Institute for Computer Science
Knowledge-Based Systems

Bachelor's thesis

Detection of Cylinders in 3D-Polygon-Meshes

Stephan Hennig

October 2012

First supervisor: Prof. Dr. Joachim Hertzberg
Second supervisor: Thomas Wiemann, Msc.

Abstract

One of the most important requirements for an autonomous robot to handle complex tasks is the ability to create a precise model of its environment from sensory input. One approach to tackle this problem is the generation of 3d polygon meshes out of 3d point clouds. The work group Knowledge Based Systems at the University of Osnabrück has developed an extensive software package aimed at this goal which already provides the means to extract planar regions from a polygon mesh. In this work, the existing software is extended with a RANSAC based approach to include the recognition of cylindric objects.

Acknowledgments and Expression of Thanks

First of all, I would like to thank Prof. Dr. Hertzberg and Thomas Wiemann for their ongoing support throughout the course of this work. Especially Thomas Wiemann deservers credit for always having an open door when there was yet another nagging question or a percieved dead-end during programming. Without his patience and insights this work would not have been possible. Overall it was a great pleasure and an immense learning experience to work in the friendly and helpful atmosphere of the robotics department. In this regard, thanks also goes out to Kai Lingemann, Jochen Sprickerhof and Sven Albrecht who helped me out with all sorts of things from setting up my workspace, to conquering LaTeX or by simply providing a second pair of eyes during debugging.

Contents

Chapter 1

Introduction

1.1 Motivation

The expectations for future autonomous robots are rather demanding. While being designed for a special purpose, it will be absolutely essential for these machines to navigate safely and independently through an unknown environment in order to perform their given tasks. In other words they should be able to translate the information of their sensors on the fly and construct a working model of their surroundings. The more accurate and reliable, the better the model will serve as a basis for solving all sorts of problems that stand between the robot and its task.

But generating a good model of the robots environment is a very expensive endevour in terms of the required computational resources since most robots rely on high resolution laser scans that provide them with thousands if not millions of datapoints. Hence, it is no suprise that any modeling software has to be based on very efficient algorithms. While there is a broad range of more or less bottom up based approaches that, literally speaking, try to make sense of the 3d point clouds directly, the scientific community has also been exploring other possibilities. One of those is the generation of 3d polygon meshes from point clouds. The usage of such meshes and the attempt of reconstructing the original environment within a reasonable approximation of its actual geometrical structure comes with some desirable side effects. Most importantly, the complexity of the sensor input can be significantly reduced while the mesh structures still contain enough information of the underlying scenery to provide a decent basis for further segementation or recognition techniques. Notwithstanding the fact that the theoretical construct of meshes originates from the realm of graph theory and comes with powerful, effective and well understood algorithms.

Nevertheless, merely generating a 3d polygon mesh, however sophisticated it might be, is only one side of the medal. For any higher reasoning that is required of the robot it is essential to also provide the tools to extract as much information as possible from the data. At the very least that means segmenting the mesh into regions of interest and shape primitives like planes, cylinders etc. Moreover, a successful clustering in different geometrical categories could be very beneficial for any number of optimization techniques that try to improve the overall quality of

a polygon mesh. For example when shapes were detected in pitted or incoherent regions of a mesh it would be possible to provide a smoother surface for these areas based on a model of the shape. In fact, both these aspects, shape detection and mesh optimization go hand in hand. Good mesh quality improves the chances of finding more complex shapes primitives and reliable shape extraction helps with optimization. Either way the result is a better map for the robot.

1.2 Goal and Scope of this Work

The work group Knowledge Based Systems in the robotics department at the University of Osnabrück and in particular Thomas Wiemann put together an extensive software package for mesh generation and surface reconstruction that, in its current state, is able to identify flat areas/surfaces with a recursive region growing algorithm using a normal criteria.

In this regard the contributions of this work in technical and scientific terms are:

1. Extending the Las Vegas reconstruction toolkit [16] to also detect cylindric surfaces by using a combination of region growing and RANSAC (RANdom SAmple Consensus)
2. Providing simple heuristics for the software to automatically determine the parameter settings for RANSAC in cylinder detection (when applied to novel datasets)
3. A fast, efficient and robust implementation with special emphasis on being augmentable

1.3 Related Work

Since point clouds are the very basis for the 3d polygon meshes that are discussed in this work, it makes sense to take a look at the existing methods to extract cylinders and other shapes primitives from them. In this respect, it is important to mention that any approach dealing with the segmentation and classification of point clouds, will basically face the same challenges as methods that are purely based on a mesh.

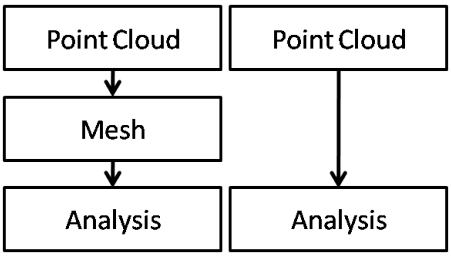


Figure 1.1: Mesh based methods are not far from point cloud based approaches

Ideally, it is desirable in both cases to find a robust and efficient solution that is capable of dealing with large quantities of sensor data, isn't susceptible to noise and doesn't overfit example datasets but provides a good enough generalization for unknown environments.

In particular to detect cylinders there already exists a broad variety of different approaches. For example [9] show, how the adaptation of a computer vision algorithm (Hough transform) can be used to detect cylinders in the reconstruction of an industrial site. While Hough transform seems to work reliable in this context and the authors even present a way to compute it more efficiently, it was unclear whether their solution would be robust enough to deal with a lot of sensor noise and fulfill the demands of a time critical application like the meshing software used in this work.

Another approach is presented by [12] where classifieres for geometrical surfaces like planes, spheres, cylinders etc. are learned by training support vector machines. The resulting classification of the point clouds is quite accurate even under noisy conditions but the entire approach is based on calculating "Point Feature Histogramms" that cannot be adapoted to work on a mesh and also require a lot of computing power.

In [10] the authors use perceptual grouping of edges for the detection of basic shapes as the first step in a method for tracking objects. Similar to this other approaches like [6] try to find symmetries or related features in point clouds in order to fit them to geometrical shapes and there are also attempts to match CAD models from databases to the data as presented by [14]. However, adapting either of these methods for this work would have been impractical since they tend to rely on predefined sceneries, consume a lot of computational resources or lack the generalization to deal with novel objects and seem to handle noise only to a marginal degree. Even though it has to be said that more often than not a mixture of different approaches or a problem specific adaption can be used to compensate for the shortcomings and applicational limits of individual methods.

One of the more frequently used solution for shape detection are random consensus based approaches as in [3, 13]. RANSAC (originally presented by [5]) has been proven to be very robust even when confronted with very noisy data. Moreover, its a very general approach and comes with a rather straightfoward paradigm. It is also favourable that it takes randomized input which accomodates quite well for some implementational details of the LAS VEGAS reconstruction toolkit (details see chapter: Regions and clustering) which is also why it has been chosen for this work. The biggest drawback with RANSAC however, is the fact that it is a highly parameter driven method that can be very expensive if applied to a huge point cloud. But while the latter is of no concern, since the amount of input data can be drastically reduced beforehand (by generating a mesh and filtering out planes), amends have to be made to tackel the parameter problem. In the case of this work this means introducing some simple heuristics.

1.4 Outline

Chapter2: The first part of this chapter provides a brief introduction into the LAS VEGAS reconstruction toolkit and the relevant classes, data structures and algorithms that provide the starting point for the implemented approach. The second part is focused on explaining the theory behind Region Growing and RANSAC.

Chapter3: In the third chapter, the details of the actual implementation are layed out as in how are planes and cylinders detected and how are the RANSAC parameter determined by the heuristics.

Chapter4: The fourth chapter looks at the results after testing the approach on different datasets.

Chapter5: Finally, in the last chapter a conclusion is drawn and possible improvements and alternatives are discussed.

Chapter 2

Prerequisites

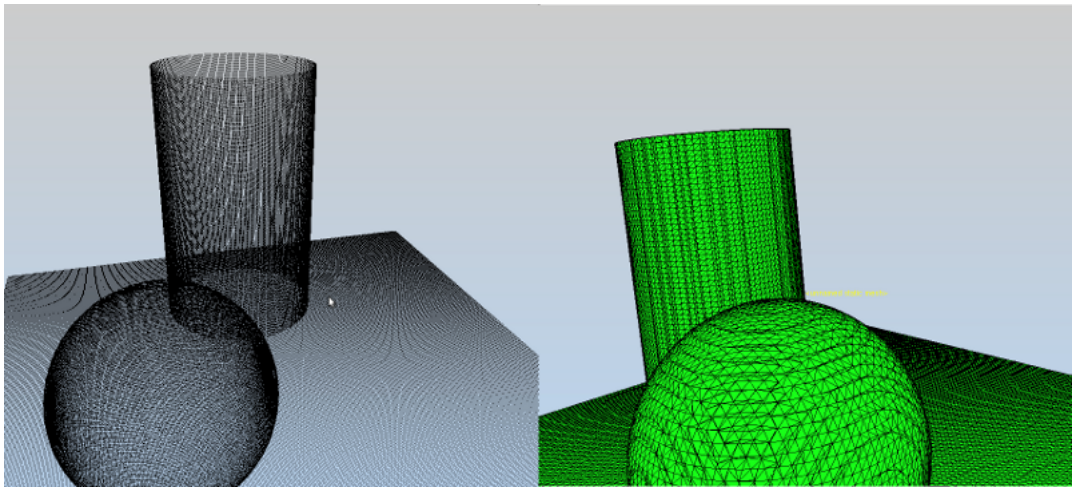


Figure 2.1: From pointcloud to mesh - the Las Vegas toolkit.

The Las Vegas reconstruction toolkit provides the basis for this work. The framework is written in C++ and released under the GNU Public License. The main author of the software is Thomas Wiemann who developed it as part of his PhD thesis about 3d maps based on polygon meshes for mobile robots (see [15]). Coauthors of the projects are Kim Oliver Rinnewitz, Sven Karl Schalk, Florian Otte, Lars Kiesow and Denis Meyer.

Offical releases, a detailed API documentation, example datasets and usage tutorials can be found under: `http://www.las-vegas.uni-osnabrueck.de`

As for the functionality of the toolkit that is relevant for this work. Las Vegas takes a point cloud and generates a polygon mesh using a marching cubes algorithm and various optimization techiques. Needless to say, that this is easier said than done but due to the scope of this work, a lot of the finer details concerning the mesh generation like the estimation of surface normals, variations of the marching cubes algorithm etc. have to be neglected. An extensive account about all the used algorithms and their technical background can be found in [15] and the above

mentioned web resource. Aside from this, it is noteworthy that most of the "heavy lifting" in terms of computing has been parallelized to optimize the performance of the software. The toolkit also provides a comprehensive viewer to visualize point clouds, normals and polygon meshes - which has been used to generate screenshot of meshes and datasets throughout the course of this work.

2.1 Half-Edge Mesh

So disregarding the exact details of its creating the starting point for the coming implementation is a polygon mesh that is stored in a Half-Edge mesh structure. Therefore, it is of importance to take a look at the general concept behind this data structure and its major components. First of all, the **HalfEdgeMesh** class in the Las Vegas toolkit is derived from an interface called **BaseMesh**. This interface describes the basic functions needed to dynamically handle and store the polygon mesh provided by the reconstruction algorithms.

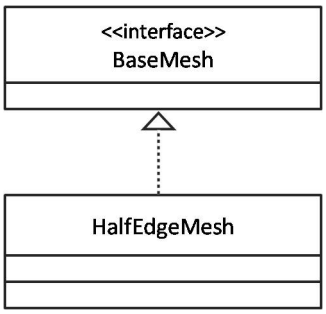


Figure 2.2: The HalfEdgeMesh class implements the BaseMesh interface.

