithms finds lines in images, I wanted to evaluate whether these lines can be used to match pictures of the real world directly to their virtual counter part, without the need to create a 3D model from the images. Later on, I also tried projecting the lines from the 2D images into the 3D point cloud, to see whether these lines can be used to find important structural elements such as corners and the intersections between the floor and walls.

4.13.1.1 Matching images with virtual counterpart

The first approach using SOLD2 was to take a single image from the captured sequence of pictures, and create an image from the same view port in the 3D cad model. To simplify this task, as I mainly wanted to see whether the lines found in both images are similar and can be matched, the camera in the CAD model was placed manually.

With these images, I ran SOLD2 on both of these images, which tried to match the found lines in both images.

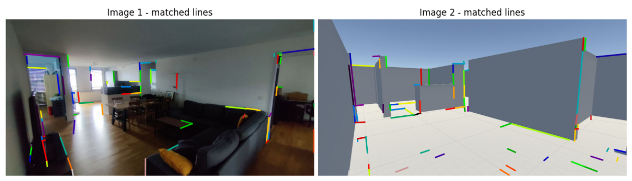


Figure 57. Output of the line matching step

As one can see in the figure 57, some of the matched lines do correspond to the same line in the real and virtual world, however most of the lines are off, and therefore SOLD2 also could not compute a homography between the two images. Therefore, this approach was not evaluated further.

4.13.1.2 Projecting lines into 3D model

After some time working on other approaches, the idea to use the SOLD2 lines and map them into 3D space came up. Here, I wanted to use these lines to find structural elements in the 3D point cloud.

One issue that became apparent during working on this approach was that the lines from the SOLD2 output don't feature any depth information, as the program is run on a 2D image. This means mapping a segment of a line to a 3D point is rather difficult, especially if the structure that lies along this line (e.g. a corner) is not completely present in the 3D point cloud. I then tried running the plane segmentation algorithm (using the RANSAC approach shown in 4.6.1), and then computing the intersection of the lines projected into 3D space and the planes.

As one can see in figure 58, this did work for some lines, while other lines were mapped to a wrong surface, e.g. the floor.

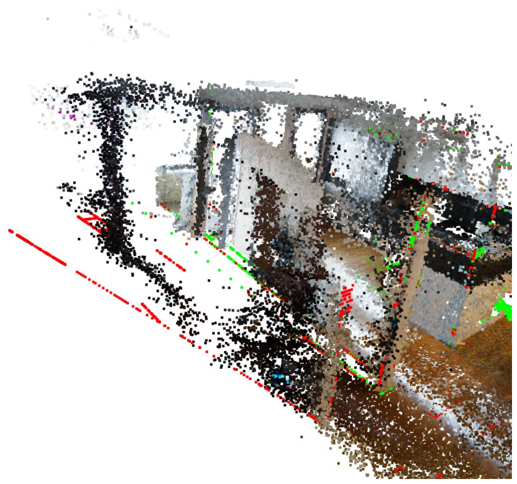


Figure 58. Mapping the SOLD2 lines from a single image to the 3D point cloud

Another idea was to run the SOLD2 program on each pair of consecutive images, only projecting the corresponding lines between two images. This way, each line would have 2 representations from different views, and as such, the depth could be computed for such a line.

The SOLD2 program did find many matched lines between the images, but these could not be used yet, as they might have different lengths, as shown in figure 59.

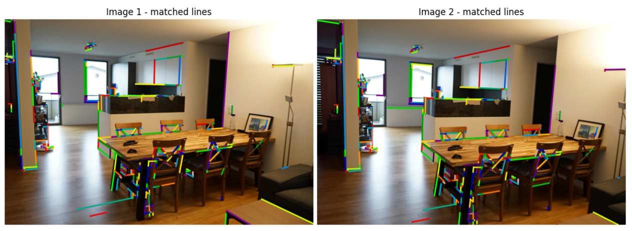


Figure 59. Matched lines between two consecutive images

However, the lines with correspondence did not result in many lines, and as such this approach was also discarded (figure 60).

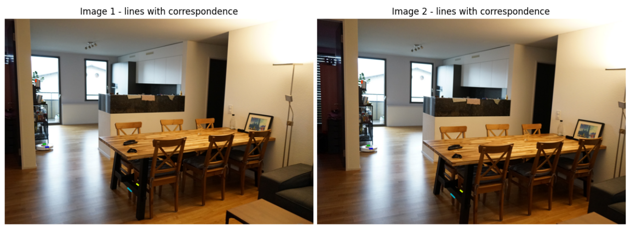


Image 1 - lines with correspondence
Image 2 - lines with correspondence

Figure 60. Lines with correspondence between two consecutive images

4.13.2 Predator registration

At the start of the thesis, I also looked into some more recent registration methods, especially the PREDATOR registration framework by Huang et al [5], which looked to be promising.

The examples for the framework did work really well on my computer, and therefore I wanted to see what the result of trying to register the captured, unprocessed point cloud with the CAD model point cloud would be. To run the framework, I down sampled the point clouds, such that they need less memory, and modified the demo code in a way such that it also accepts `.ply` point cloud files [6].

However, the program kept crashing, as it tried to allocate more memory than was available on the GPU. As the file sizes of the two point clouds were rather large, I ran the program again with versions of the point clouds that were down sampled even more (from approx. $600'000$ points to $10'000$ points), however the program still crashed.

I then tried running the program in CPU only mode, but again with the same result, the program wanted to allocate 120GB of memory, where as my computer only has 64GB of memory.

I found an issue in the repo of the code [7], where the author of the code states that some parts of their network require a lot of memory and that they used GPUs with at least 12GB of memory.

With this in mind, I decided to stop evaluating this approach, as the hardware requirements of the PREDATOR program are just too high for usage on normal PCs. Even many higher-end computers would struggle to satisfy the hardware requirements, and an average computer currently has no where near the required hardware.

4.13.3 Capturing point cloud with Kinect

After trying SfM to capture a point cloud as shown in 4.3, I wanted to see whether using a camera with a depth sensor results in better, denser point clouds. For this, I used a Xbox 360 Kinect sensor, as it features a RGB-D camera, is inexpensive and easily available.

With the Kinect camera, I then used several methods to capture a point cloud. The first approach was to use the *Kinect Fusion Explorer* app, which is based on the paper KinectFusion: Real-Time Dense Surface Mapping and Tracking by Newcombe et al [8], however this app was only able to generate a mesh from a static view of the Kinect sensor. The field of view of the Kinect is rather small, and the depth perception is limited to a few meters, and as such the resulting mesh was rather small, as shown in figure 61.

[6] `https://github.com/prs-eth/OverlapPredator/issues/27`
[7] `https://github.com/prs-eth/OverlapPredator/issues/2#issuecomment-766168105`

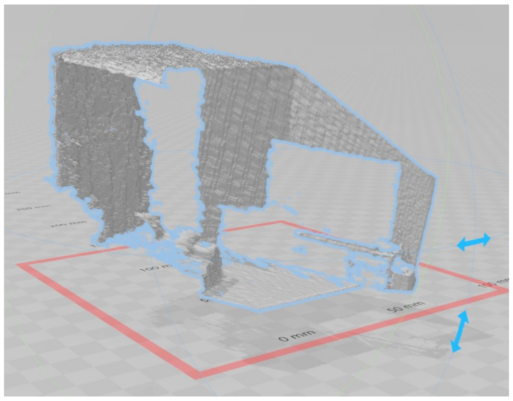


Figure 61. Mesh generated with Kinect camera, standing on the table, looking towards wall with TV and entry hallway

Interestingly enough, the Kinect failed to capture any data at the area where the TV was standing, this is probably due to the TV being a reflective surface.

I also used the program *Skanect*, which allows the user to walk around with the Kinect and scan an object or room and then re-construct a model from that scan. However, the software performed quite poor, and even in my room, which is about 3×4 meters, it kept losing the tracking, which messed up the alignment of the surfaces. This especially was the case when the Kinect was pointed towards a segment of the room where there isn't a lot of structural information, e.g. a wall. Because of the small field of view (which is significantly smaller than the field of view of my smartphone or Sony camera), it was relatively difficult to always keep enough structure in the view for the tracking to work.

As this approach did not result in better point clouds, I decided to keep using only RGB images as input, and work with the point clouds I have from OpenSfM.

4.13.4 Semantic segmentation

As previously elaborated, the main issue with the captured point cloud is that it contains many points which do not belong to the structure of the room (see figure 5 for an example of this categorization). The idea to use semantic segmentation came up, i.e. to use a pretrained neural network to label the point clouds by labeling the individual images and then projecting the labels into the 3D point cloud, keeping only the points with relevant labels.

