UNIVERSITY OF TARTU

INSTITUTE OF COMPUTER SCIENCE

Computer Science

Sander Tiganik

Rough estimation of interior dimensions using structure from motion techniques

Master's thesis (30 EAP)

Supervisor: Artjom Lind

TARTU 2017

Rough estimation of interior dimensions using structure from motion techniques

Abstract:
Nowadays structure from motion algorithms have become accurate enough to compete with laser scanner accuracy, however most of the algorithms require points of interest and textured surfaces in order to give better results. Most algorithms will have poor performance when it comes to monotonically coloured or textureless surfaces. Furthermore, the output of the algorithms will have gaps in the projection of the structure it is trying to recreate. This kind of projection would be useless in a case where consistency and completeness of surfaces is more important than the level of detail. In this thesis the author will try to use structure from motion techniques and new ideas to create a projection of an interior room which focuses on the essence of the room (I.e aspect ratio, correct floor plan) rather than on the level of detail of objects in the room. The goal of this thesis will be to create an algorithm which can generate a projection out of a sparse point cloud (result of SfM) that is consistent enough to allow it to be used for applications that require a more complete model rather than a detailed one (I.e robot pathfinding, indoor people tracking).

Keywords: Computer vision, Structure from motion, Scene reconstruction, Face extraction

CERCS: P170

Ligikaudne siseruumi mõõtmete hindamine kasutades liikumisest struktuuri eraldamise võtteid

Lühikokkuvõte:
Tänapäeva uusimaid võtteid kasutades on võimalik liikumisest piltide peal eraldada struktuure, mille täpsust võib võrrelda laser skännerite täpsusega. Kahjuks vajavad need algoritmid siiski huvipunkte ning tekstuurseid pindasid, et anda parimaid tulemusi. Enamus algoritmidel on halb jõudlus kui nad peavad töötlema monotoonsete värvidega või tekstuurivaeseid pindu. Lisaks

on tihti algoritmide tulemustes auke, mida algses stseenis ei esine. Selline projektsioon ei ole väga kasulik ega kasutatav juhul kui on stseenist oodatakse järjepidavust ja täielikkust, mitte detailide täpsust. Selles töös proovib autor kasutada liikumisest struktuuri eraldamise võtteid koos uudsete ideedega, selleks et luua projektsioon siseruumist fookusega toa olemusele (õiged pikkuste suhted, õige põranda plaan), mitte objektide detailsusele. Selle töö eesmärgiks on välja pakkuda algoritm, mis suudab taasluua stseeni, mida oleks võimalik kasutada rakendustes, kus rõhk on stseeni täielikkusel, mitte detailsusel (Nt. Õige tee leidmise probleemid).

Võtmesõnad: Arvuti nägemine, Struktuur liikumisest, Stseeni taasloomine, Tahkude tuvastamine

CERCS: P170

Contents

1 Introduction **6**

1.1 Introduction into structure from motion 6

1.2 Goals and research questions 6

1.3 Thesis structure 7

2 Related work **8**

2.1 Pointcloud creation 8

2.1.1 Feature detection 9

2.1.2 Feature matching 13

2.1.3 Two view epipolar geometry 13

2.1.4 Fundamental matrix 14

2.1.5 Essential matrix 15

2.1.6 3-space point triangulation 16

2.1.7 N-view reconstruction 17

2.2 Scene reconstruction 19

2.2.1 Manhattan world-stereo 19

2.3 Summary 21

3 Hypothesis **23**

3.1 Idea 23

3.2 Method 24

3.2.1 Extracting dominant axes 24

3.2.2 Assigning points to axes 25

3.2.3 Reducing point cloud size 26

3.2.4 Assigning points to planes 26

3.2.5 Reconstructing faces 27

3.2.6 Ensuring structural completeness 29

3.3 Summary 30

4 Implementation **31**

4.1 Structure storage format 31

4.2 Extract dominant axes 33

4.3 Assigning points to axes 35

4.4 Reducing point cloud size 36

4.5 Assigning points to planes 36

4.6 Reconstructing faces 37

4.7 Ensuring structural completness 38

4.8 Summary 40

5 Results **41**

6 Conclusion **45**

7 Future work **46**

1 Introduction

1.1 Introduction into structure from motion

Structure from motion is a technique used in computer vision and visual perception to use local motion signals in order to gain information about a scenes structure using no other data than camera images. This allows one to rebuild a three dimensional scene using only mono camera photos.
The reconstruction process is divided into two parts. The first part is the calculation of a pointcloud. A pointcloud, like the name suggests, is a collection of points. The points can be calculated using the mono camera images and epipolar geometry. The second part of the reconstruction is the face extraction. It is the algorithm used to convert the point collection into a three dimensional mesh. There are many algorithms to achieve it, but it still remains a complicated problem especially in case of pointclouds with high levels on entropy (noise).
In this thesis we will focus on the second half of the reconstruction process. As a basis for our new algorithm we will use an existing idea called Manhattan world stereo, which is an algorithm used to reconstruct highrise buildings from pointclouds.