The polygons in the HalfEdgeMesh are of triangular shape and consist of faces, edges and vertices. Respectively, the classes representing them are called **HalfEdgeFace**, **HalfEdge** and **HalfEdgeVertex**. Now the special property of a Half-Edge Mesh is the fact that each of its components is linked with its neighbouring components. But before these links can be described in more detail it is necessary to understand the role that each component plays in the HalfEdgeMesh.

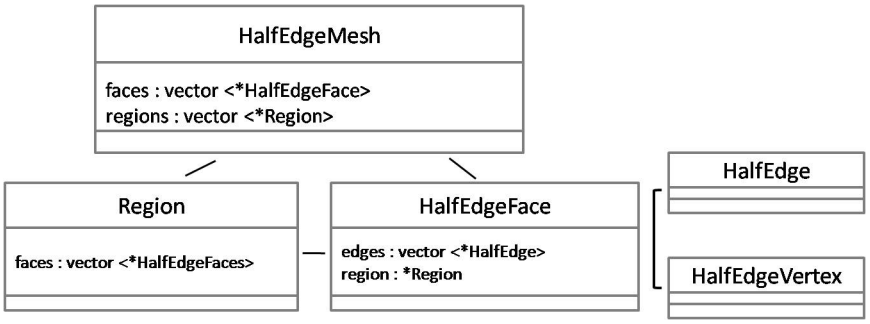


Figure 2.3: A simplified overview of the mesh components (Region class will be explained later).

HalfEdgeFace	HalfEdge	HalfEdgeVertex
visited : bool normal : Normal edges : vector <*HalfEdge> region : *Region	start : *HalfEdgeVertex end : *HalfEdgeVertex face : *HalfEdgeFace next : *HalfEdge pair : *HalfEdge	position : Vertex normal : Normal in : vector <*HalfEdge> out : vector <*HalfEdge>

Figure 2.4: The essential components of the HalfEdgeMesh and their most relevant attributes.

2.1.1 Vertices, Edges and Faces

The **HalfEdgeVertex** represents the vertices of the mesh. As such, it stores the coordinates of a vertex and additional attributes like color information or a normal corresponding to its position. Moreover, it contains a list with pointers to the edges that are connected to the vertex.

Theses edges are implemented by the **HalfEdge** class that contains the two vertices the edge is connected to. The object also has pointers to its associated face, its neighbouring edge and the next edge. Note that "next" in this context means the counter-clockwise traversion of a face.

Finally, the **HalfEdgeFace** represents the faces (or facets) of the mesh. Even though a face is formed by three vertices that are connected via three edges, the class only contains pointers to the edges since their pointers can be utilised for accessing the vertex informations. In addition to this, faces have pointers to their corresponding faces. Thus, a face somewhere in the polygon mesh knows its three neighbours. It is also relevant for the course of this work that each face object comes with methods to calculate its surface area, determine its centroid and access its surface normal. Furthermore, a face can be marked as used which is later important to ensure that no face is visited twice during clustering.

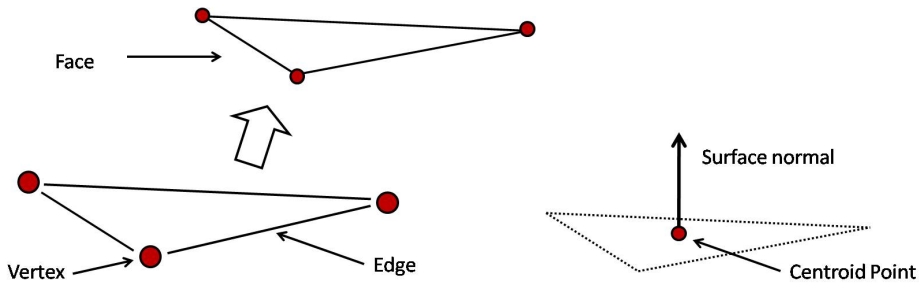


Figure 2.5: The smallest mesh components and important face properties

2.1.2 Linking

The actual algorithms for generating a polygon mesh are omitted in this introduction. This makes it even more crucial to clarify how exactly the previously described components of the mesh are linked with each other. In this sense, it is helpful to take a look at the initial steps of creating a HalfEdgeMesh.

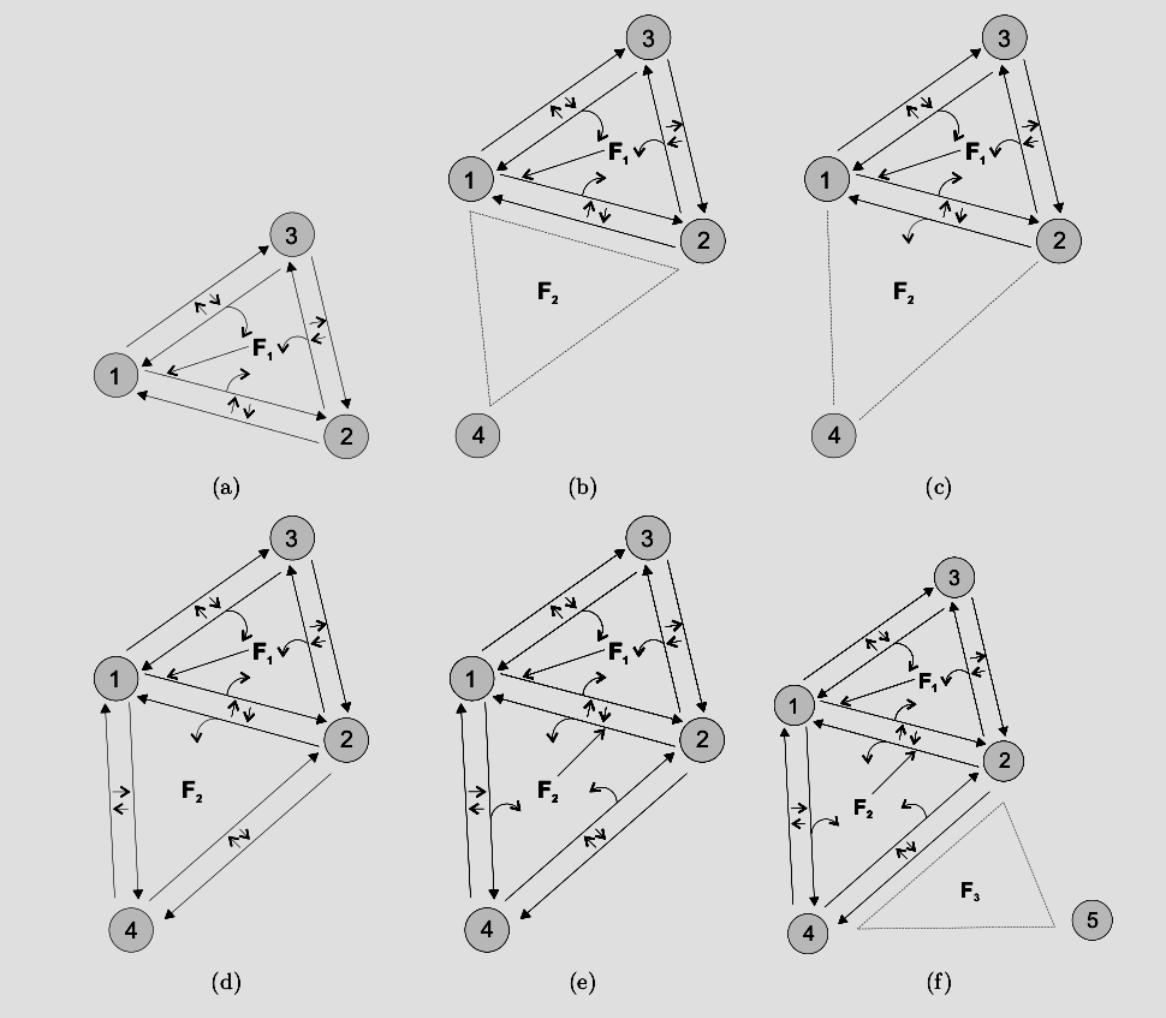


Figure 2.6: The picture is taken from the PhD thesis of Thomas Wiemann [15] who explains the creation of a HalfEdgeMesh in the following way: (a) The first face is initalized and its components are linked in the correct order. (b) A new face is created but is not yet linked. (c) All Half-Edges that have been edges of borders previously but now share vertices with the new face will get linked with the new face. This makes the new face part of the mesh. (d) If there are no more edges that border the new face like in the afore mentioned step the missing edges are newly generated and linked. (e) As soon as a another new face is added the whole process is repeated.

2.1.3 Regions

A region in the mesh can be defined as an accumulation of joint faces or simply put an object that gets faces assigned. But region objects are not merely keeping a list of pointers to all the faces belonging to them. They also come with some important methods of their own for optimization and further analysis (e.g regression plane for planar regions or region contours). To cluster a mesh into different regions the Half-Edge Mesh class implements a recursive region growing algorithm.

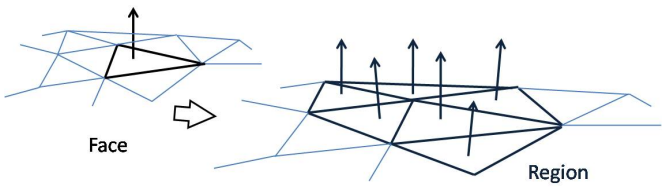


Figure 2.7: A face within the mesh compared to a region of faces

2.1.4 Region Growing

In its original form, Region Growing represents a method to segment images [2] but has been adapted to work on a polygon mesh. Its main purpose in the Las Vegas toolkit is to find planes for optimizing the quality of the mesh and to provide clustering. The basic principle is quite straightforward. A region is expanded from a face to its neighbouring faces as long as certain conditions are met. The algorithm aborts a recursion run-through if there are no more neighbouring faces, the neighbouring faces have already been visited or the normal criteria is below the user defined threshold.

Hereby the motivation regarding the normal criteria is the fact, that the surface normals of polygons representing a plane can be expected to point roughly in the same direction. Of course sensor noise has to be factored in which is why a threshold is introduced that determines how much the compared normals can differ.

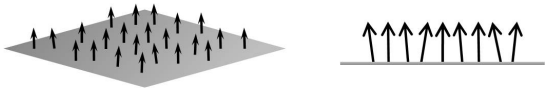


Figure 2.8: Surface normals of polygons representing a plane will deviate due to noise.

Comparing the normals of neighbouring faces for each and every face in a large polygon mesh can be expected to result in a lot of calculations. This is why the whole implementation of the region growing algorithm is based on pointer operations to ensure maximum efficiency. For the same reason it is also desirable to keep the underlying equation as simple as possible. Thus, all the vectors representing surface normals are normalized in a previous step of the mesh generation to allow the normal criteria to take the following shape in mathematical terms.

Let $\vec{v}$, $\vec{u}$ be surface normals belonging to neighbouring faces f_i and f_j.

$\forall \vec{v}, \vec{u} \in [-1, 1]^3, threshold \in [0, 1]$:

$$result = |\vec{v} * \vec{u}| > threshold \tag{2.1}$$

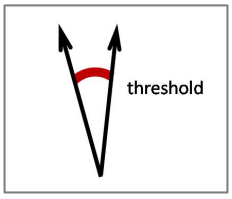


Figure 2.9: The normal threshold determines how far two normals can deviate.

So during the recursion, the face f_j belonging to the normal vector $\vec{u}$ is added to the current region if *result* is above the threshold. Afterwards, in the next recursion cycle, the neighbours of f_j are checked if they hold against the criteria and the other conditions. This continues until no more faces can be assigned to a region and no more regions are initalized once every face of the mesh has been visited. Just to clarify the overall principle a bit further the following figure depicts the intial steps of the algorithm.