4.13.4.1 Labeling individual images

The first step in this approach is to label the individual images. For this task, the network *lightweight refinenet* by Nekrasov et al [7] was chosen. For this network, multiple pretrained models are available, which simplifies the task of labeling the images. After setting everything up, the sequence of images was labeled with each available pretrained network, with the labels being exported as RGB images, as well as numpy matrices, such that it's later possible to easily access a label of a pixel.

One type of pretrained model assigned every pixel a label, for example the model that was pretrained on the NYU dataset [6]. An example of this model labeling images can be seen in figure 62.

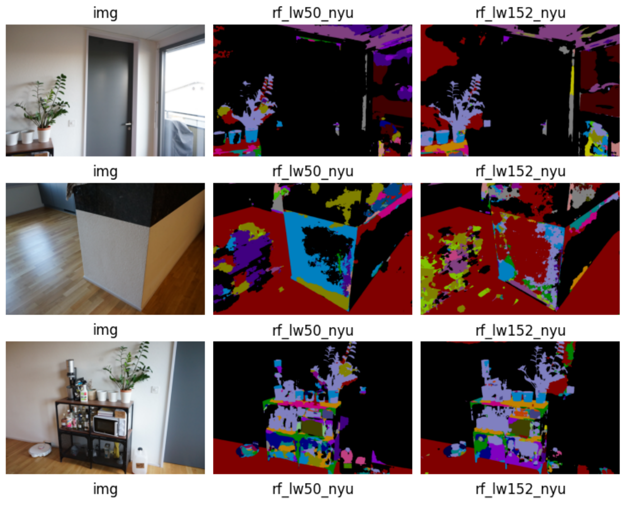


Figure 62. Example output of the model pretrained on the NYU dataset

The other type, which contained a model trained on the VOC dataset[8] assigned objects (such as furniture) labels, while assigning the zero label to everything else (i.e. walls, floors, etc). When the output is overlaid onto the RGB input image (figure 63), one can see that the labels do capture most of the furniture, however not all the pixels corresponding to furniture are labeled correctly.

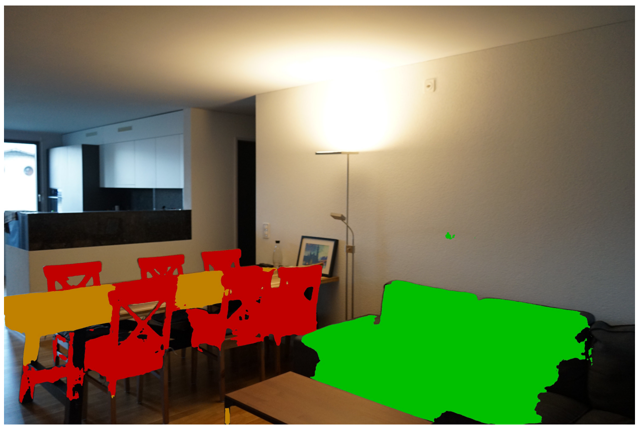


Figure 63. Example output of the model pretrained on the VOC dataset

[8] http://host.robots.ox.ac.uk/pascal/VOC/

4.13.4.2 Mapping labels to 3D points

The next step was to map the labels from the 2D images into the 3D space. The OpenSfM library also contains a python package, which makes it possible to access the data generated by OpenSfM. With this, it's possible to compute the 2D coordinate xy of a 3D point in the point cloud for a given image (or rather camera position from which the image was taken). This means it's possible to compute all the 3D points which, if projected onto a 2D plane, could be visible from the camera perspective. These points then can be mapped to the corresponding 2D image, as seen in figure 64.

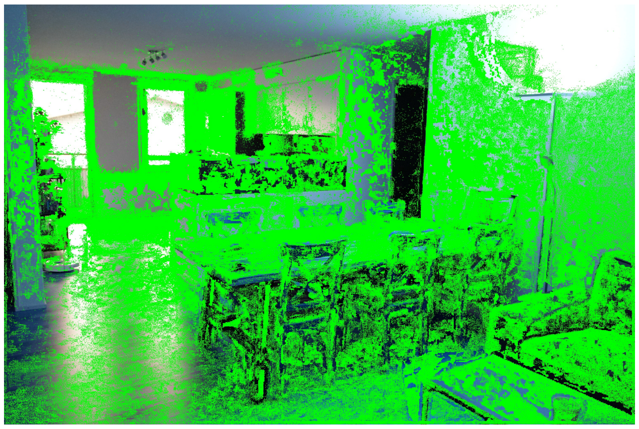


Figure 64. Mapping the 3D points to 2D

With this, I can now iterate over the images, compute for each visible point for this image it's image coordinates, and then fetch the label for this pixel from the data that was created in the previous step. An example of this is visible in figure 65.

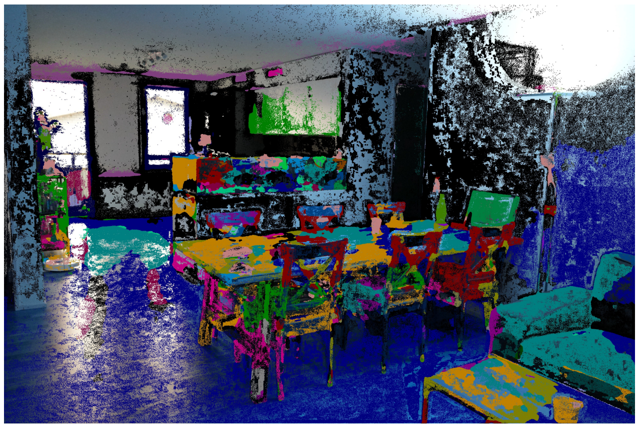


Figure 65. Labels mapped to the points

The program computes the labels of points for each image, returning a map of point identifiers to a label for each image. This allows the program to process the images in parallel, speeding up the process significantly.

After these maps have been computed for all the images, the program merges the maps, counting the occurrences of each label for each 3D point (as the labels for a point might be different from varying perspectives). After this, the label with the highest number of occurrences for each 3D point is kept as the definite label.

4.13.4.3 Using depth data to cull points

One issue with the previous approach is that *all* points which could be visible from the camera perspective for an image are fetched. Points outside the view frustum (i.e. also behind the camera) are removed, however every point in front of the camera is a valid point with this approach. This also means that points behind obstacles, such as walls or furniture, are still in the image.

If the captured point cloud is rather simple and mainly contains a single room, this is not an issue, as there aren't many points occluded by other surfaces. When the point cloud contains multiple rooms or turns (e.g. from walking down a hallway), this can become a bigger issue, as visible in figure 66. Many of the mapped points, mainly in the bottom left, should not be visible from this view, as they are hidden by structures in front of them (e.g. the wall on the left side or the pillars).

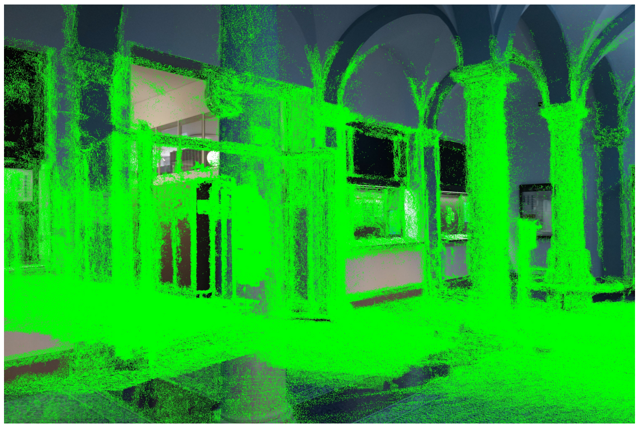


Figure 66. Points visible which should be hidden

By mapping pixel labels to these points, many points now receive incorrect labels from surfaces in front of them, which decreases the accuracy of labeling the 3D points.

Simply filtering out points laying behind another point does not yield satisfactory results, since there are many spots where there are no points in front of the occluded points (e.g. if OpenSfM was not able to track a part of a wall).

I then tried using *GluonCV*[9], which features a neural net-

[9] https://cv.gluon.ai/

work to compute depth in images. Computing the depth of the images worked rather well for the images, for example when inputting image 67 in the network, the output was the data visualized in figure 68.