1.2 Goals and research questions

The goal of this thesis is to design an algorithm that could be used to create a rough esitmation of an interior room from a pointcloud. The estimation of the room has to be complete in the sense that, unlike many algorithms, the reconstruction may not have neither illogical holes nor disconnected faces in the final mesh. The estimation may replace complex details from the scene with simpler structures like cubes and cylinders, but the dimensions and sizes of the room have to remain unaltered.
Next we will state the main research questions that this thesis will try to address:

- How to define an algorithm that reconstructs an interior room, without altering its dimensions.
- How to ensure that the algorithm connects up all the faces without leaving gaps.

- Can the above mentioned be achieved without using any further information than the provided point cloud.

1.3 Thesis structure

The thesis is divided into six chapters. The first chapter titled "Related work" focuses on prior work and state of the art in structure from motion techniques. It explains the complex process of extracting motion signals from monocular camera (hereinafter mono camera) pictures in order to calculate pointcloud points in 3-dimensional space (hereinafter 3-space). The chapter also introduces relevant face extraction algorithms that are similar to what we are trying to achieve in this thesis and that we might use as a basis for our new face extraction algorithm.
The second chapter titled "Hypothesis" talks about the algorithm we will try to develop and gives insight into what the algorithm does and what its capabilities should be. Also it explains the methods that will be used to achieve the goals we have set in this thesis.
The third chapter called "Implementation" deals with the practical application constructed during the writing of this thesis. In this chapter all details concerning the implementation are explained in depth, starting from pointcloud and mesh storage formats all the way till the face extraction algorithm itself.
The fourth chapther titled "Results" covers the results generated by the newly developed algorithm. We compare the performance and output of our idea to other algorithms, and make an assessment whether the new algorithm suggests an improvement compared to existing ones. Also we will assess if the goals we set in the beginning have been achieved.
The fifth chapther named "Conclusion" wraps up the topic of scene reconstruction and notes all the work done in this thesis. In this chapter we will also estimate whether we succeeded in answering the main research questions stated in the introduction chapter 1.2.
In the sixth and final chapter titled "Future work" we will delve into the possibilities and ideas that were not researched in this thesis and that might be worth looking into in the future. The author will try to give as much information about shortcomings and improvements as possible so that future researchers may have it easier to continue from where he left off.

2 Related work

In this chapter we will explain in depth everything one needs to know about mono camera image scene reconstruction. We will also break down and explain the current state of the art in the field of structure from motion.

2.1 Pointcloud creation

The first part of any scene reconstruction is always the extraction of a point cloud from a collection of images. Point clouds can be extracted from a wide variety of image collections, taken by cameras with different setups. Different camera types include Mono cameras (one lens), Stereo cameras (multiple lenses with fixed positions) and even cameras with RGB-d and LIDAR (Light Detection And Ranging) sensors. In this thesis we will only focus on Mono cameras, as they are the most common type of cameras that most people will have at home.
Point cloud extraction from images has a well established pipeline of steps that need to be done in order to calculate the point cloud. The steps are:

- Feature detection - In this step all the images in the collection are processed using an algorithm, which tries to find points of interest on the picture (Depending on the algorithm, points of interest may be edges, corners, points that differ from their surroundings etc.)[fea].

- Feature matching - Trying to compare features on different images with each other, to see whether a feature on one image might correspond to a feature on another image [fea].

- Motion matrix calculation - Calculating the Fundamental and Essential matrix which describes camera motion between two images (The first gives us a projective and latter a similarity transformation) [RH04].

- Point triangulation - Using the motion matrix and features of two images and calculating a point in 3-space [RH04].

- Adding more views - Increasing the point count in the point cloud, by adding additional image features and triangulating more points [RH04].

Using this pipeline a point cloud depicting a scene from n views can be extracted from a collection of images.