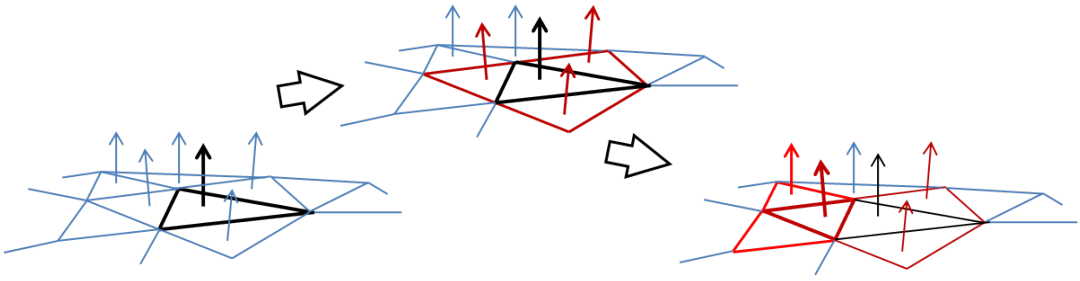


Figure 2.10: After choosing a start face a region is recursively expanded

One problem that can occur with this approach is that finding a good threshold for the normal criteria highly depends on the quality of the mesh. Which then again depends on the noise factor in the underlying point cloud. Thus in a very noisy dataset the chances to detect all planes with a strict criteria are very low, simply because the calculated normals of the polygons are more “chaotic” in their alignment. This issue can be resolved by running more than one iteration of the region growing algorithm with optimization techniques. In the current state of the reconstruction toolkit the prefered method for optimizing planes is pulling all the faces of a planar region into a regression plane using a least square algorithm.

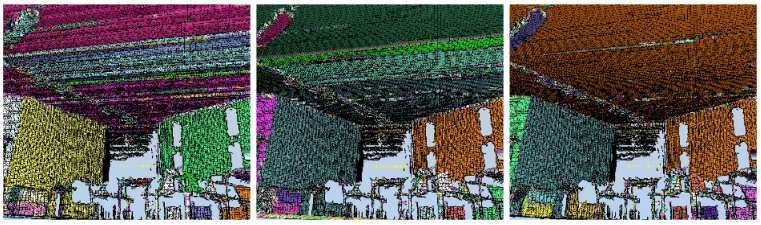


Figure 2.11: Another picture taken from the PhD thesis of Thomas Wiemann [15]: By running more than one iteration of region growing with optimization the final results for plane detection can be improved significantly.

In the end, the implemented Region Growing variant benefits greatly from the unique properties of the Half-Edge Mesh like quick access to normals, neighbours etc. and has a complexity of $\mathcal{O}(n)$ since none of the faces is visited twice (n being the number of faces). It therefore represents a very fast method for plane detection and clustering that can deal with noise by introducing optimization techniques and running multiple iterations.

Nonetheless it is limited when it comes to detecting more complex shape primitives, since the surface normals of these objects cannot be captured by a simple criteria. This is where methods like RANSAC are more suitable candidates.

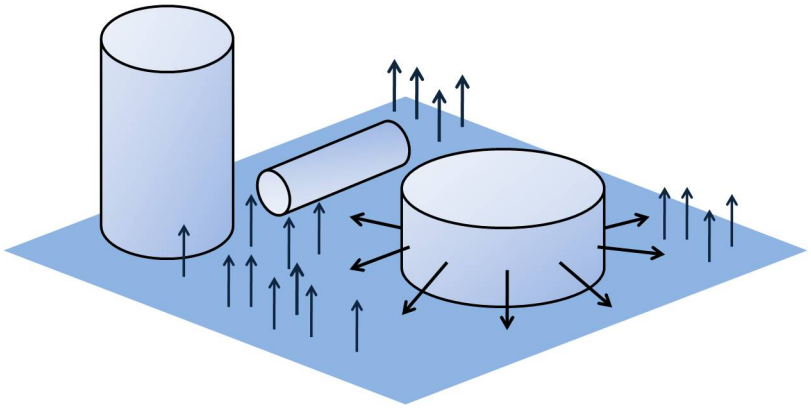


Figure 2.12: The surface normals of shape primitives such as cylinder are not as "nicely" aligned as those belonging to a planes. They also vary depending on the size and the orientation of the object.

2.2 RANSAC

RANSAC - short for "RANdom SAmple Consensus" was first presented by [5] and describes an iterative way to approximate the parameters of a mathematical model from data. In the RANSAC paradigm it is assumed that a model and parameters exist that match the data optimally and that the data itself can be divided into two sets, one containing "inliers" and a second containing "outliers". Hereby the "inliers" can be described by a model and its respective parameters while the "outliers" do not support it (either being noise or because a wrong model is matched to the data).

RANSAC starts with a randomized subset of the data as input and assumes the datapoints to be hypothetical inliers of a model. Subsequently, the hypothesis underlying these hypothetical inliers is testes against the rest of the data. Either a model is being rejected because not enough datapoints can be classified as inliers or its being kept since it contained less errors than the previous model. This entire evaluation principle is iteratively repeated until a fixed number of iterations is reached.

The RANSAC article in Wikipedia [17] describes the details of this iterative proccess:

1. A model is fitted to the hypothetical inliers, i.e. all free parameters of the model are reconstructed from the inliers
2. All other data are then tested against the fitted model and, if a point fits well to the estimated model, also considered as a hypothetical inlier
3. The estimated model is reasonably good if sufficiently many points have been classified as hypothetical inliers
4. The model is reestimated from all hypothetical inliers, because it has only been estimated from the initial set of hypothetical inliers
5. Finally, the model is evaluated by estimating the error of the inliers relative to the model

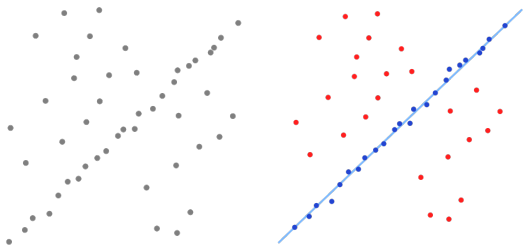


Figure 2.13: Wikipedia's example [17] shows a two dimensional dataset where RANSAC fits a line. Outliers of the model are marked red and Inliers are marked blue.

Despite being quite general in terms of its application, RANSAC is a highly parameter driven method. Which means that finding an sufficiently accurate set of parameters is essential for the

final result of its output. But especially for complex datasets, determining such parameters can be very time consuming and more or less dependent on the experience of the person handling the software.

In the generic form of RANSAC the main parameters in order of their importance are:

- The distance threshold of how far outliers can deviate from hypothetical inliers
- The number of total iterations
- The number of inliers required to accept a model

The **number of inliers** for acceptance is dependent on the data and the respective model which is supposed to be found. Its a parameter that has to be determined empirically. The same goes for the **number of iterations**, even though, in this case there is also the possibilty to deduce it theoretical. The most crucial parameter however, is the **distance threshold**. Here a wrong setting can lead to severe over- or underfitting of the model. In certain data distributions a distance threshold that is too high can lead to situations where too many outliers are detected as inliers. Thus, its possible to wrongly fit a model and yet find the same amount of outliers as with a correct match. To give an example the following pictures depicts such a situation.

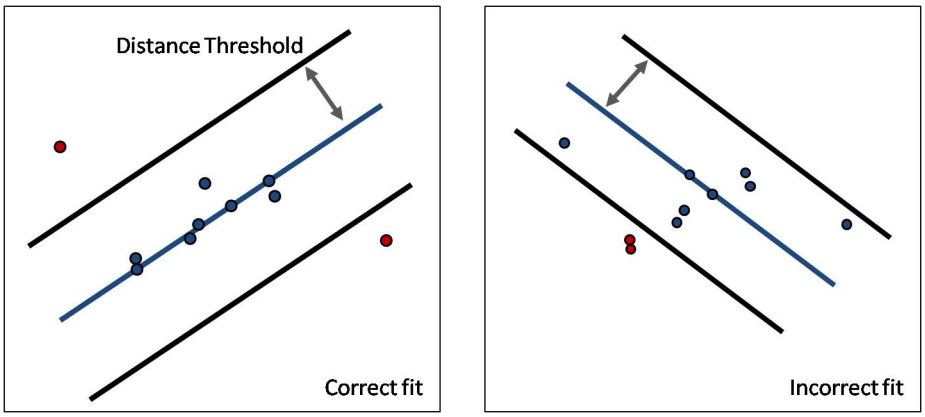


Figure 2.14: A line is to be fit but the distance threshold is too high which allows incorrect fitting in the second case. Just as before outliers are red and inliers are blue. (Pictures are a recreation of the example situation provided by [18])

Similar to this, a distance threshold that is too low can cause problems by allowing a model to be fit despite the fact that only a fraction of the actual inliers are considered.

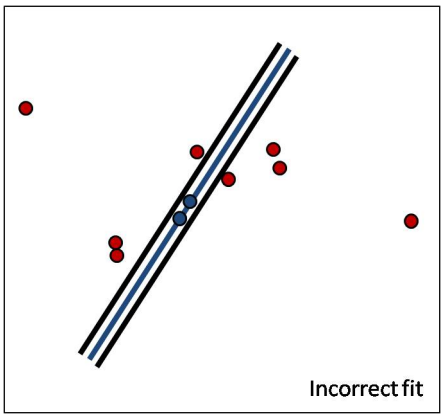


Figure 2.15: Same data but this time the distance threshold is too low and not enough inliers are used for fitting. (Also a recreation of the example provided by [18])

In the end, finding a threshold can also be done theoretically if the standart deviation of all the data points is known. But since this is most often not the case, it is once more a task that has to be solved empirically.

With this whole aspect of parameterisation comes an important conclusion. While RANSAC can deal with a whole lot of noise and still work reliable, it cannot be used to find different types of the same model with one set of parameters. For example using fixed parameters and trying to find all the circles within a large data sample wont work in one go. Instead one would have to segment the data first, determine good parameters and then apply RANSAC to find all the circles.

2.3 Summary

This chapter took a look a the starting point of the coming approach. The software takes a point cloud as input and produces a polygon mesh that is stored in a Half-Edge Mesh structure. Here the existing solution for clustering is a recursive Region Growing algorithm that exploits the normal properties of planes and the advantages of the data structure. The current Region Growing implementation is a very fast and efficient method but lacks the generalization to work on other shape primitives besides planes. On the other hand, RANSAC represents a slower yet robust way to find more complex models in the data as long as its parameters are well chosen.

Chapter 3

Detecting Planes and Cylinders

In the coming chapter the details of the implemented approach are clarified. The main aspect of this work was to expand the functionality of the Las Vegas reconstruction toolkit to detect cylinders. This was done by combining Region Growing with a standart RANSAC variant that estimates the model of a cylinder.

For datasets in a robotic context it is a reasonable assumption that planes belong to the most frequently occuring shape primitives. In particular when it comes to an indoor setting it is very likely that doors, walls, the ceiling etc. will make up a large majority of the total datapoints. Taking this in consideration, it seemed pragmatic to look for planes first in order to narrow down the subsequent search for more complex shapes, like cylinders, which might not occur that often in the data.

Since the existing implementation of the Region Growing algorithm already excels at clustering and plane detection there was no need to replace it with a different method and it was used for just that. This left cylinder detection as an exclusive task for RANSAC.