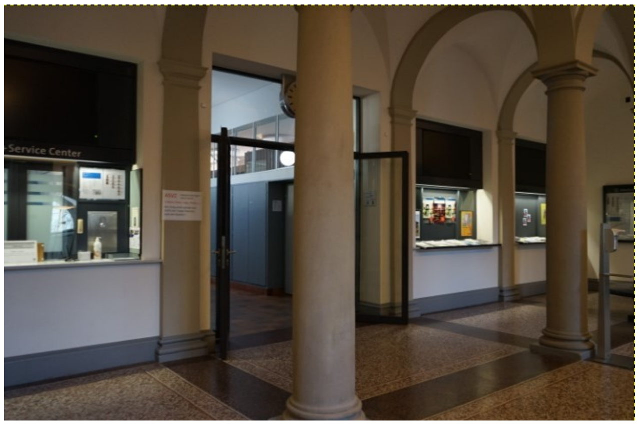


Figure 67. Input view of GluonCV

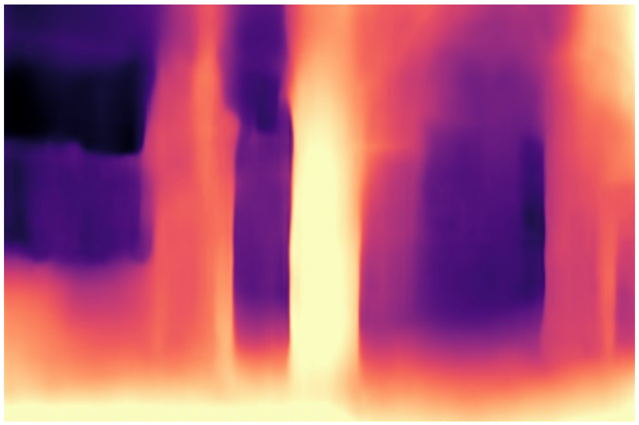


Figure 68. Output of GluonCV visualized

While this output managed to compute the depth differences well, the issue is that the depth output from GluonCV is $x \in [0, 1]$ for each pixel, i.e. it only outputs relative depth. Directly filtering with this data is not possible, as the depth of the points in the image range from 0 to an unknown depth.

After using GluonCV, I also found some auxiliary data generated by OpenSfM, which contains the depth data for the images as well. Here, the depth of the points was in a larger range, specifying the depth of the point in relation to the camera position, visualized in figure 69.

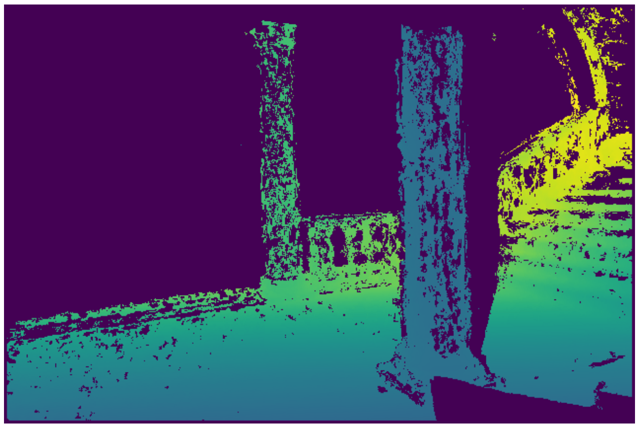


Figure 69. Depth data from OpenSfM

The only issue with the depth data from OpenSfM is that it only tracks depth until a certain depth is reached, points further than this have their depth set to zero. This is visible in figure 70, where points with $depth = 0$ are highlighted in yellow. The green points are the points that are not obstructed by anything in front, while red points are the points hidden by other surfaces in front of them.

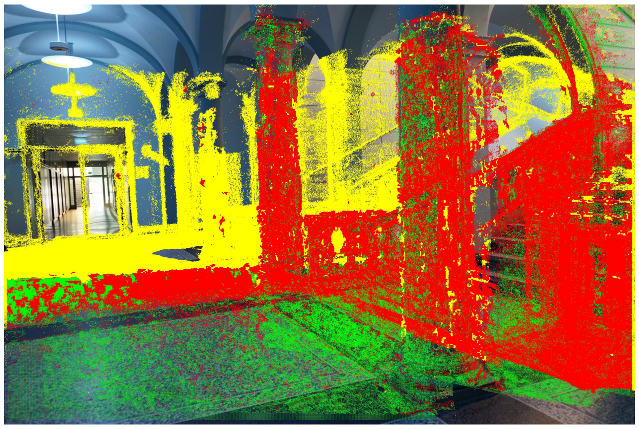


Figure 70. Points filtered with OpenSfM depth data

As one can see, using the depth data from OpenSfM works well, and indeed, many hidden points are removed by this. For experimenting, I added two modes, one where all points with $depth = 0$ are discarded (visualized in figure 71), and another where all points with $depth = 0$ are kept (visualized in figure 72).

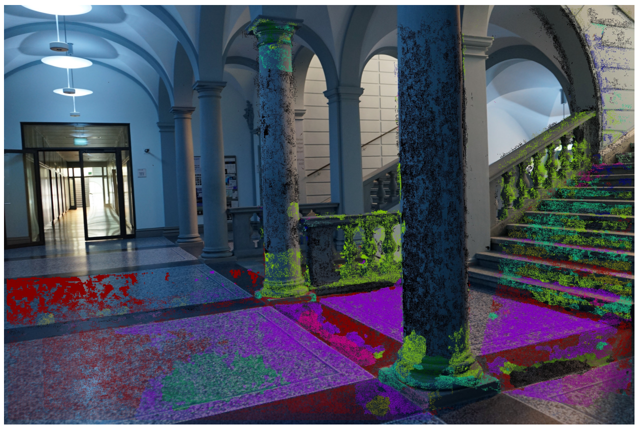


Figure 71. Labels mapped to points, ignoring points with $depth = 0$

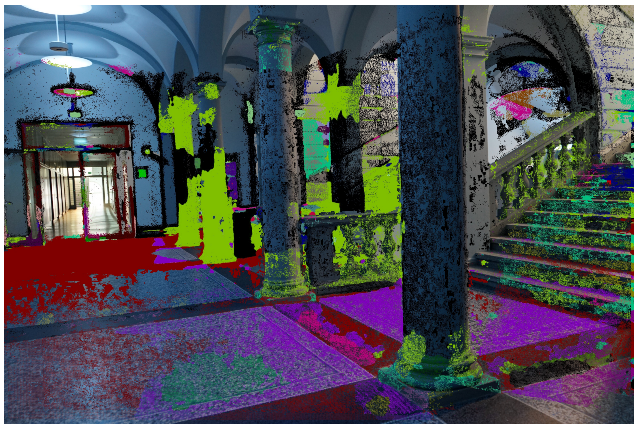


Figure 72. Labels mapped to points, keeping points with $depth = 0$

As the mode where all points with $depth = 0$ results in labels being mapped to 3D points which might be incorrect, I decided to discard the points with $depth = 0$, which results in fewer points being labeled in each image, but there are fewer incorrectly labeled points.

Also note, that this issue with points having no depth mainly occurs in views with a large depth difference, if the furthest points visible are only a few meters from the camera, this does not happen and almost all pixels in the image do have a depth assigned.

4.13.4.4 Result of semantic segmentation

With the depth data now being used to only map applicable labels to points, I can map the labels of every image and then create a labeled point cloud from the data, as explained in 4.13.4.2, with the final result of this visualized in figure 73.

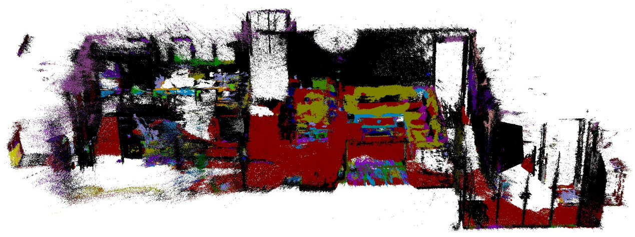


Figure 73. Labeled point cloud

As one can see in this image, most floor and wall points have been labeled correctly, however a lot of the furniture contains wrongly labeled points. If I only keep the points with relevant labels (i.e. wall, floor, ceiling etc.), one can see that there are still many points belonging to furniture and other object left in the point cloud (figure 74).

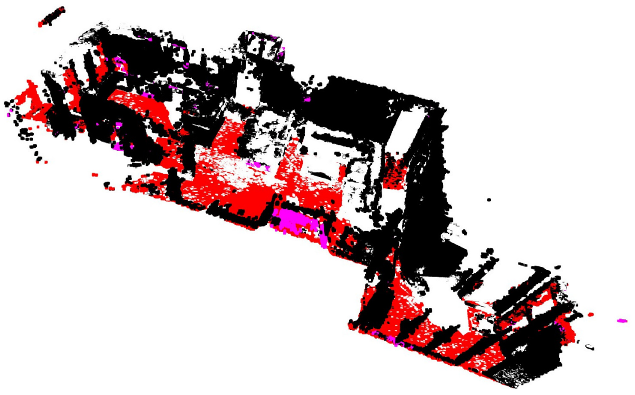


Figure 74. Labeled point cloud, keeping only relevant labels

Going back to the labeled images, it becomes clear that the refinenet does achieve a high success rate with labeling, however there are still too many incorrectly labeled pixels, which then also has an effect on the quality of the labels in the point cloud.

After spending a considerable time on the approach using semantic segmentation, I decided to try another approach (which was the RANSAC approach explained in 4.6.1).