2.1.1 Feature detection

Feature detection is the process of determining which points on a set of data are the points of interest. In structure from motion the data is a collection of images and the points of interest are certain pixels of the images that stand out or differ from all other points. A feature descriptor is a collection of information for each point of interest. They not only contain a specific point (pixel), but also information to reliably indentify the point later on other images, that may have different scale, noise and illumination [Low99]. Descriptors usually lie on high-contrast regions of an image like edges, corners, ridges and blobs. In this chapter we will focus on a good algorithm that deals with feature descriptors called SIFT (Scale-invariant feature transform)
SIFT is a local feature detection and description algorithm developed by David Lowe in 1999. It is very capable as it can detect objects among clutter and partial occlusion. SIFT is a significant advancement in computer vision, becuase it promises invariance to image translation, scaling and rotation [Low99]. It also claims to have partial invariance to illumination changes and affine or 3D projection [Low99]. The SIFT algorithm consists of 6 steps:

- Constructing a scale space [Wit83]
- Laplacian of Gaussian approximation
- Finding keypoints
- Reducing keypoints
- Calculating keypoint orientations
- Creating SIFT features

The algorithm starts with the creation of a scale space. In computer vision scale space is a set of smoothed images with progressively less and less details on them. Scale space is achieved by using Gaussian Blur, which has the mathematical property of removing details without adding new false details. The idea behind scale space is to represent an image structure at different scales [Low99]. The SIFT algorithm takes scale space a bit further. Instead of having only a set of ever more blurry images, SIFT generates a set of sets of images. Each of this sets is called an octave. In each octave the size of the picture is halved, compared to the last octave. This way we end up with a scale space with n octaves, where the first octave is k images of original size

that are gradually more blurred. The second octave is k images with half the size of the last octave, that are gradually more blurred etc. (illustrated by figure 1) [Low99].

Figure 1: SIFT algorithm scale space [sif]

The second part of the algorithm is the application of the Laplacian of Gaussian (LoG) operation on the created scale space [sif]. LoG is an operation that allows one to locate edges and corners on an image. This is achieved by taking the second order derivate on the image. Since the second order derivate is very sensitive to noise a blurring (Gaussian) operation is also required to stabilize the derivate. Unfortunately the second order derivate is computationally intensive, but fortunately there is a shortcut. One can use Difference of Gaussian (DoG) for the approximation of LoG. DoG works by subtracting two images with different levels of blurring from each other, giving you an approximately equivalent result as LoG. Using this information we can calculate the difference of consecutive scales of all octaves in our scale space. (Illustrated in figure 2) [Low99]
Next we will use the DoG images generated in the second part to find keypoints. First we have to find the maximum/minimum of the DoG images. This is achieved by looking at the pixel and its surrounding neighbours, both on the same scale and also on neighbouring scales. This means for each point in an image 26 checks are made in order to determine whether it is a maximum/minimum point. Nine checks on both the previous and next scale and eight checks around the point in question (illustrated in figure 3). After the

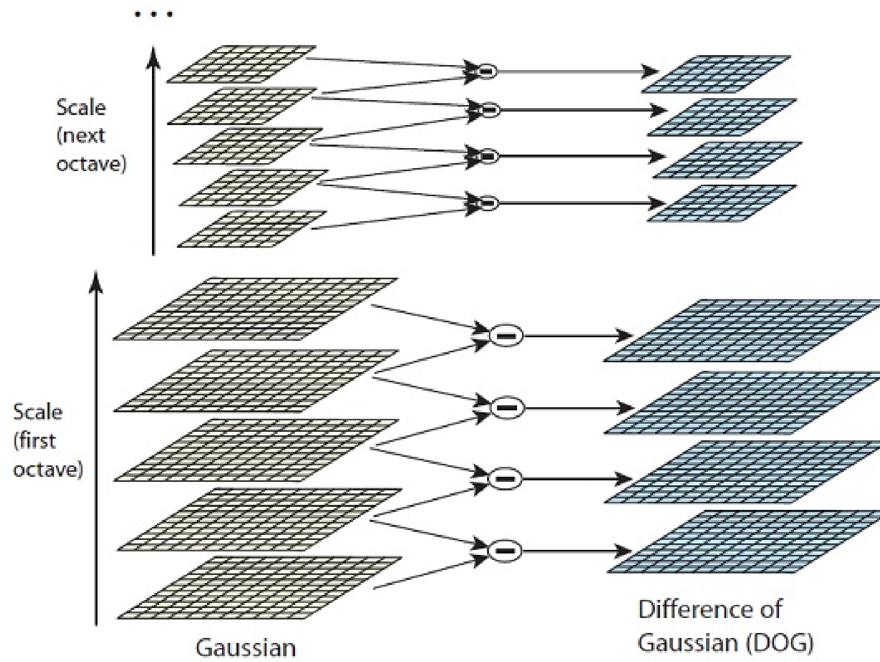

Figure 2: Example of Difference of Gaussian (DoG) [sif].