As for an overview of the implemented approach:

Algorithm 3.1 General schematic for finding planes and cylinders

Require: Initialized polygon mesh in a Half-Edge-Mesh structure

```
Apply Region Growing with high normal threshold to find planes
Apply Region Growing with low normal threshold to recluster non planar regions
for all non planar regions do
    Estimate RANSAC parameters with heuristics
    Apply RANSAC for cylinder detection
    if Cylinder model fit then
        Label region as cylinder
    end if
end for
```

For this to work a number of changes to the present architecture of the framework were required. The most noteworthy ones included adding a labeling object for regions, modifying the clustering algorithm and implementing RANSAC for a cylinder model in the first place.

3.1 Modifications to Las Vegas

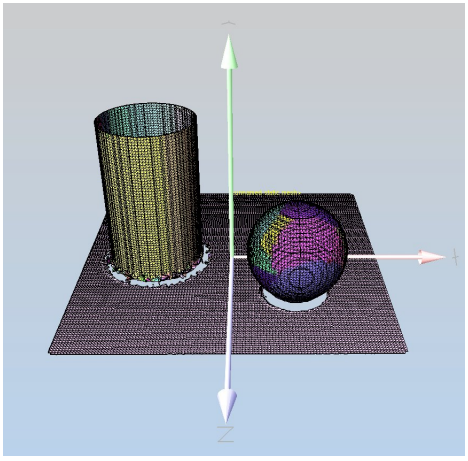


Figure 3.1: Clustering of a simple example dataset with a low normal threshold.

The above picture features the existing clustering method in the Las Vegas toolkit. While the results look promising for the human eye the software was limited to differentiate only between planar and non planar regions. That is why the region class got a new property in form of a label object. With this simple label attribute in place it would be possible to flag regions later on and expand the possible categories for regions indefinitely.

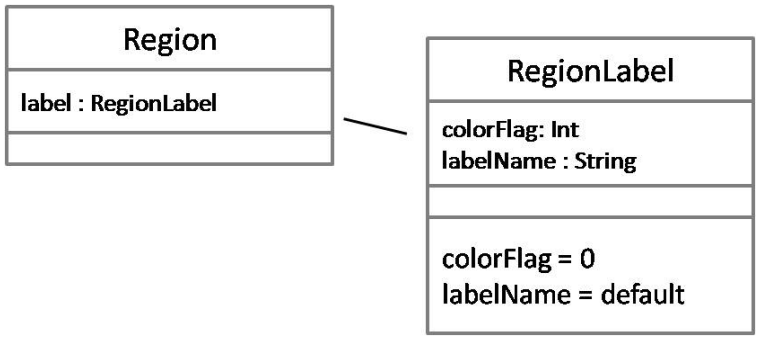


Figure 3.2: The new label attribute for the region class

Another aspect of this was to colour regions according to their specific label. Consequently, a new colour classifier was added that would paint default regions grey, planes green, cylinders

blue and other regions yellow. This entailed some changes to the classes involved in the colouring process of the final polygon mesh. But since these modifications are not relevant for the theoretical nature of this approach they won't be explained any further.

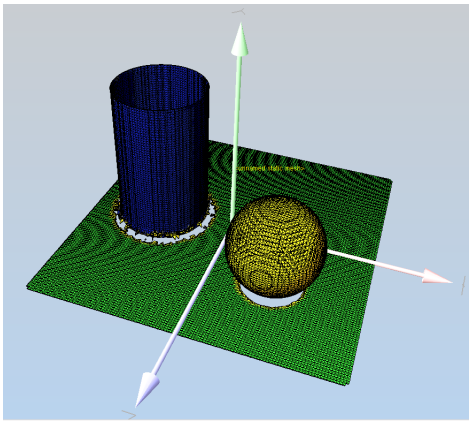


Figure 3.3: New colouring for regions with label.

However, to make use of this label a new method for clustering had to be added to the earlier described HalfEdgeMesh class. The task of this function was to flag regions after segmenting the mesh with Region Growing.

As it has been previously described (see 2.1.4) the implemented Region Growing method takes a **normal threshold** and tries to establish regions by connecting neighbouring faces whose surface normals match the normal criteria. Executing the Region Growing function changes the face and region attributes of the HalfEdgeMesh class. Namely, the list of face pointers and the list of region pointers are altered in the following way:

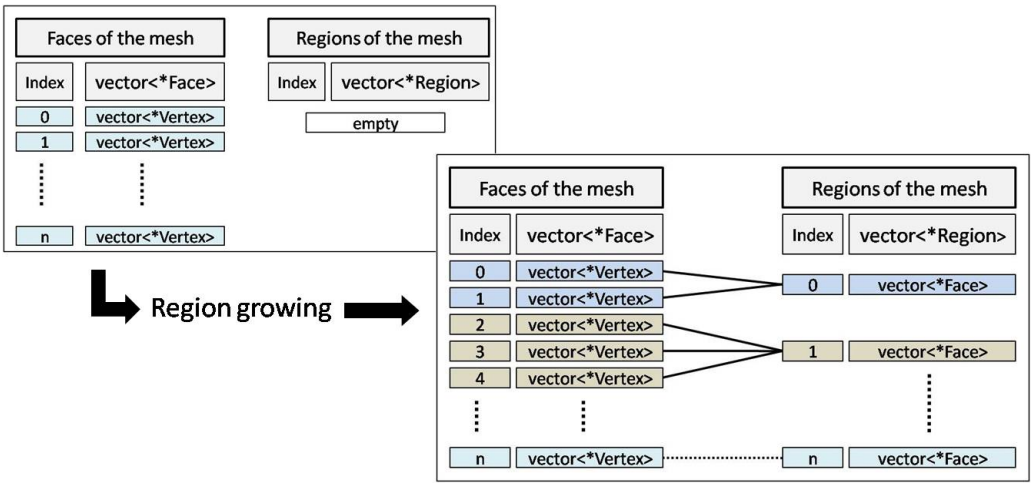


Figure 3.4: As a results of Region Growing faces are linked with regions and vice versa. However, note that the faces of the mesh are not ordered in any particular order.

So the new clustering function just had to call the established Region Growing function with the appropriate parameters and change the label of a found region to a user given value. But aside from the **normal threshold** another parameter for the **minimal region size** was included (following the style of existing methods in the HalfEdgeMesh class). This additional parameter determines at which size a list of faces is accepted as a region. Its purpose is to ensure more robustness before adding regions to the HalfEdgeMesh. At last, the function was coined "specificClustering" and got implemented in the following way:

Algorithm 3.2 Labeling regions while clustering

```
1: function specificClustering(label, threshold, minRegionSize, regions, faces)
2:     for all face in faces do
3:         if face ≠ used then
4:             R ← newRegion
5:             faceNormal ← face.getFaceNormal
6:             regionSize = regionGrowing(face, faceNormal, threshold, R)
7:             if regionSize >= minRegionSize then
8:                 R ← label
9:                 regions ← R
10:            end if
11:        end if
12:    end for
13: end function
```

Besides all the so far explained changes it was also necessary to extend some of the classes dealing with the paraphernalia of bringing everything together. For example adding new program options etc. But even though these amendments are important for smoothly running the software, they are once more not relevant for approach itself and won't be discussed in detail.

3.2 Shape Extraction Principle

More importantly, after establishing a way to differentiate between regions it was possible to add a method for shape detection. Applying this method to regions of a certain category was simply a question of filtering the right label. But aside from implementing a way to detect cylinders it was desirable to leave room for additional methods that would deal with the extraction of different shapes. Considering this, the following principle seemed feasible:

Algorithm 3.3 Step wise, nested approach to the extraction of shapes

Find planes first and label them

```
for all regions do
    if region label ≠ plane then
        Look for Shape_1 with method_1
        if Shape_1 found then
            label ← Shape_1
        else
            Look for Shape_2 with method_2
            if Shape_2 found then
                label ← Shape_2
            else
                ...      // and so on till Shape_n
            end if
        end if
    end if
end for
```

In this general shematic, it is assumed that planes are labeled first as they are the most basic of all shape primitives. Subsequently, it is expected that $Shape_j$ is more complex than $Shape_i$ (with $i < j$). This way, the most sophisticated and presumably most costly methods would only be applied to a smaller and smaller getting portion of the mesh.

For the detection of cylinders, RANSAC was the chosen method. At least partially because the literature about its application in the shape extraction context seemed most convincing. But also because the RANSAC paradigm agrees with the way faces are added to a region. Which is to say, that the Region Growing algorithm adds them in a rather unpredictable fashion so any method working with the list of faces would have to to compensate or agree with that fact. In this sense, the handy aspect about RANSAC is that it grabs random samples. Meaning, no specific ordering of the input data is required in the first place.

3.3 Adapted RANSAC

Since RANSAC is such a frequently used method in research involving point clouds and meshes it made sense to look for already existing implementations. This quickly lead to a project that is related to the Las Vegas reconstruction toolkit. Namely, the **SA**mple **C**onsensus package (short **SAC**) of the Point Cloud Library.

The **P**oint **C**loud **L**ibrary (short **PCL**) is an international, open project that provides a vast amount of speed optimized algorithms for all sorts of task that are related to point clouds like overall processing, visualization, registration, segementation etc. It is freely available for scientific and commercial use and can be found under: `http://pointclouds.org/`

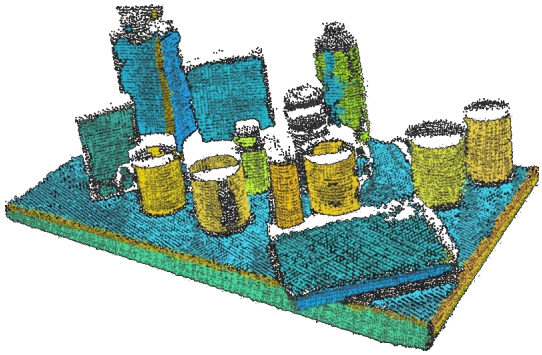


Figure 3.5: PCL SAC provides the means to detect differents models. A detailed API description and documention can be found under `http://docs.pointclouds.org/trunk/group__sample__consensus.html` (also source of the above example picture)

Some functions of the PCL are already used in other parts of the Las Vegas toolkit which made it very comfortable to include its SAC library as well. But aside from the convenience aspects, PCL-SAC was chosen because of the following advantages:

- It implements models of RANSAC estimators for lines, planes, cylinders and spheres.
- In addition to the standart RANSAC implementation it provides a great variety of improved versions (e.g. M-Estimator-Sample-Consensus, Progressive Sample Consensus, Least Median of Squares to name a few)
- The library is well tested, speed optimizied and under continuous development. It can be expected that future releases will further improve its current state (e.g. models for cones and tori are work in progress).

All these points level the playing field for additional functionality and improvements if this work was to be expanded in the future. They also go hand in hand with the goal of providing a fast solution.

3.3.1 Estimator Input

Now the assigned task of any RANSAC implementation was to detect cylinders in the regions that were provided by the previous clustering step. In order to apply these regions to the cylinder estimator provided by PCL SAC, it was necessary to convert them into a conform input format. The relevant class (later described in more detail) takes a point cloud and normals as input. But initializing the required point cloud object meant adding to the complexity of the whole approach. Therefore, it would have been counterproductive to use the vertex coordinates of faces. For example: Considering n faces the result would be a cloud with $3 * n$ datapoints and $3 * n$ normals left to process. This is why the input for the cylinder model was constructed

with the centroid and the surface normal of each face belonging to a non-planar region.

In the end, building the input objects was a rather straightforward task:

Algorithm 3.4 Construction of the Input Point Cloud and Normals

```
for all faces in non planar region do
    listOfPoints ← faceCentroid
    listOfNormals ← faceNormal
end for
pointCloud ← listOfPoints
normals ← listOfNormals
```

3.3.2 Sample Consensus Cylinder Model

Most of the source code for initializing and using the SAC cylinder estimator has been directly taken from the excellent usage tutorials provided by PCL (see [7] and [8] respectively). The general principle for using the SAC library looks like this: After choosing a RANSAC method and a model, all the relevant parameters are setup and the estimator gets its input. Subsequently, it computes the model coefficients and tries to fit the model.