4.13.5 Using a histogram to detect planes

Another approach was to create a histogram of the points, i.e. flattening all points onto a single plane and then visualizing the density of points. The idea was that regions where walls are to be found contain a higher density of points, which allows to find lines in this 2D representation which lie along these walls.

One issue however is that not only the the walls have high densities of points, but also other objects (such as furniture) might have a high density of points. Shown in figure 75, the general layout of the room can be seen, however some walls are barely visible (due to them having fewer points than other areas in the point cloud). Also, some parts of the furniture, e.g. the table, are also visible, as these regions have a high density of points.

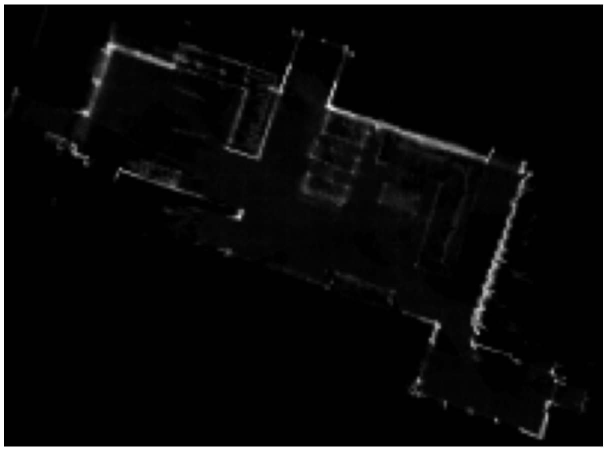


Figure 75. The computed histogram, using a suitable neighbourhood size to compute the value for a pixel

I tried detecting lines in this image with OpenCV[10] using edge detection with the Canny edge detector, followed by `HoughLinesP`. This did indeed detect some of the lines that make up the walls of the point cloud, but also some lines belonging to other surfaces (for example the table), as seen in figure 76. Lowering the threshold for detecting lines added too many false positives, while increasing it removed many of the desired lines along walls.

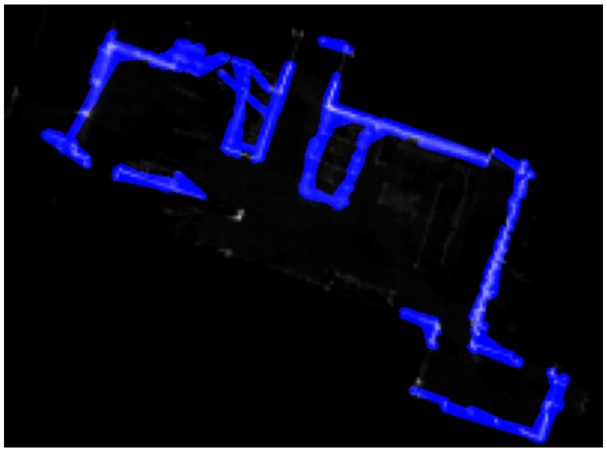


Figure 76. Lines found on the previously computed histogram

[10]https://opencv.org/

I therefore chose to stop following this approach, as it appeared to be too dependent on the structure of the data (i.e. it might yield usable results for point clouds mainly containing the walls, and unusable results for point clouds having many other objects within).

4.13.6 Constrained RANSAC

This approach which I call *constrained RANSAC* was already mentioned in 4.6.2.3, using it to find planes in point clouds where I already know the orientation of the plane.

Usually, in normal RANSAC, the n points to fit the plane trough are picked at random from all points in the data set. The RANSAC algorithm then fits a plane through these n points and computes the number of inliers (i.e. points which are within a certain distance to the plane). This process is repeated a number of times, keeping track of the plane which results in the highest number of inliers.

This usually works really well, however one downside of this algorithm is that it can find an arbitrarily oriented plane in the data (which might be the best fitting plane and not only resulting from a low number of sampling steps). For this application, this is not optimal, as I'm only interested in the floor plane and wall planes.

When considering the Manhattan world assumption 2.4, it can be seen that the planes (or rather their normal vector) forming relevant surfaces must align with one of the three main axis of the point cloud (X, Y or Z).

These directions can be computed using the approach shown in 4.6.1.4, and with these it's now possible to use this *constrained RANSAC* approach. In addition to the number of iterations as well as the point cloud itself, this method also takes the direction the normal vector of the plane must face.

The algorithm then picks a single point (instead of n points), and constructs the plane equation from the point and the normal:

```
a, b, c = plane_normal
x0, y0, z0 = random_point
d = -(a*x0 + b*y0 + c*z0)
plane = [a, b, c, d]
```

With this plane, I apply the normal logic for RANSAC: Compute the number of inliers, and compare if this plane has a larger number of inliers compared to the previous iteration. This approach forces all planes to be parallel to each other (following the direction given by the principal component for this direction), only moving the planes along their normal vector.

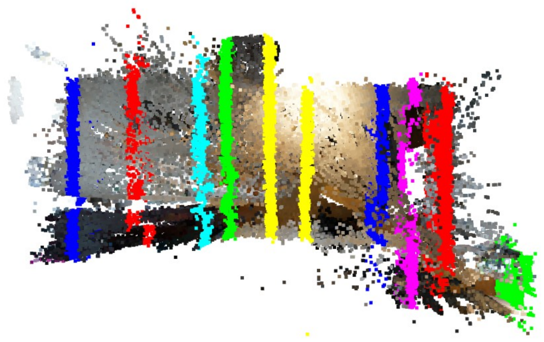


Figure 77. 10 planes found by the constrained RANSAC approach

In figure 77, I visualized this algorithm being run 10 times for a certain direction. Many of these planes are relevant, however the algorithm also fits some planes through the middle of the point cloud. The main issue with this approach is that the number of times the constrained RANSAC approach is applied to the point cloud determines the number of planes found. For example, if the point cloud contains 2 planes in X direction, and 10 planes in Y direction, the user would need to specify this, as running both directions 10 times would result in 20 planes. Running the normal RANSAC 20 times (with the PCA alignment check explained), the algorithm can find up to 20 planes, with invalid planes being discarded. This means the chance of finding only the really needed planes is higher with the normal RANSAC approach.

Also, even if the number of planes to be found is known, the algorithm might not find these planes, especially if these planes are rather small, as it might find a plane with a higher number of inliers elsewhere in the point cloud.

I therefore decided not to use this approach alone, and only use the method in 4.6.2.3 as a part of the method of finding boundary planes.

5. Experiments

5.1. Comparison with ICP without previous global registration

The first experiment is comparing the result from my method with a registration using plain ICP.

For the ICP, I use two implementations of the algorithm: The version in Open3D, which I also use for the local optimization step 4.10, as well as the version implemented in the software CloudCompare [11].

5.1.1 ICP in CloudCompare

The version of ICP in CloudCompare offers a version of ICP where the scale of the point cloud to be registered can be changed to fit the reference. As such, I first try the version where the scale can be changed as well, as the scales of the two input point clouds (captured point cloud and CAD model point cloud) might not match each other. In figure 78, one can see the input of this test.

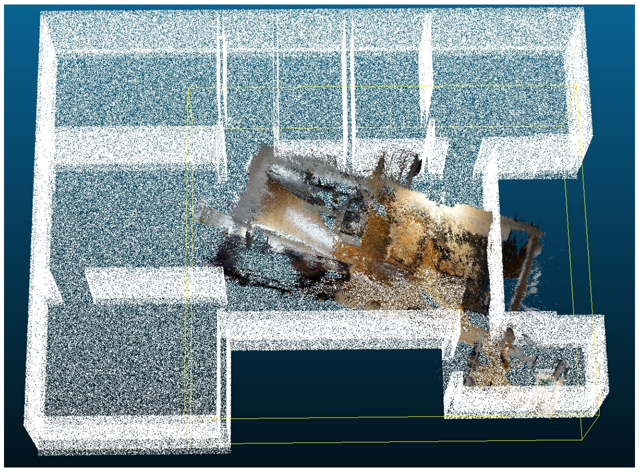


Figure 78. Input for ICP with scaling enabled

After a few seconds, the algorithm is done, however the algorithm scaled the captured point cloud down, as this probably yielded the best fitness for these two input point clouds. In figure 79, the captured point cloud is visible as a very small collection of colored dots in the right side of the image.