maximum/minimum points have been found the points have to be translated to pixel coordinates, because the points found almost never lie exactly on a pixel. [Low99, sif]

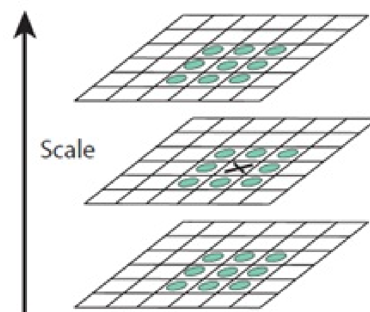

Figure 3: Finding minimum/maximum points [sif]

Once we have found our keypoints we need to start removing some of them. Not all of the points found are suitable as features, for instance low contrast areas or edges. Low contrast areas can be sorted out by checking the intensity value of each keypoint pixel. If the intensity is lower than some magnitude the point is rejected. With the low intensity points removed we can focus on removing all keypoints on edges and flat regions, which can be achieved by using a Hessian Matrix. [KB07]
Next an orientation needs to be assigned to the keypoints that have been

detected so far. Doing this gives us rotation invariance. The magnitude and orientation of all pixels around the keypoint are calculated (figure 4) and a histogram is created. If there are multiple dominant orientations the keypoint is split into n identical keypoints each with one of the dominant orientations found on the histogram. The size of the orientation information collection area is dependant on the scale of the keypoint. [Low99, sif]

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$
$$\theta(x,y) = \arctan((L(x,y+1) - L(x,y-1))/(L(x+1,y) - L(x-1,y)))$$

Figure 4: Magnitude and orientation formula [Low99, sif].

The last thing that needs to be done, is to calculate a fingerprint (feature), that allows us to later distinguish features from one another. One restricting obstacle about the feature is that it has to be relatively robust and fault tolerant, since nothing ever is exactly the same in images, especially lighting conditions and camera position. A feature fingerprint is a 128 value vector, consisting of sixteen 4x4 grids, where the keypoint is the center of this square grid. The 4x4 grids are filled much like in the last step using magnitude and orientation information. What is different from the last step is the usage of a gaussian weighting function. This function applies a gradient to the values of the fingerprint in such a manner, that the values closer to the keypoint have more weight than the ones further away (much like a flashlights light dissipates the further it gets from the focus point 5). The feature is also normalized once more to ensure rotation and illumination invariance. [Low99, sif]

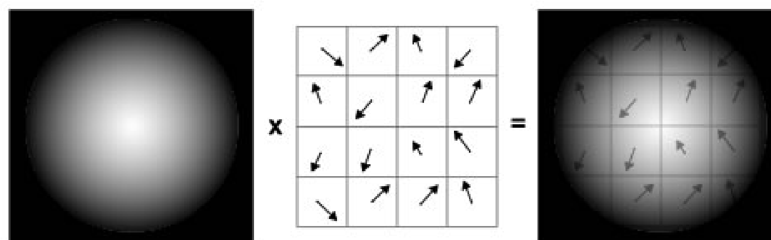

Figure 5: Gaussian weight function applied to fingerprint [sif].

After completing the six steps one ends up with a set of SIFT features, which are scale and rotation invariant and also partially illumination and occlusion invariant.

2.1.2 Feature matching

Feature matching is a probabilistic operation. No two images will ever feature the same object in the same position and lighting conditions, unless the same image is observed twice. The features are compared to each other and a probability is calculated to show the strength of the match between features. The higher the probability the liklier it is, that the two features are the same. There have been many algorithms suggested for feature matching, but usually the most common brute-force approach is good enough for most purposes. [fea]

2.1.3 Two view epipolar geometry

Epipolar geometry is the mathematical principal used in stereo vision. Epipolar geometry specifies the case that if two cameras are viewing the same scene from different positions, then there is a relation between the two 2D projections of the actual 3D points. [RH04]
Epipolar geometry works using coplanarity. If we take a point X in 3-space (3-dimensional space) and view this point in two distinct images, where in the first image the point is projected to an image point x and in the second image it is projected to an image point x', then the image points x and x', 3-space point X and camera centers of the two images are all coplanar (They lie on the same plane). This plane is called the epipolar plane in common literature and is usually denoted with the symbol π. When one connects the camera centers of the two images with each other, then one gets what is called the baseline. The baseline and the ray projected by either point x or x' defines the epipolar plane π. This means that there is a relation between the projected points x and x'. If we define the plane π using the baseline and the ray projected by the image point x, then the second image point x' has to be on a line l' which is back-projected from the ray from x onto the second image. This relation is useful to narrow down the search for a projected point from an entire image to just one line (the above is illustrated in figure 6). [RH04]
Next some terminology used in epipolar geometry:

1. The epipole is the point of intersection of the line joining the camera centres (the baseline) with the image plane [RH04].