In this regard the used model is represented by the SACMODEL_CYLINDER class (author being Radu Bogdan Rusu [11]) and calculates seven coefficients: x_1, y_1, z_1, x_2 ,x_2 , x_2, r

- A point axis of the cylinder - P1 (x_1, y_1, z_1)
- The axis direction - P2 (x_2 ,x_2 , x_2)
- The radius - r

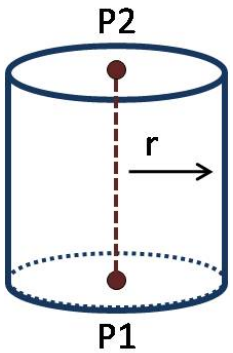


Figure 3.6: The SACMODEL_CYLINDER class comes with methods to compute cylinder coefficients before the model is applied to the data.

The chosen RANSAC method was the standart version of the algorithm with meant that the following parameters had to be setup for the SAC library to process the input and compute the model:

- minRadius
- maxRadius
- NormalDistanceWeight
- MaxIterations
- DistanceThreshold

As the name might suggest, **minRadius** and **maxRadius** are used to limit the radius of the cylinder model. The **NormalDistanceWeight** determines to which degree the input normals are considered or simply put, how much to trust the underlying normal estimation. Finally, **MaxIterations** and **DistanceThreshold** fulfil the exact same purpose as mentioned before (see chapter 2.2 about RANSAC). While the first parameter is responsible for the total iterations of the selected RANSAC version, the second one describes how far points can deviate from the cylinder model and still be considered as inliers.

Since the SAC library takes care of most of the computing all that was left was setting up a way to evaluate the results. With regard to the fact that the the estimator is applied to a region, it seemed logical to calculate the inlier percentage of a region and introduce an acceptance threshold. For example if 80% of the faces in a region are found to be inliers and the threshold is set to 70% the region will be marked as cylindric. The simple equation looks like this:

$$(inliers/regionSize) > acceptanceThreshold \tag{3.1}$$

The final implementation looks like this:

Algorithm 3.5 Integration of the SAC Cylinder estimator

Require: point cloud - Constructed from face centroids of a region
Require: normals - Constructed from face normals of a region

```
seg ← new SACSegmentationFromNormals

Set ModelType(SACMODEL_CYLINDER)
Set Method(SAC_RANSAC)

    // RANSAC parameters

Set NormalDistanceWeight
Set MaxIterations
Set DistanceThreshold
Set minRadius
Set maxRadius

    // Input

Set input point cloud
Set input normals

    // Compute segementation

Estimate model coefficients from input and apply RANSAC
inliers ← result

p ← inliers / region size
if p >= acceptance threshold for cylinders  then
    region label ← cylinder
end if
```

3.3.3 Summary

The input for the estimator is build with the centroid points and the normals of faces belonging to a region. Once the coefficients have been obtained by the model class, a standart version of RANSAC is applied and the inlier percentage is checked against an acceptance threshold to determine whether or not a cylinder was found.

3.4 Parameter Heuristics

The estimator requires a whole lot of parameters and having a fixed set of those would not cover all possible cylinder models in a large dataset. Being too generous with the settings would lead to over- and choosing too harsh values to underfitting. Hence, applying regions to the SAC cylinder estimator in the previously proposed principle for shape extraction (see algorithm 3.3) would have been impractical.

This motivated the introduction of heuristics to fix this issue. By approximating good parameter settings, the software was expected to handle different sized cylinders without any user driven influence. In this regard, the first step in developing the heuristics was to find out which parameters had the most effect on the final result. So after some initial testing it quickly became clear that the following parameters had the most influence:

- **minRadius**
- **maxRadius**
- **DistanceThreshold**

This was not suprising considering their role in the RANSAC paradigm. The distance threshold determines how far points can spray off the model (see 2.2) and the radius is a very important attribute to describe a cylinder. So these were the most promising candidates for heuristics. The rest of the parameters got put aside and were read in from a parameter file. After all, changing too many parameters at the same time is seldom a good idea.

During the early testing stages it was also observed that in particular the smaller regions would reflect wrong detection. For example a lot of the small cluster on sphere objects were wrongly classified as cylinders. Hereby, small means regions with a maximum size of 300 faces. Presumably, this happened because the distance threshold was too high in these regions. But lowering the value was no option since it would rule out large cylinders and there is no telling how big these can get in unseen datasets. And so, instead, it seemed a better idea to lower the threshold for smaller regions. Therefore the following, very basic heuristic was introduced:

Algorithm 3.6 Heuristic to determine the Distance Threshold

Require: Base value for the threshold

```
if big region then
    take base value
else
    if region middle sized then
        max(base value - 0.05, 0.2)
    else      // small region
        max(base value - 0.1, 0.01)
    end if
end if
```

The radius heuristic was slightly more sophisticated as it is inspired by the underlying equation to calculate the surface area of a cylinder. The surface area A of a cylinder is calculated by multiplying 2π with the cylinder radius r and the height h.

$$A = 2\pi * r * h \tag{3.2}$$

So by rearranging the formular we get:

$$\frac{\pi A}{2h} = r \tag{3.3}$$

Since the HalfEdgeFace class comes with a method to return the area of a face, it is also possible to calculate the area of a region. Thus, with a few adaptions to the above equation it seemed possible to at least get a good estimate of radius. The formular could not be used unchanged since there were still uncertain factors. Namely, it is very hard to tell how the mesh would cluster different types of cylinders. The following picture clarifies what is meant by that.

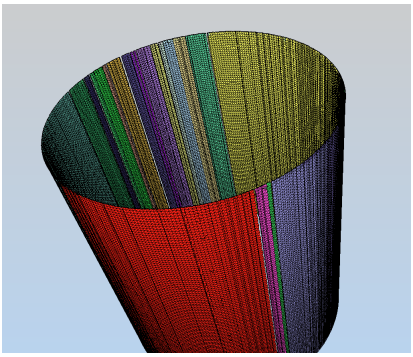


Figure 3.7: Coarse clustering of a medium sized cylinder reveals that the respective region areas only cover a fraction of the total surface area.

The implemented heuristic:

Algorithm 3.7 Heuristic to determine min- and maxRadius

Calculate region area by adding the area of all its faces

if region size > 100 **then**

relation = regionSize / (regionArea * 100)
estimatedRadius = (regionArea * relation) / 2π
minRadius = max(estimatedRadius - 1, 0.1)
maxRadius = estimatedRadius + 3
else
// Values for very small regions got determined empirically
minRadius = 0.1
maxRadius = 2.5
end if

Note, the relation value was determined empirically as a useful modification factor.

To summarize, the heuristics are very straightforward and simplistic in their nature to keep the computational effort at a minimum level. The **distanceThreshold** is expanded or decreased from a base value depening on the region size. For the **Radius** the surface area of the entire region is considered in calculating a min- and maxRadius value. All the remaining parameters like MaxIterations, NormalDistanceWeight etc stay untouched. They are read in from a file and have to be set by hand.

3.5 Combination of Region Growing and RANSAC

After the required functionality for the previously described general outline of the approach was added to the Las Vegas framework, it was time to bring everything together. For this, a new method was introduced to the HalfEdgeMesh class that would run the respective functions in the right order and ensure the necessary requirements for each step. The method was called segmentShapes and got implemented in the following way:

Algorithm 3.8 Combining plane extraction and RANSAC estimator for cylinders

```
1: function segmentShapes
2:
3:         // optional preprocessing with existing plane opimization
4:
5:     label ← plane
6:     angle ← plane normal threshold
7:     specificClustering(label, angle, minRegionSize)
8:
9:     for all regions do
10:        if region label ≠ plane then
11:            Mark faces as not visited
12:        end if
13:    end for
14:
15:    label ← other
16:    angle ← low normal threshold
17:    specificClustering(label, angle, minRegionSize)
18:
19:    for all regions do
20:        if region label ≠ plane then
21:
22:            for all faces of region do
23:                normals ← face normal
24:                points ← face centroid
25:            end for
26:
27:            Initalize PCL input object with normals and points
28:            inliers ← 0
29:            parameters ← heuristics
30:
31:            Initialize Cylinder model with PCL input object and parameters
32:            inliers ← estimateModel
33:
34:            p ← inliers / region size
35:            if p >= acceptance threshold for cylinders  then
36:                region label ← cylinder
37:            end if
38:            normals ← empty
39:            points ← empty
40:
41:        end if
42:    end for
43: end function
```

3.6 Approach Summary

The overall principle of the presented approach follows a very generic paradigm for shape detection where simple shapes are extracted before complex ones. Consequently, planes are detected first by applying Region Growing with a high normal threshold. Afterwards, all the faces not belonging to planes are reclustered and get labeled as “other” regions. For this, the Region Growing algorithm is used with a low normal threshold. In the next step, the freshly clustered regions are evaluated by simple heuristics to determine better parmeters. This is done to reduce incorrect fitting for the implemented RANSAC estimator. At the same time, the input for the estimator is build and applied to it once the heuristic values have been obtained. Finally, the inlier percentage of the cylinder model is checked against an acceptance threshold to dermined whether or not a cylinder had been found. Depending on the results, a region is labeled as “cylindric“ or keeps the “other” flag.

Chapter 4

Testing and Results

The best way to evaluate any algorithm that is supposed to have a practical application is- and always will be real life data. Aside from this simple truth, there are plenty of complex and versatile sets freely available throughout the scientific community. Many of these sets cover a great variety of real life sceneries and range from simple objects to very large and complex exterior views. The fact that they are coming from independent sources and depict real life actualities makes them a great testing ground to benchmark the robustness and efficiency of the approach. However, the problem with these datasets is that they not always include enough different cylindric objects that are of interest for this work and to ensure a correct evaluation of the resulting cylinder detection one has to count cylinders in the scenery by hand.

This is why a small tool was written to create control datasets for early testing. The point clouds generated by this tool provided simple example sceneries with planes, cylinders and spheres and were used as reference datatsets in the initial stages of programming. However, the tool never reached a development stage where it included artificial noise so this has to be considered while reviewing the results.

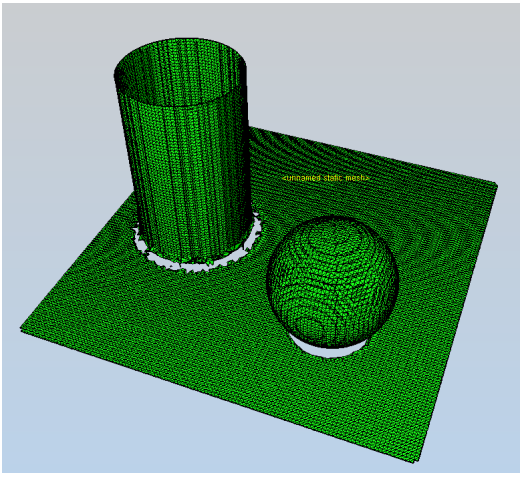


Figure 4.1: Example of a simple artifical dataset used for early testing.

Unless exceptions are stated, the following parameters were fixed for every trial:

- Coarse clustering threshold: 0.85
- minRegionSize for planes: 10
- minRegionSize for others: 10
- Base value DistanceThreshold: 0.25
- Base value minRadius: 0.1
- Base value maxRadius: 10.0
- AcceptanceThreshold for Cylinders: 0.8 (80%)

The following datasets where used for testing:

- Artifical dataset (noiseless control) - with 1.055.574 points
- The scan of a Byzantine church - with 2.149.565 points
- The scan of a medieval castle - with 5.281.343 points

And, finally, the machine used for testing and time measurements was a Intel® Core™2 Quad CPU Q9450 2.66GHz x 4 with 3,8 GB RAM.