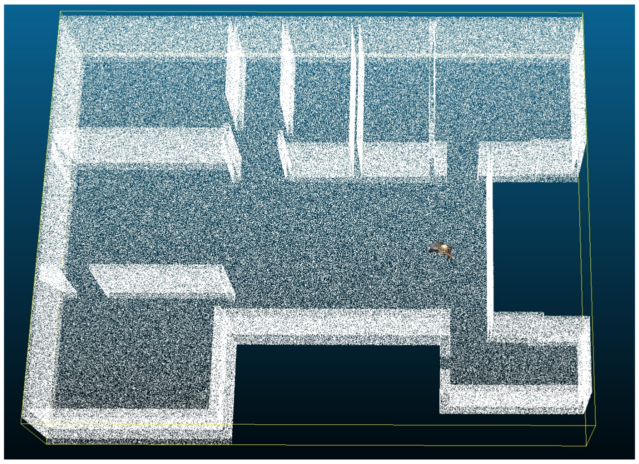


Figure 79. Result for ICP with scaling enabled

With multiple runs of this version, the result was always similar, i.e. the captured point cloud was scaled down too much. I therefore tried the version without scaling. The main issue here is that the point clouds both need to have the correct scale already, i.e. any difference in scale might already negatively impact the result. Therefore, I manu-

[11] https://www.danielgm.net/cc/

ally scaled the captured point cloud such that it matches the scale of the CAD model point cloud, as visible in figure 80.

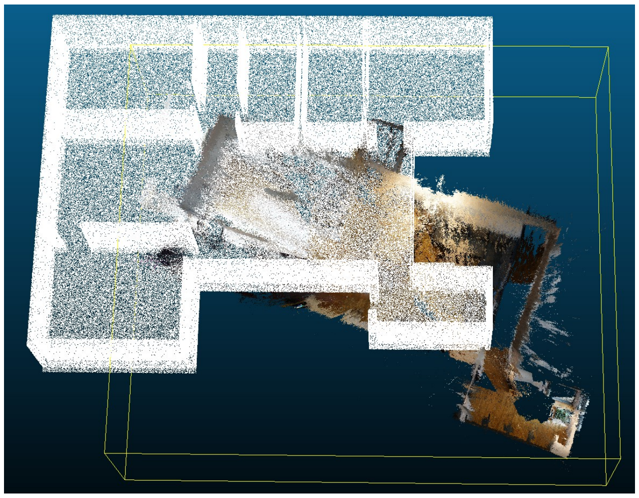


Figure 80. Input for ICP with scaling disabled

Running this version seemed to yield a good result, as one can see in 81, however, when viewed from the side 82, the result still is not optimal. This is probably due to the points corresponding to furniture and other objects present in the captured point cloud negatively impacting the result for ICP.

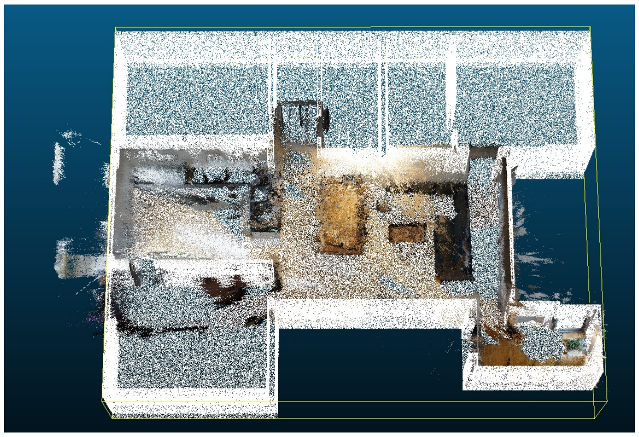


Figure 81. Result for ICP with scaling disabled, top view

And of course, this result was only possible because I manually scaled the point cloud before running ICP, as this version does not change the scale of the point clouds. Running ICP without either changing the scale or have the ICP algorithm change the scale is pointless anyway, as there will be no useful result which could be used to merge the two point clouds into each other.

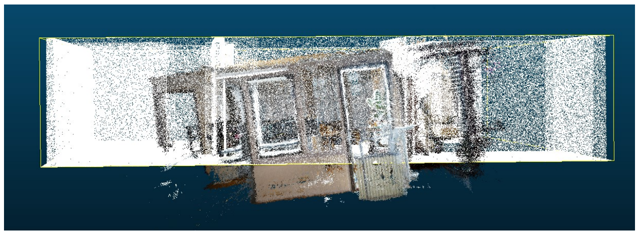


Figure 82. Result for ICP with scaling disabled, side view

Lastly, I ran the ICP of CloudCompare on an input which is similarly unaligned as the input for the global alignment step 4.9. Here, I chose to use the sampled point cloud, with the same alignment as the one in 47 as the input, to compare the two approaches on the same input data. The only difference here is that I scaled the input for the CloudCompare ICP, as it does not optimize the scale of the point clouds.

In figure 83, one can see the input of this, and in figure 84 the output. While this seems pretty well aligned, one can also see that the alignment is still off, especially in the rotation about the Z axis. This is mainly due to the fact that ICP may rotate the point cloud in any arbitrary direction to optimize its goal, while the global registration step in my method first aligns the floor and wall planes, and then is constrained to 4 rotations about the Z axis.

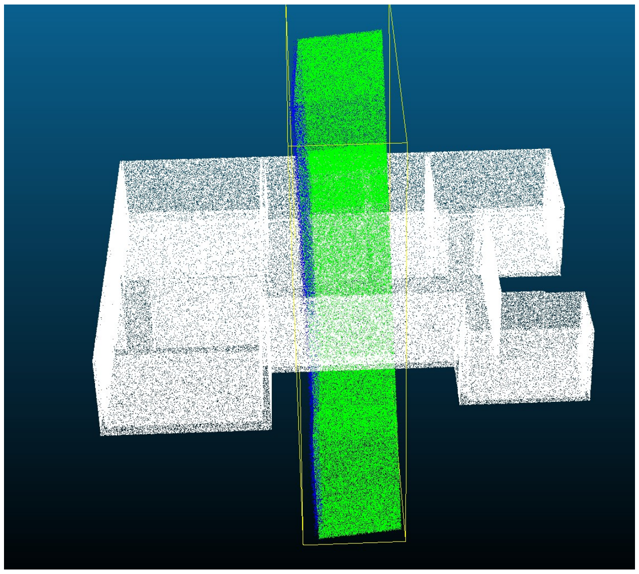


Figure 83. Input for CloudCompare ICP with unaligned point clouds

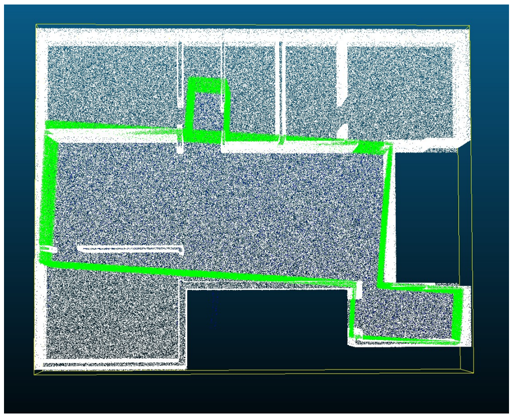


Figure 84. Result for ICP with unaligned point clouds

When comparing the fitness score of this result, which is 0.39717 with the fitness score of the best fit found by my method, which is in the region of 0.49 (see 5.3) it's clear that my method yields a better result for the same input. Also, one thing to keep in mind is that my method works on point clouds which have different scales, which is not the case for ICP.

5.1.2 ICP in Open3d

Open3D also features an ICP implementation, which I do use in the local optimization step explained in 4.10. However, this version also does not change the scale of the inputs, and therefore would not work if the inputs have a different scale.

I therefore again need to set the scale manually, and run the ICP algorithm on this scaled input, shown in 85. To get a sensible result, I had to use a rather large threshold for the ICP method of Open3D, as using only a small threshold did not result in sensible results. The threshold parameter is the maximum correspondence distance, i.e. the radius around each point the algorithm will try to find a corresponding point.

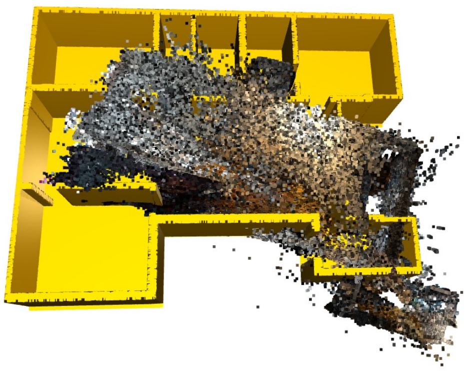


Figure 85. Input for ICP in Open3d

With a large enough threshold, the algorithm did manage to find a transformation which is not the identity transformation, however the resulting transformation still did not align the point clouds well enough, as seen in 86. Increasing the threshold also increased the runtime of the algorithm significantly.

In contrast, in the local refinement in 4.10, I'm able to use a small radius, as the point clouds are already roughly aligned, which keeps the runtime in a sensible time frame.