2. An epipolar plane is a plane containing the baseline. There is a one-parameter family (a pencil) of epipolar planes [RH04].

3. An epipolar line is the intersection of an epipolar plane with the image plane [RH04].

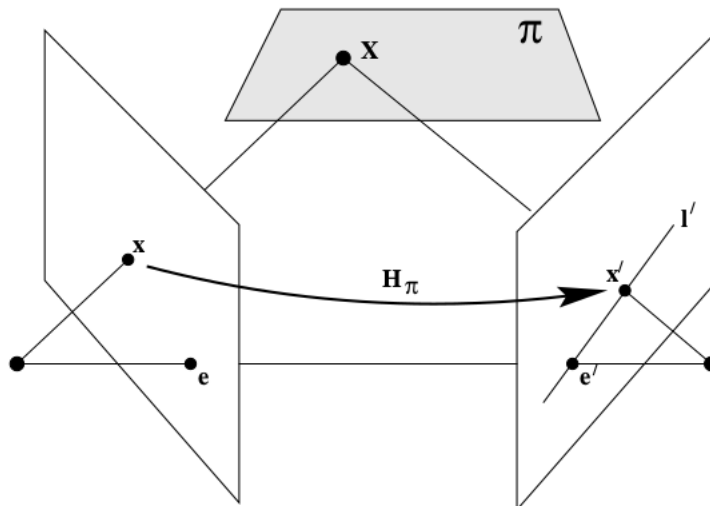

Figure 6: Two view epipolar geometry [RH04].

2.1.4 Fundamental matrix

The fundamental matrix is the algebraic representation of epipolar geometry. From the previous chapter we know that there is a mapping from a point x in one image to its corresponding epipolar line l' in the other image (figure 7).

$$x \mapsto l'$$

Figure 7: Mapping from image points to epipolar lines [RH04].

This is achieved in a two step process of first transferring the point via a plane and then constructing an epipolar line on the other image [RH04]. This process defines the most basic property of the fundamental matrix (figure 8).

$$x'^T F x = 0$$

Figure 8: Basic property of the fundamental matrix [Fau92, Har92, RH04].

This property means that if two image points x and x' correspond, then x' lies on the epipolar line $l' = Fx$ corresponding to the point x. It also works the

other way around. If two image points satisfy the relation $x'^T Fx = 0$, then the rays defined by these points are coplanar [RH04]. Next we will list a number of properties of the fundamental matrix:

1. The fundamental matrix of two images aquired by cameras with non-coincident centres is a unique $3x3$ rank 2 homogenous matrix which satisfies the relation $x'^T Fx = 0$ [Fau92, Har92, RH04].
2. If F is the fundamental matrix of the pair of cameras (P, P'), then F^T is the fundamental matrix of the pair in the opposite order: (P', P) [RH04].
3. For any point x in the first image, the corresponding epipolar line is $l' = Fx$ [Fau92, Har92, RH04].
4. For any point x (other than e) the epipolar line $l' = Fx$ contains the epipole e' [Fau92, Har92, RH04].
5. F is a projective map taking a point to a line [Fau92, Har92, RH04].

2.1.5 Essential matrix

The essential matrix is a specialization of the fundamental matrix. It encompasses normalized image coordinates. The essential matrix adds the assumption of calibrated cameras. The essential matrix has additional properties compared to the fundamental matrix.
Normalized coordinates are independant of camera parameters. When taking an image using a camera, there are several parameters that define how a world point X will be projected onto the image plane. This can be expressed as an equation $x = PX$ where P is a matrix describing the parameters for the camera used. P can be broken down into a matrix of rotation and translation $P = K[R|t]$. If we happen to know the calibration matrix K then normalized coordinate for the image x can be calculated by applying the inverse of the calibration matrix on the image point. This can be expressed mathematically as $\hat{x} = K^{-1}x$. [LH81, RH04]
The equation that defines the essential matrix is illustrated in figure 9.