Note, for finding planes, the Region Growing algorithm was run with 3 iterations of the currently implemented optimization techniques. That includes: flicker removal, small region deletion and fitting planes into a regression plane. Details about the these methods can be found in the PhD thesis of Thomas Wiemann [15].

4.1 Artifical Dataset

Changed parameters (accomodating noiseless condition):

- Plane normal threshold: 0.999999344
- minSize for planes: 100
- Coarse clustering threshol: 0.75 instead of 0.85

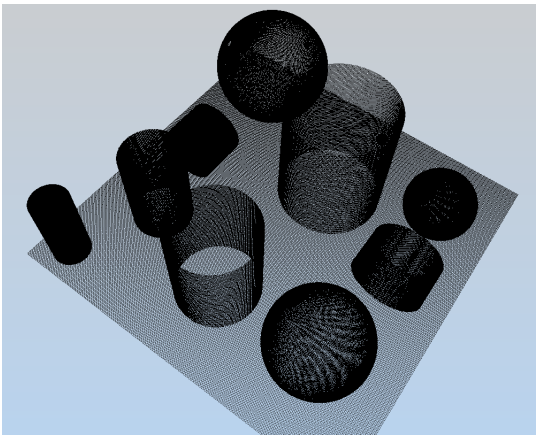
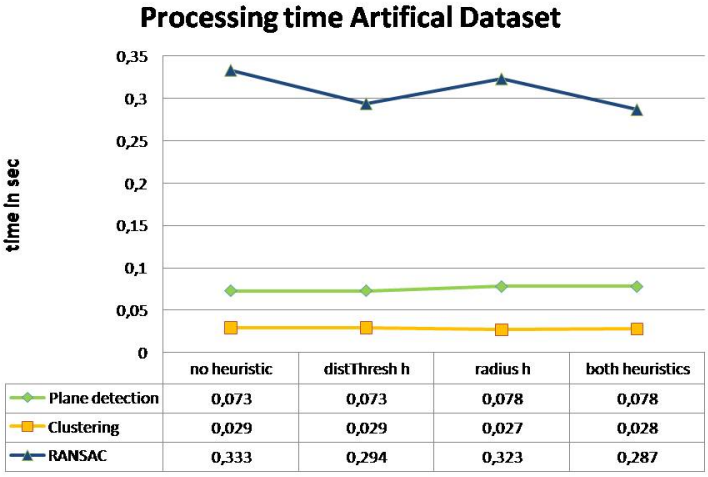


Figure 4.2: Point cloud of the data set



	no heuristic	distThresh h	radius h	both heuristics
Plane detection	0,073	0,073	0,078	0,078
Clustering	0,029	0,029	0,027	0,028
RANSAC	0,333	0,294	0,323	0,287

Figure 4.3: Time to process

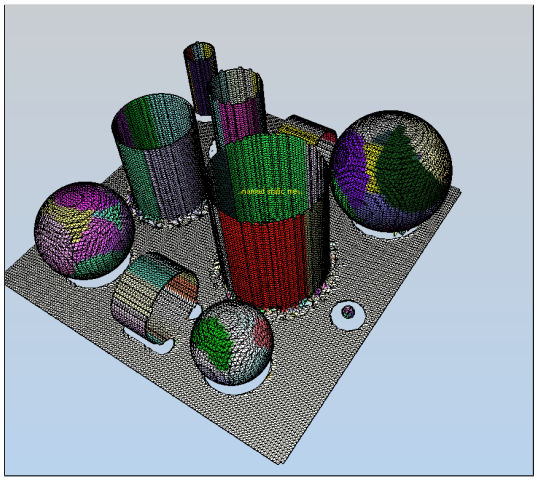


Figure 4.4: Coarse clustering

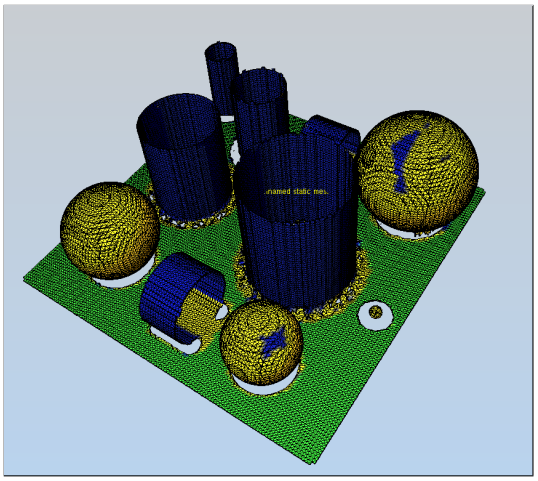


Figure 4.5: No heuristics

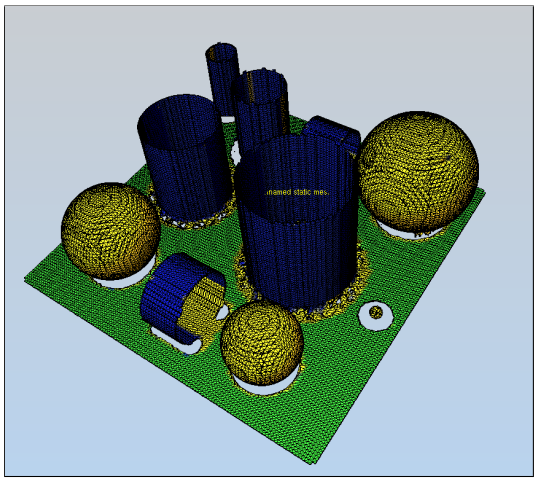


Figure 4.6: DistTreshold heuristic only

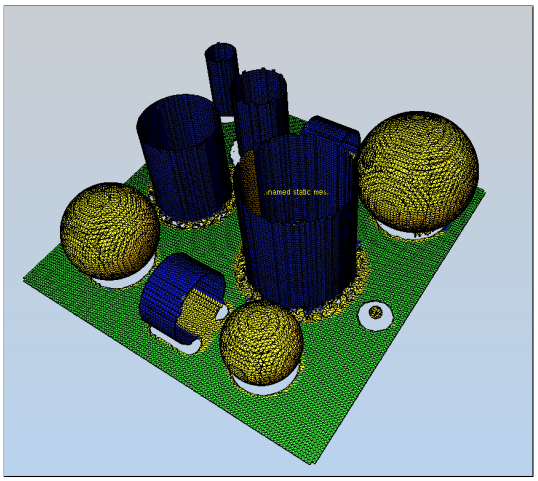


Figure 4.7: Radius heuristic only

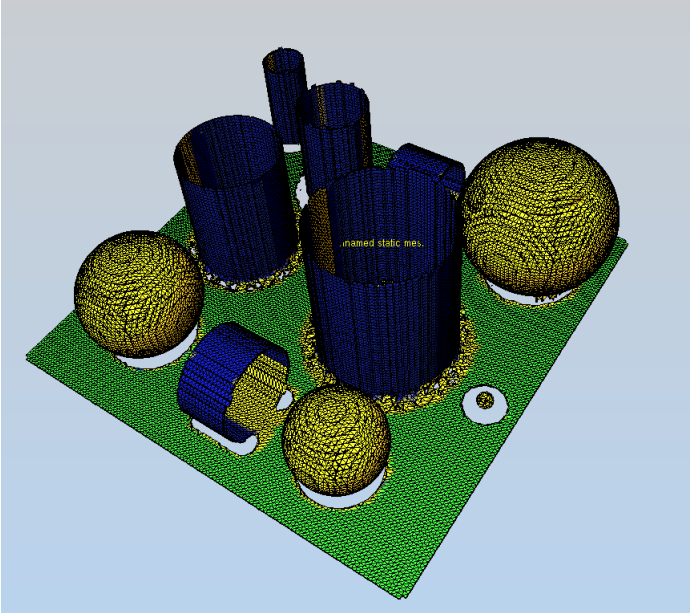


Figure 4.8: Both heuristics.

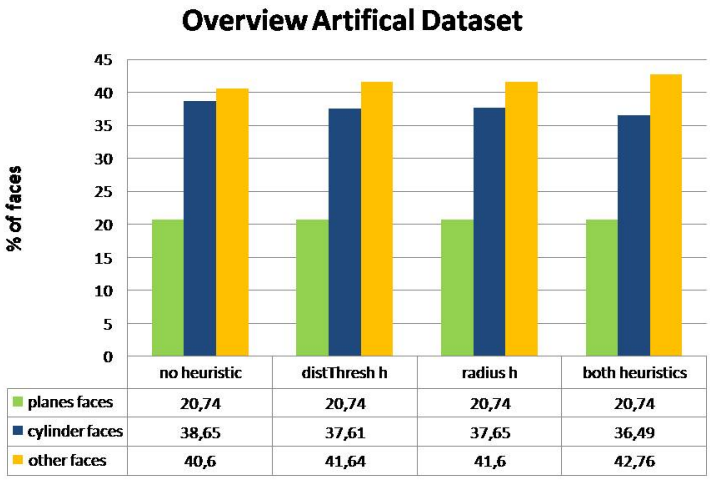


Figure 4.9: The percentages of the detected shapes

4.2 Church Dataset

Changed parameters:

- Plane normal threshold: 0.999933
- minSize for planes: 10

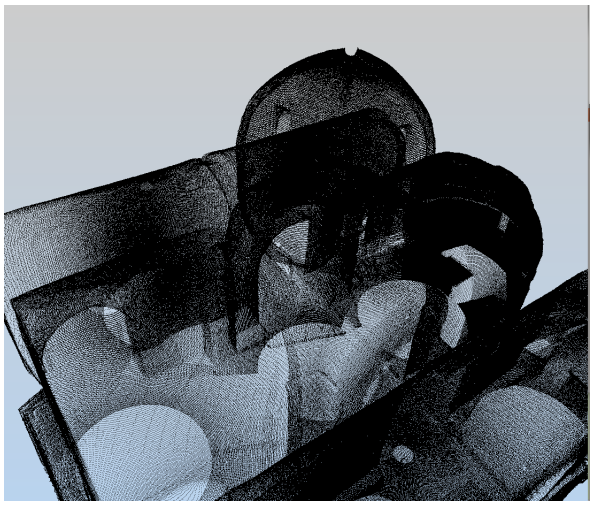
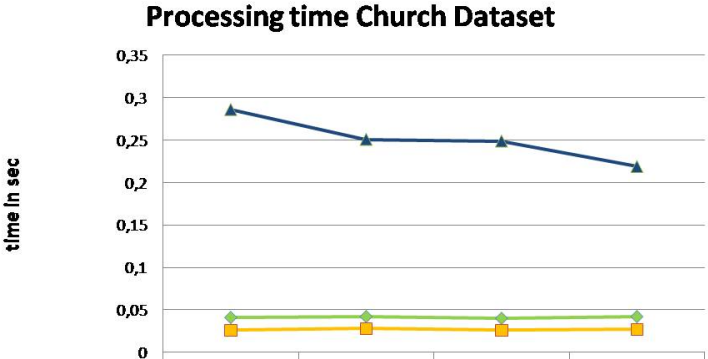


Figure 4.10: Point cloud of the data set



	no heuristic	distThresh h	radius h	both heuristics
Plane detection	0,041	0,042	0,04	0,042
Clustering	0,026	0,028	0,026	0,027
RANSAC	0,286	0,251	0,249	0,219