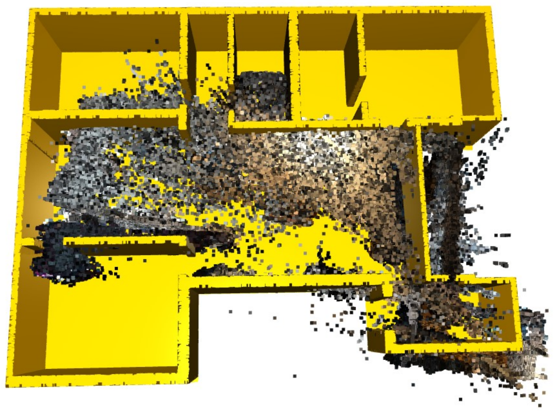


Figure 86. Output for ICP in Open3d

I also ran ICP of Open3d against the sampled point cloud, as I did previously for the ICP implementation of CloudCompare, with the input as shown in 87.

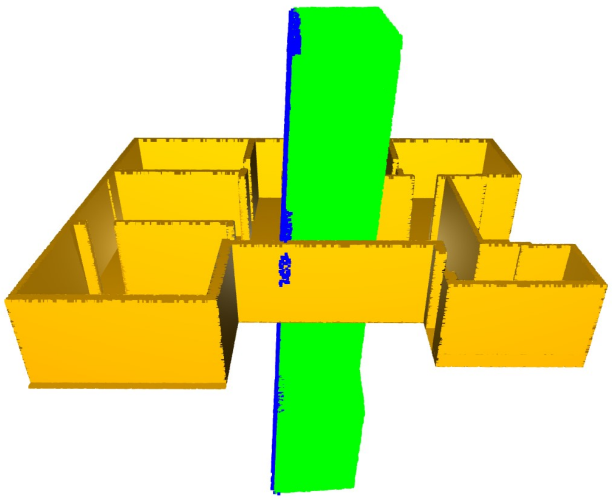


Figure 87. Input for ICP in Open3d, with unaligned point clouds

In 88, I visualized the outputs with increasing threshold sizes, however, none of these thresholds resulted in a correctly rotated point cloud.

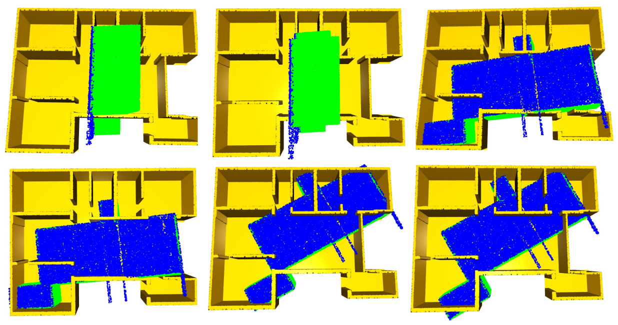


Figure 88. Output for ICP in Open3d, with various threshold sizes (0.5, 1, 5, 10, 15 and 25)

5.1.3 Results

While the ICP with the correctly scaled, sampled point cloud in CloudCompare results in a good result, it still has a significant lower fitness score than the result from my method, as the rotation about the Z axis is off. The implementation in Open3D did not manage to return an usable result at all.

Furthermore, both methods require two point clouds as input that already have a correct scale, where my method also can work with differently scaled methods.

5.2. Comparison with global registration from Open3D

Open3D also features methods for global registration [12], which I want to compare to my method. As the method for Open3D features two main parameters which can be changed, `threshold` (again referring to the distance radius for correspondence search) and `voxel_size`, I run the algorithm on a variety of combinations for these two inputs.

The algorithm is ran on the same input I use in 4.9.1, as I've already used this for the previous experiments, again with the sampled point cloud being scaled correctly (in contrast to the input of my method, where the scale is not correct).

I run the algorithm 5 times for each combination of $threshold \in [0.5, 1, 5, 10, 50, 100, 250, 500]$ and $voxel\_size \in [0.1, 0.25, 0.5, 1, 2, 5, 10]$, and compute the average fitness over these 5 runs. The result is plotted in 89. As one can see, the fitness scores do differ significantly depending on the chosen parameters, with the lowest fitness score being 0.05581, and the highest score being 0.29671 (as explained in 4.9.5.2, this is the fraction of inlier correspondences / number of points). This is still significantly lower than the score from the best fit found by my method, which is around 0.49.

[12] http://www.open3d.org/docs/0.13.0/tutorial/pipelines/global_registration.html

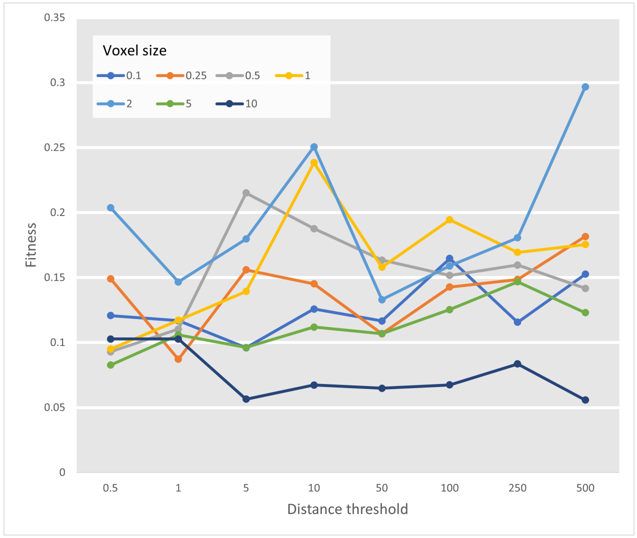


Figure 89. Plotting the fitness for the combinations of threshold and voxel size

Shown in 90 are the input (top left) and the outputs of running the global registration with $voxel\_size = 2$ and the various distance thresholds. As one can see, the point clouds are mostly oriented in an arbitrary way, with the alignment produced by the global registration not being usable for the purposes I intend it to.

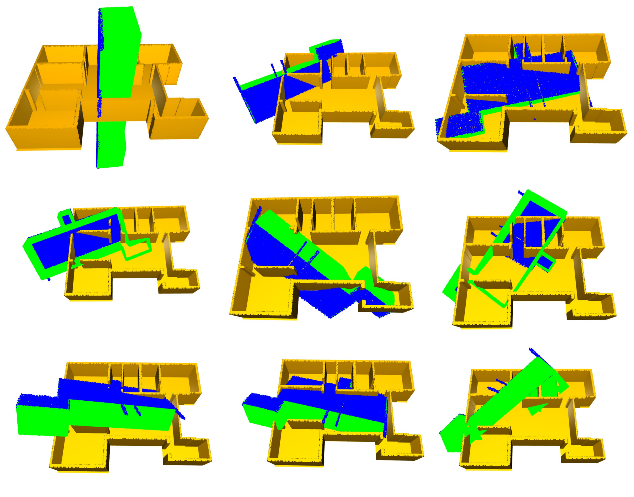


Figure 90. Input (top left) and outputs for the distance thresholds

Another issue with the global registration is again that the scales need to be correct to begin with, so passing in point clouds with different scales would result in even worse registrations.

5.3. Running method with different transformations applied on inputs

Finally, I run the method on variations of the input, to compare the result and the fitness scores after the global registration step (4.9) as well as after the final local optimization step (4.10).

As before, I ran every input 5 times and computed the mean and standard deviation of the results, which are shown in figure 91. The descriptions & visualizations of the inputs, along with the result data, can be found in appendix A.1.

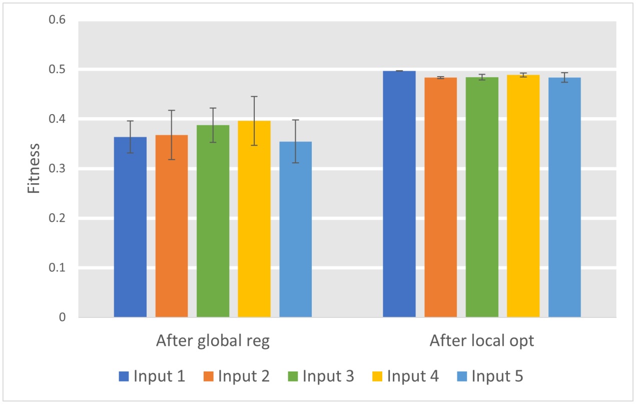


Figure 91. Results of running the method on various inputs

As before, the fitness is the fraction of inlier correspondences / number of points. As one can see, after the global registration steps, the results differ a bit, with the standard deviation being between 0.03 and 0.05 (i.e. an absolute difference of 3% to 5% in total fitness). However, after running the local optimization step, which improves the alignment & scaling on a local scale, the resulting fitness scores for the various inputs are very similar, with the standard deviation being less than 0.01 for each set of inputs.

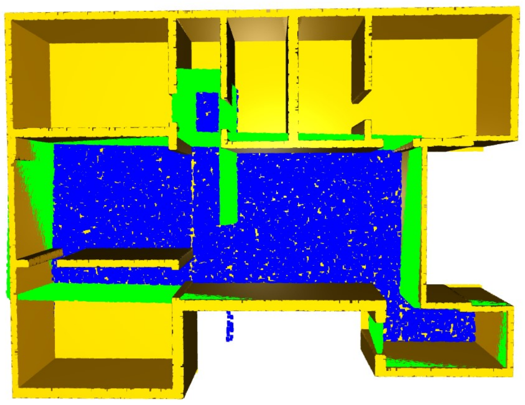


Figure 92. Resulting registration

This shows that the registration approach of my method is able to register the two point clouds consistently, without needing an initial alignment or have the point clouds be the correct scales already. It also shows that the local optimization step does indeed improve the registration, improving the fitness score by up to 0.14 from the result from the global registration.

In figure 92, one can see an example of a final registration produced during this experiment (result of a run with input 3).

6. Conclusion

In this section, I'll finally present my conclusions about the method developed in the thesis, discuss some of the more important limitations and some possible further work to tackle these limitations.

6.1. Limitations

6.1.1 Manhattan world assumption needs to hold

During many of the steps, the Manhattan world assumption 2.4 is assumed to hold, for example to filter invalid RANSAC planes in 4.6.1.5 or to compute the outer planes in 4.6.2. While the Manhattan world assumption does hold for a large number of buildings, it limits the type of buildings that can be processed.

Buildings with diagonal walls such as 93 will not work, as the diagonal lines are discarded by checking them against the principal components (and computing the principal components in regards to the walls as explained in 4.6.1.4 might also not work as expected).

This could be resolved by not checking the directions of the found planes, but this leaves the RANSAC step open to find other arbitrary oriented planes.

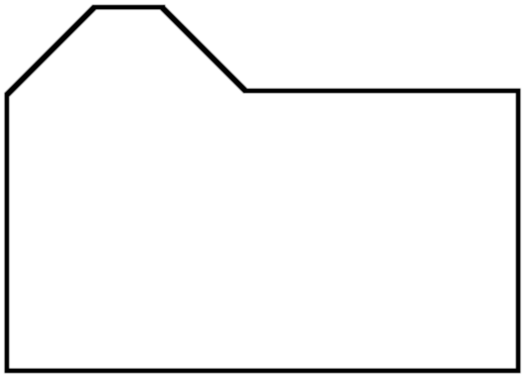


Figure 93. Floor plan with diagonal walls

Floor plans with curved walls, such as the example shown in figure 94 on the other hand do not work at all, as the method I presented only works with straight lines. Therefore, point clouds containing such walls are currently not supported and might yield suboptimal results.

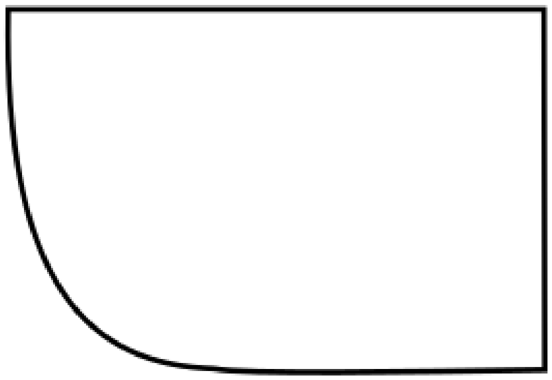


Figure 94. Floor plan with curved walls

6.1.2 Quality of RANSAC planes depend on input data

I introduced many additional checks and the step to find bounding planes shown in 4.6.2 to get a set of planes that should represent the structure well, however the quality of the found planes still depends a lot on the input data.

If the point cloud does not contain enough data along the walls, e.g. because the SfM step did not manage to track the walls or if the user did not capture any images containing enough views of the walls, the step to find planes might perform badly, negatively impacting the whole result.

6.1.3 Approach only works for single floor point clouds

Currently, the approach only works for point clouds containing data of a single floor. If the user changes floors during capturing of the data, the algorithm will not be able to align the point clouds. Furthermore, the algorithm will only find the floor plane of the lower floor, as it's assumed that the floor plane is on the very bottom of the point cloud, with only few points below this plane. Horizontal planes with more than a percentage of points below it are currently discarded as invalid horizontal planes.

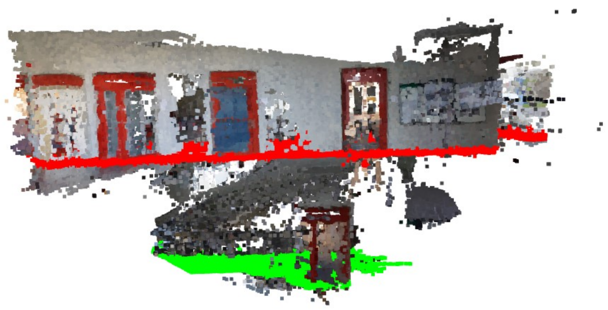


Figure 95. Point cloud with multiple floors, manually highlighted

Point clouds with multiple floors, such as the point cloud in figure 95 cannot be processed with the current method and would need to be cut in half before processing.

6.1.4 Running method on floor plans with repeating patterns

The global registration step 4.9 works especially well for smaller floor plans, where the rooms differ enough from each other. However, if the floor plan is large and contains repeating patterns, e.g. offices along a hallway all having the same size, this step might find a registration which is not optimal. One example is a floor plan shown in figure 96.

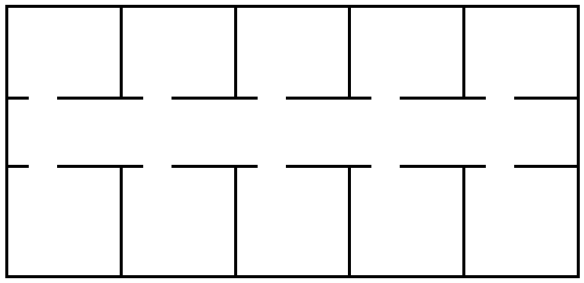


Figure 96. Floor plan with repeating rooms

If the user were to capture only a small subset of data, such as the one shown in figure 97 (path of user in red, walls captured in black), the algorithm might not find the correct location for this data, as the data might fit into multiple locations.

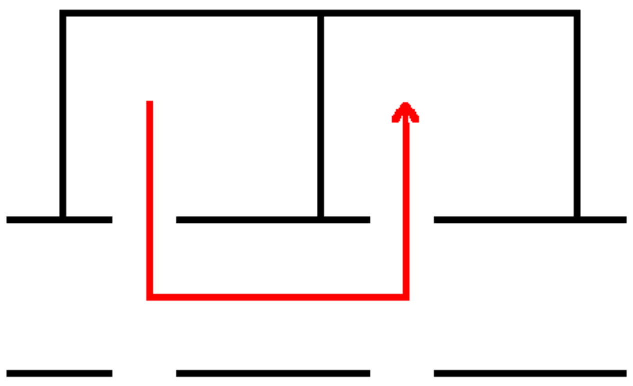


Figure 97. Example path, with captured data

6.2. Further work

6.2.1 Improve quality and reliability of finding planes

As previously explained, the quality of the whole result depends on the quality of the planes that were extracted from the point cloud. The current approach with RANSAC works well, however this step still depends on randomized algorithms (RANSAC), which means that the results might differ between runs. This can be partially avoided by increasing the number of iterations of each RANSAC step, which however also increases the runtime of the algorithm.

As the various parts of the method work independent of each other, one could replace the plane search with RANSAC by a more reliable approach, for example using a neural network or some analytical approach. Any method that is able to return a set of planes with corresponding bounding boxes works here, which means there is a lot of potential for improving this step of the method.

6.2.2 Use color information of previously added data

It is possible to run the method for multiple sets of inputs, e.g. from two different sequences of pictures, on the same CAD model point cloud. After the last step 4.11 of the run with the first data set, the user has a point cloud where the points from the CAD model P_{cad} are merged with the points from the captured data P_{capt}, resulting in a new point cloud P_{merg}.

When the method is now run a second time with the captured data from the second sequence and the new point cloud P_{merg}, the algorithm runs the same as if it would be run with the original CAD model point cloud P_{cad}.

However, the merged point cloud contains more relevant data in the form of point colors from the first captured point cloud, which could be taken into account to improve the accuracy of the registration.

6.2.3 Handle multiple floors

As elaborated in 6.1.3, the method currently only works for point clouds with a single floor. One improvement would be to add the ability to handle multiple floors, where stairs connecting the floors could be an important factor for initial alignment in the registration step. Usually, buildings have a small number of stairs connecting the floors, which would reduce the number of translations the algorithm would need to test to find the optimal global registration.

6.2.4 Incorporate position data

As shown in 6.1.4, the global registration step might struggle to find the correct location for the captured data if the floor plan (and therefore the CAD model point cloud) contains the same shape multiple times. One approach to tackle this issue would be to localize the position of the initial shot of the sequence of images used to produce the captured point cloud (using GPS, WiFi or some other technique) inside the CAD model.