Figure 4.11: Time to process

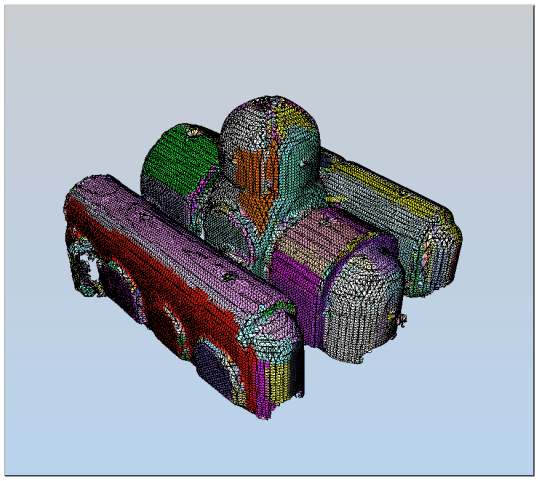


Figure 4.12: Coarse clustering

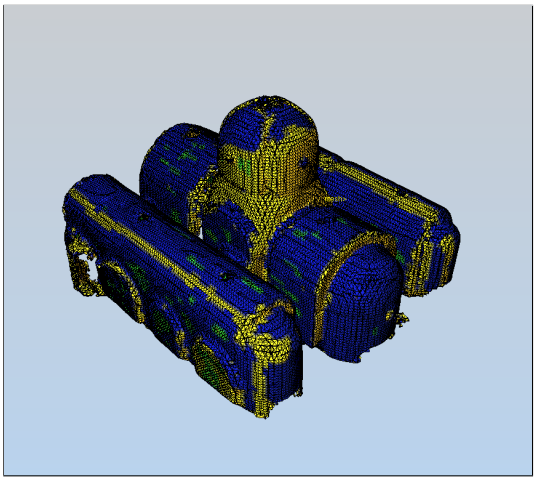


Figure 4.13: No heuristics

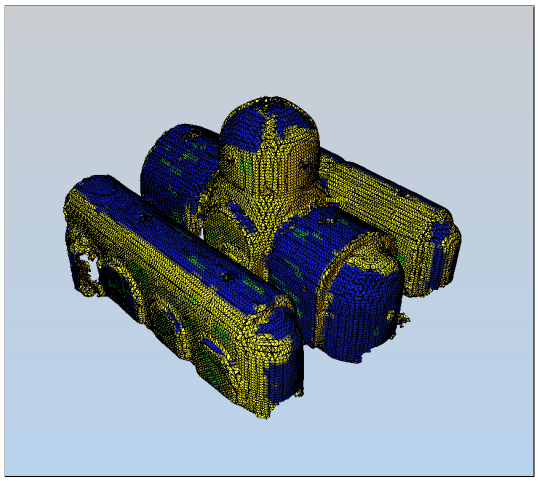


Figure 4.14: DistTreshold heuristic only

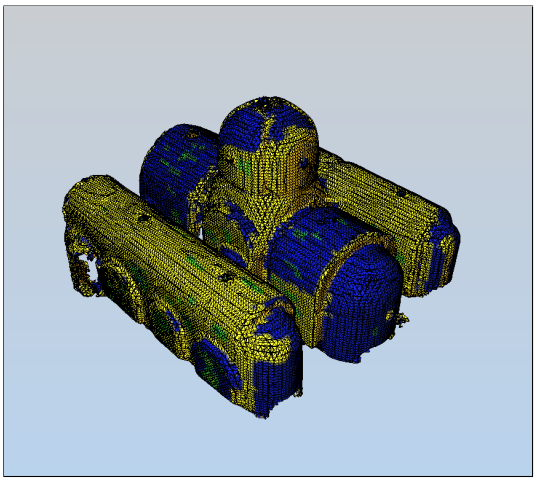


Figure 4.15: Radius heuristic only

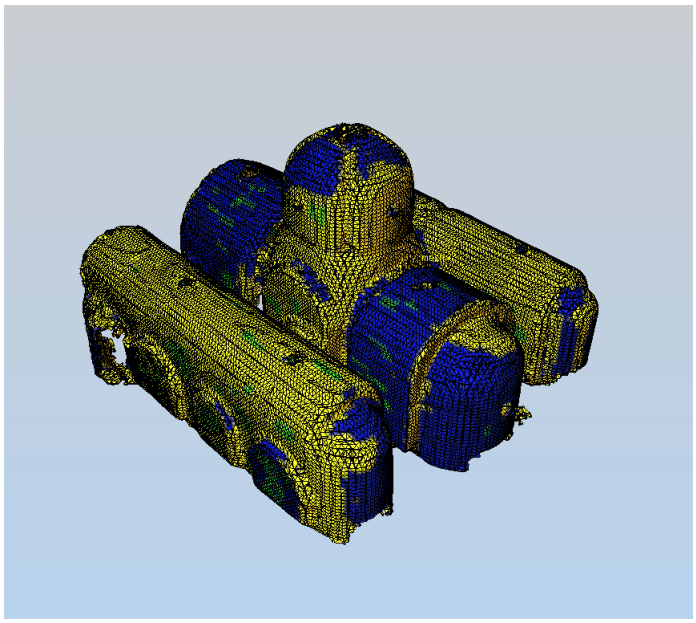


Figure 4.16: Both heuristics.

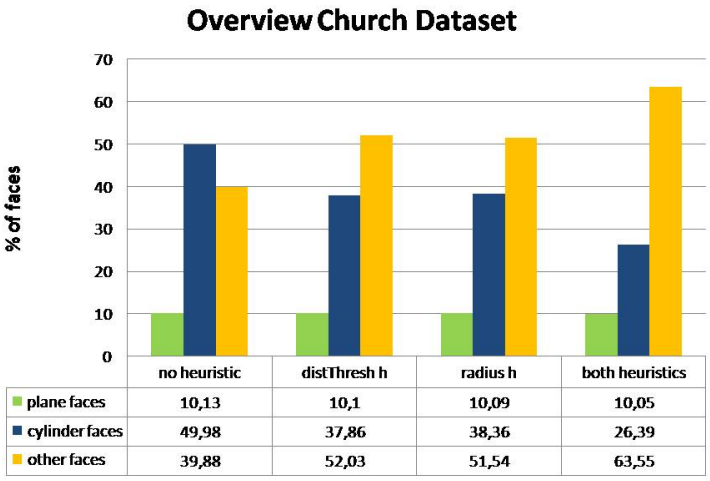


Figure 4.17: The percentages of the detected shapes

4.3 Castle Dataset

Changed parameters:

- Plane normal threshold: 0.9976
- minSize for planes: 250
- Coarse clustering threshol: 0.75 instead of 0.85 (accomodating noiseless condition)