With this localization, which would not have to be very accurate, we could constrain the search for valid registrations to a region around this point, e.g. by computing the point of the camera for this shot in the captured point cloud (which should be already done from the SfM framework) and forcing this point to be nearby to the previously located position of the camera in the CAD model point cloud.

For this to work, we would need to have some way of roughly locating the camera device for each of the images in the input sequence, e.g. by extracting GPS data from EXIF metadata. Furthermore, the CAD model point cloud would need to contain such information as well, e.g. by using the real-life location of the building to compute a location inside the building (and therefore on the floor plan) for a given GPS coordinate.

While this certainly would add additional overhead and require more data, it could be interesting to see whether the whole process could be improved to work on larger, repeating floor plans, as well as speeding up the global registration by not having to search over the whole area of the CAD model point cloud.

6.3. Summary

In this section, I'll summarize the thesis, its results and therefore whether the goals stated in 3 have been successfully met.

In general, the approach explained in this thesis works reasonable well to fit a noisy point cloud containing many points not related to the structure of the room into a 3D model of the building. As shown in 5.3, the method works with any oriented and scaled input, reducing the needed manual work to align the data.

The final result shown in 4.12 shows an example of such a fit, where the program was able to fit the captured data into the CAD model point cloud without requiring an initial alignment.

While there are still some limitations and challenges (as elaborated in 6.1), I think the approach can indeed be used as a baseline for such an approach. As the individual parts of the program are only loosely coupled (by their parameters and return values), one could replace individual parts and improve the performance of the whole program easily (for example replacing the RANSAC approach with a more sophisticated approach at finding planes in the point cloud).

One of the important findings is that reconstructing the captured data by creating a new point cloud from the extracted structure can reduce the complexity of the point cloud by a large margin, which means there are fewer points in this point cloud that cannot be matched. The task of registration can therefore be simplified, yielding a higher registration fitness in the intermediate steps. By keeping track of the needed transformation steps and then composing a final transformation matrix, the original captured data still can be fit into the CAD model point cloud, such that the goal can be achieved.

As explained in 2.4, using the Manhattan world assumption greatly simplifies the task at hand: First off, the planes found in the step 4.6 can only be oriented in one of three directions, which allows the program to discard invalid planes easily. Second, it allows to reduce the number of rotations in the global registration step to only 4 rotations about the Z axis.

While I am satisfied with the outcome, I think the tasks outlined in 6.2 still have the potential to make this approach more usable, and would definitely be an interesting project to tackle.

6.4. Acknowledgements

I would like to express my gratitude to Dr. Iro Armeni for working on this thesis with me and for her continued support during the thesis.

6.5. Resources

The code for the thesis is available on GitHub: `https://github.com/Adrian-Hirt/MasterThesisPointcloudRegistration`

References

[1] PJ Besl and ND McKay. A method for registration of 3-d shapes, ieee t. pattern anal., 14, 239–256, 1992. 2

[2] J.M. Coughlan and A.L. Yuille. Manhattan world: compass direction from a single image by bayesian inference. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 941–947 vol.2, 1999. 2

[3] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996. 5

[4] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, jun 1981. 2

[5] Shengyu Huang, Zan Gojcic, Mikhail Usvyatsov, Andreas Wieser, and Konrad Schindler. Predator: Registration of 3d point clouds with low overlap, 2021. 26

[6] Pushmeet Kohli Nathan Silberman, Derek Hoiem and Rob Fergus. Indoor segmentation and support inference from rgbd images. In *ECCV*, 2012. 27

[7] Vladimir Nekrasov, Chunhua Shen, and Ian Reid. Light-weight refinenet for real-time semantic segmentation, 2018. 27

[8] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 127–136, 2011. 26

[9] Rémi Pautrat, Juan-Ting Lin, Viktor Larsson, Martin R. Oswald, and Marc Pollefeys. Sold2: Self-supervised occlusion-aware line description and detection, 2021. 25

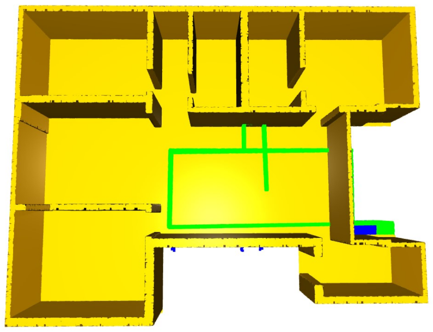
[10] Shimon Ullman. *The Interpretation of Structure from Motion*, pages 133–175. 1979. 2

[11] Zhengyou Zhang. Iterative point matching for registration of free-form curves and surfaces. *International journal of computer vision*, 13(2):119–152, 1994. 2

A. Appendix

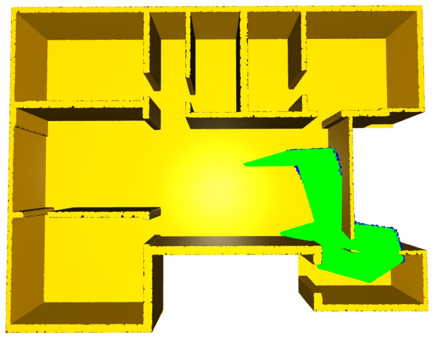
A.1. Inputs and results of global registration experiments

Input 1
Applied transformations to sampled point cloud: None.



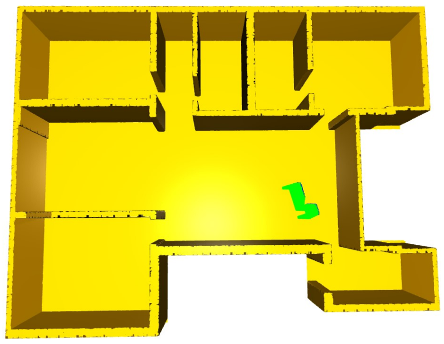
Run	Fitness after global reg.	Fitness after local opt.
1	0.352150837	0.496451156
2	0.37853988	0.497471111
3	0.324351409	0.496779142
4	0.353375645	0.497367116
5	0.409605463	0.495830183
Avg	0.363604647	0.496779742
SD	0.032077154	0.000605414

Input 2
Applied transformations to sampled point cloud: Rotated around all 3 axis.



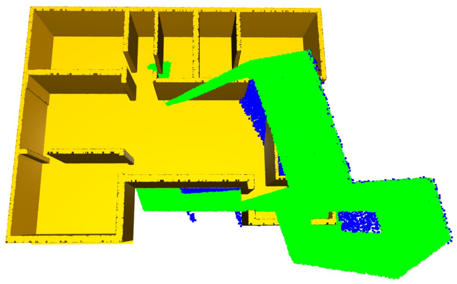
Run	Fitness after global reg.	Fitness after local opt.
1	0.408714694	0.483904708
2	0.398322384	0.481617809
3	0.274022937	0.485965618
4	0.361837954	0.481198827
5	0.395316038	0.483220738
Avg	0.367642801	0.48318154
SD	0.049380807	0.001711841

Input 3
Applied transformations to sampled point cloud: Rotated around all 3 axis & scaled down by factor of 0.3.



Run	Fitness after global reg.	Fitness after local opt.
1	0.321456408	0.483202739
2	0.403444308	0.481248825
3	0.411906618	0.480953838
4	0.385517574	0.49499722
5	0.415098541	0.479937883
Avg	0.38748469	0.484068101
SD	0.034575333	0.005565971

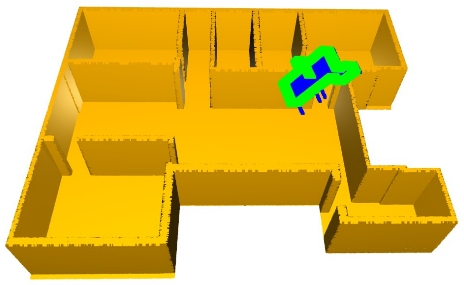
Input 4
Applied transformations to sampled point cloud: Rotated around all 3 axis & scaled up by factor of 2.



Run	Fitness after global reg.	Fitness after local opt.
1	0.41825335	0.483514725
2	0.318264484	0.483686718
3	0.44289797	0.491535372
4	0.359722377	0.491603369
5	0.440337008	0.492409334
Avg	0.395895038	0.488549904
SD	0.049029017	0.004053035

Input 5

Applied transformations to sampled point cloud: Rotated around all 3 axis, scaled down by factor of 0.3 & translated in positive Z direction.



Run	Fitness after global reg.	Fitness after local opt.
1	0.294399287	0.467934411
2	0.412203541	0.481515813
3	0.345915451	0.481154829
4	0.325650447	0.49046342
5	0.394499499	0.49613317
Avg	0.354533645	0.483440329
SD	0.043468931	0.009589502