Figure 4.18: Point cloud of the data set

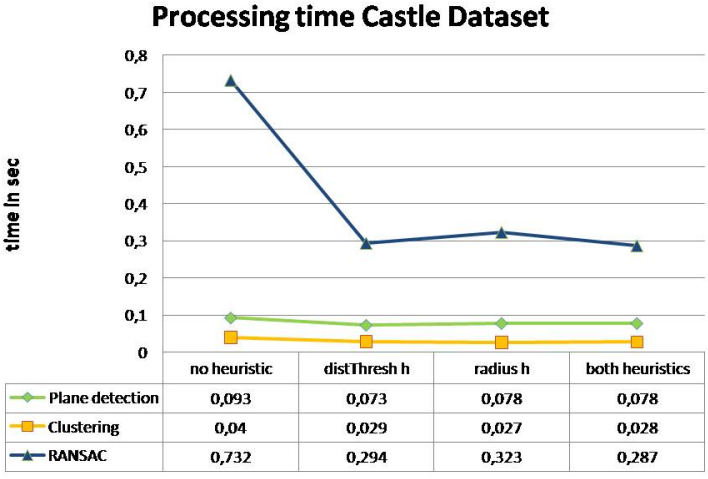


Figure 4.19: Time to process

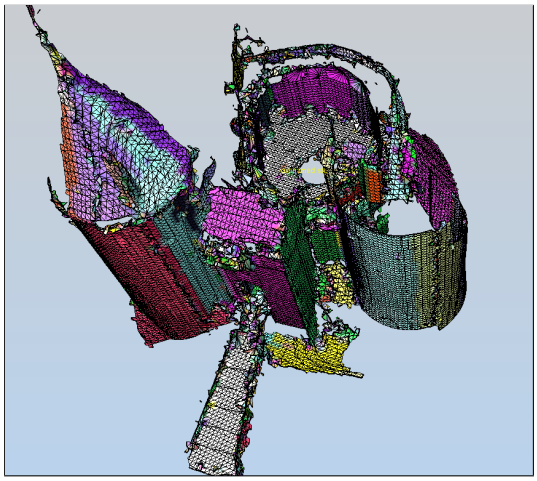


Figure 4.20: Coarse clustering

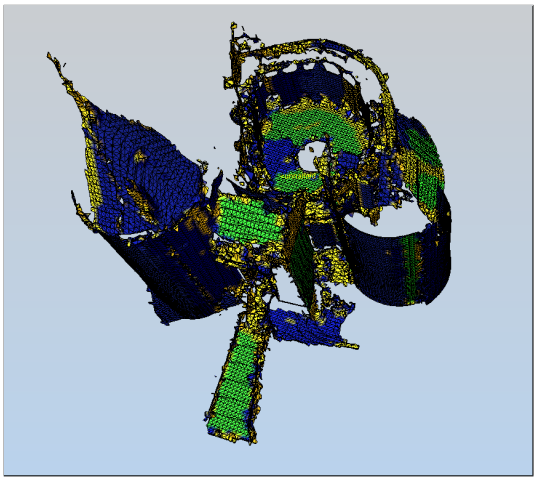


Figure 4.21: No heuristics

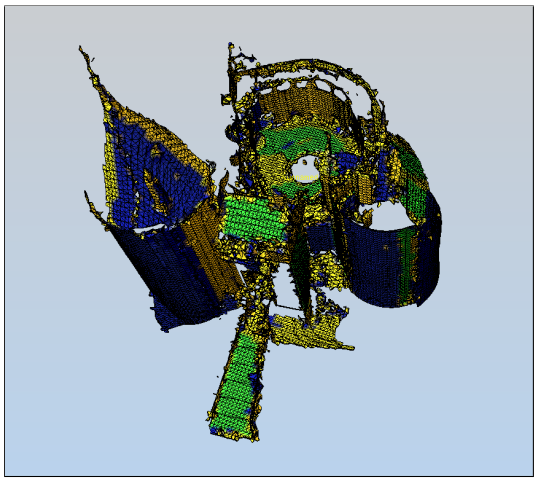


Figure 4.22: DistTreshold heuristic only

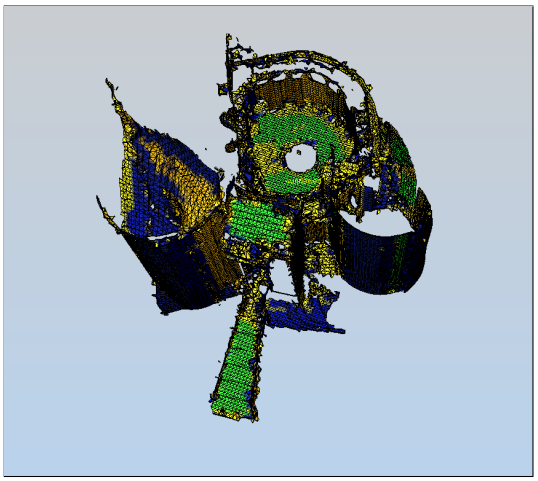


Figure 4.23: Radius heuristic only

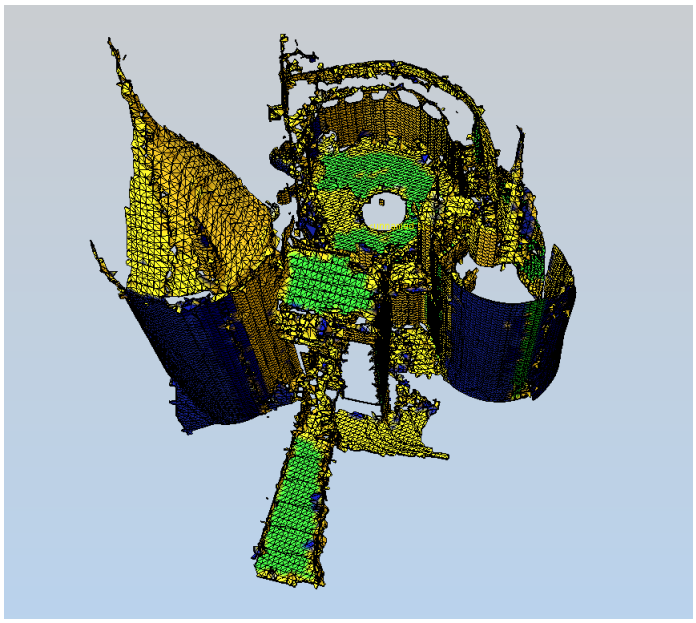
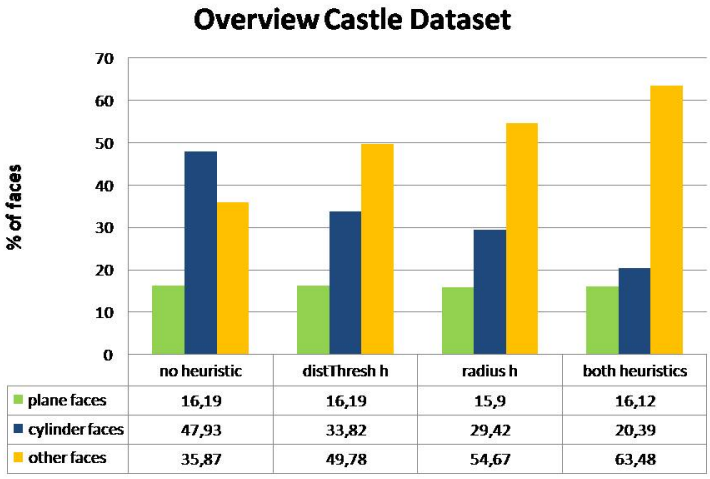


Figure 4.24: Both heuristics.



	no heuristic	distThresh h	radius h	both heuristics
plane faces	16,19	16,19	15,9	16,12
cylinder faces	47,93	33,82	29,42	20,39
other faces	35,87	49,78	54,67	63,48

Figure 4.25: The percentages of the detected shapes

4.4 Level of Reduction

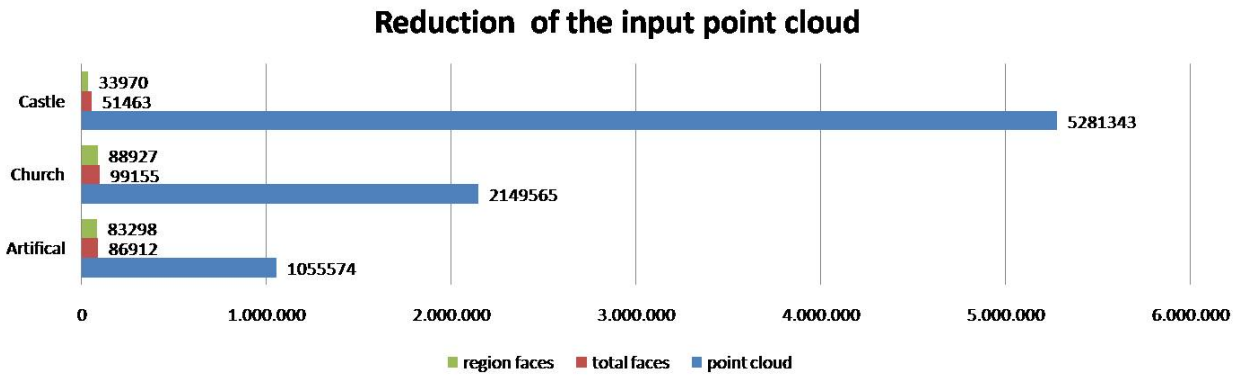


Figure 4.26: Reconstruction of the polygon meshes from the input point cloud lead to a considerable level of reduction.

4.5 Reproducabilty

In order to provide the means for a reproduction of the results, the used datasets with extensive documentation of all the used software parameters will be made available on the Las Vegas website [16]. They can also be obtained by writing an email to: shennig@uos.de.

The package will furthermore include a complete list of all the external libraries and their respective versions. It also contains the parameters used for the reconstruction software.

Chapter 5

Discussion

5.1 Reviewing the Results

The purpose of this work was to add fast and reliable cylinder detection to the Las Vegas reconstruction toolkit. This was achieved by adapting a RANSAC estimator intended for point clouds and, as the results show, the current implementation became quite fast, despite the fact, that the mesh information had to be translated into a new point cloud object. The speed of the detection also got increased by the added heuristics although this wasn't even their intended purpose. But it makes sense considering their role in enforcing limits for the possible cylinder model. However, originally these heuristics got included to resolve the issues of a fixed set of paramters. Namely, to decrease the overall level of over- and underfitting. While they seem to work reasonably well in case of the castle dataset they clearly need more refining as seen in the church dataset. It is also noteworthy that the castle was the most reduced dataset (from 5 million points to 50k). This suggests that some sort of scale for the level of reduction might come in handy if the heuristics were to be improved and that the current ones work best on strongly reduced data. More concretely, improving upon this idea would mean having different sets of heuristics for small, medium and large original datasets that factor in the reduction. A drawback of the heuristics is that they sacrifice a certain degree of precision in order to rule out false positives. This can be seen in all datasets where some regions of obvious cylinders are not detected as such. Still, if for example the information about cylinders is used in the optimization of the mesh, it is better to polish a few correctly classified regions rather than to include incorrect fits.

Another factor in all this is the construction of the model input. Since it was constructed with the centroid points and the face normals there is a good chance that too much information was lost overall. Thus, by using the coordinates of the three face vertices instead, the precision of the RANSAC estimator could be increased. However, there was no previous experience with any results of cylinder detection and the whole functionality had to be implemented from scratch. So, at the time, it seemed a reasonable idea to go with the less complex input first as there is always the option to add more information later on. Now that there are established basic results, these can be used to compare the product of refinements and alternative approaches.

5.2 Sketching an alternative Approach

In the inital stages of this work a small tool, coined the "datacraft-kit", was used to construct reference datasets to provide the means for an evaluation of the first results. Meaning, it was perfectly clear how many cylinders were placed in the data and had to be detected. If this tool was to be improved into a more sophisticated version where adjustable levels of artifical noise and more customized models of cylinders or other shape primitives were to be included, it would open the door for machine learning approaches.

Machine learning techniques like **S**upport **V**ector **M**achines [4] (short SVM) can be used as classifiers after a supervised learning process. In this respect, the datacraft-kit could be used to build an appropriate training environment. Also, programming the tool rather than looking for an existing solution would have an advantageous side effect. The algorithms used for constructing the shape primitives in artifical point clouds could probably be recycled to provide replacement models in mesh optimization as well. But to state the idea behind using SVMs more precisely, another requirement for this to work would be a way to discretize the information stored in faces and regions. During the course of this work, Thomas Wiemann suggested to build some sort of histogram representing this information. So, assuming a good function with the following properties was established.

$$f\colon \mathbb{R}^3 \to \mathbb{R}, (x, y, z) \mapsto value \tag{5.1}$$

The vertex, face and normal coordinates could be used to construct the bars for the histogram in addition to any number of extra information. The next step would be to create such histograms for different types of cylinders and train the SVM classifier with it. Afterwards testing would commence on appropriate datasets.

Aside from SVMs, there is another perceivable alternative. Provided that the heuristics of the presented approach are refined, it might be possible to combine these with a set of learned parameters (all of those with no heuristic). The ideal setting of these parameters could be found by applying evolutionary algorithms [1]. That is to say, after building a noisy training set with n cylinders, the modified approach would be applied again and again. Each time changing only one parameter and evaluating whether or not the new value produces better results by counting the number cylindric faces.

5.3 Conclusion

The presented approach succeeds in providing a rough but fast detection of cylinders. At the same time, the described heuristics show promise but come at the cost of precison and require further refinement. In terms of added functionality of Las Vegas toolkit, the basis is laid out to extract additional shapes as well. For example for models of cones, tori or spheres this is simply a matter of using the respective classes with the implemented RANSAC estimator. At last, considering the question of what to do with the extracted information. The in- and outliers of the found cylinders could be used to optimize the mesh quality. This could be done by moving

the faces into the position of a cylinder model. In this regard, the earlier described datacraft tool might come in handy for constructing such models.

Bibliography

[1] D. Ashlock. *Evolutionary computation for modeling and optimization*, volume 200. Springer, 2005.

[2] P. Azad, T. Gockel, and R. Dillmann. *Computer Vision: principles and practice*. Elektor Electronics Publishing, 2008.

[3] T. Chaperon, F. Goulette, and C. Laurgeau. Extracting cylinders in full 3d data using a random sampling method and the gaussian image. In *Proceedings of the Vision Modeling and Visualization Conference*, pages 35–42, 2001.

[4] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[5] M.A. Fischler and R.C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[6] Z.C. Marton, L. Goron, R. Rusu, and M. Beetz. Reconstruction and verification of 3D object models for grasping. *Robotics Research*, pages 315–328, 2011.

[7] PointCloudLibrary. Pcl cylinder model segmentation documentation. `http://pointclouds.org/documentation/tutorials/cylinder_segmentation.php#cylinder-segmentation`, October 2012.

[8] PointCloudLibrary. Pcl random sample consensus model documentation. `http://pointclouds.org/documentation/tutorials/random_sample_consensus.php#random-sample-consensus`, October 2012.

[9] T. Rabbani and F. Van Den Heuvel. Efficient hough transform for automatic detection of cylinders in point clouds. *ISPRS WG III/3, III/4*, 3:60–65, 2005.

[10] A. Richtsfeld, T. Mörwald, M. Zillich, and M. Vincze. Taking in shape: Detection and tracking of basic 3d shapes in a robotics context. In *Computer Vision Winder Workshop*, pages 91–98, 2010.

[11] Radu Bogdan Rusu. Pcl random sample consensus cylinder model documentation. `http://docs.pointclouds.org/trunk/classpcl_1_1_sample_consensus_model_cylinder.html`, October 2012.

[12] Rusu, R.B. Semantic 3d object maps for everyday manipulation in human living environments. *KI-Künstliche Intelligenz*, 24(4):345–348, 2010.

[13] Schnabel, R. and Wahl, R. and Klein, R. Efficient RANSAC for Point-Cloud Shape Detection. In *Computer Graphics Forum*, volume 26, pages 214–226. Wiley Online Library, 2007.

[14] Ulrich, M. and Wiedemann, C. and Steger, C. Cad-based recognition of 3d objects in monocular images. In *International Conference on Robotics and Automation*, pages 1191–1198, 2009.

[15] Thomas Wiemann. Automatische Generierung dreidimensionaler Polygonkarten für mobile Roboter. 2012.

[16] Thomas Wiemann. Las Vegas Reconstruction Toolkit. `http://www.las-vegas.uni-osnabrueck.de/`, October 2012.

[17] Wikipedia. Wikipedia random sample consensus example. `http://en.wikipedia.org/wiki/RANSAC`, October 2012.

[18] Wikipedia. Wikipedia random sample consensus example ii. `http://de.wikipedia.org/wiki/RANSAC-Algorithmus#Maximaler_Abstand_eines_Datenpunkts_vom_Modell`, October 2012.

Proclamation

Hereby I confirm that I wrote this thesis independently and that I have not made use of any other resources or means than those indicated.

Osnabrück, October 2012