$$\hat{x}'^T E\hat{x} = 0$$

Figure 9: Basic property of the essential matrix. [LH81, RH04]

Unlike the fundamental matrix which has projective ambiguity (The 3-space point X may be situated on any position on the epipolar line projected by x), the essential matrix allows for "up to scale" and "four-fold" ambiguity [LH81, RH04]. Four-fold ambiguity means, that there are only four different possible solutions for a 3-space point X to be situated on an epipolar line projected by x. It is also possible to figure out which of the 4 possibilities is the correct one to achieve an "up to scale" transformation where angles and distances between points are preserved. This is called a similarity transformation giving us the basis for a reconstruction pipeline, that preserves angles, scale and distances of any reconstructed object.

2.1.6 3-space point triangulation

Once the points of interest are found and a suitable fundamental and/or essential matrix is calculated, one can move onto finding the actual 3-space points that are represented in the two images by projections x and x'. It is possible to try to use a naive triangulation approach by back-projecting the image points and trying to find an intersection of the two rays. This will usually fail due to measuring errors in the image points through which the rays are cast. This means the rays will never intersect, because the images are not accurate enough (illustrated in figure 10). Therefore a best solution must be estimated for the 3-space point by defining and minimizing a suitable cost function. [RH04]

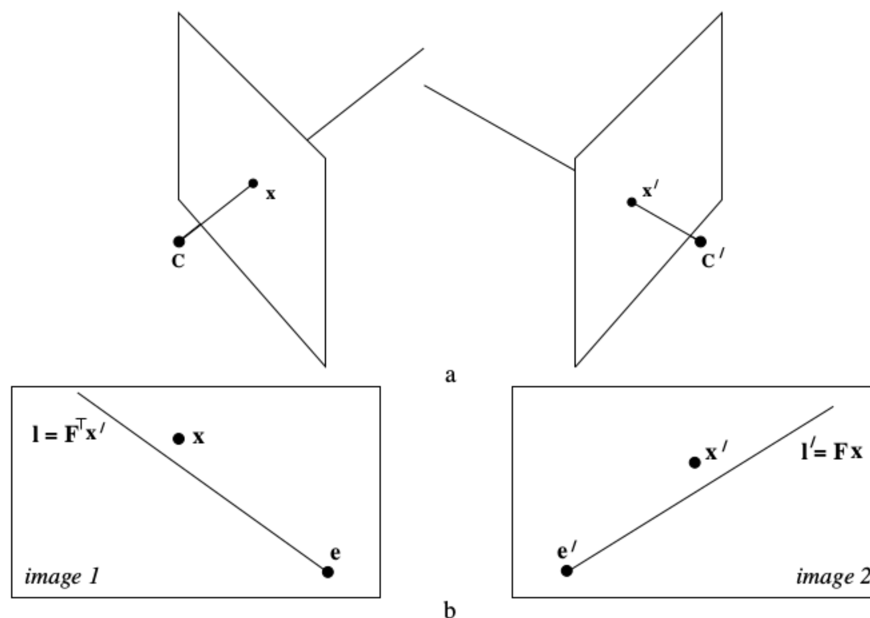

Figure 10: Naive 3-space point triangulation [RH04].

There are many different methods for triangulation of 3-space points. In this chapter we will look at one linear triangulation method. Between the two images we have two equations $x = PX$ and $x' = P'X$. (P, P') are the camera matrices of the two images. These equations can be combined into a form of $AX = 0$ which is an equation linear in X [RH04]. For the first image we would get $x \times (PX) = 0$ which written out is:

$$\begin{array}{rcl} x(P^{3T}X) - (P^{1T}X) & = & 0 \\ y(P^{3T}X) - (P^{2T}X) & = & 0 \\ x(P^{2T}X) - y(P^{1T}X) & = & 0 \end{array}$$

Figure 11: Linear equation created from image point relations to 3-space point X. [RH04]

Where P^{iT} are the rows of the camera matrix P. An equation of the form $AX = 0$ can be created with a matrix A of linear relations [RH04].

$$\mathrm{A} = \begin{bmatrix} xP^{3T} - P^{1T} \\ yP^{3T} - P^{2T} \\ x'P'^{3T} - P'^{1T} \\ y'P'^{3T} - P'^{2T} \end{bmatrix}$$

Figure 12: Linear relations A for 3-space point triangulation. [RH04]

By solving the equation $AX = 0$ we get the 3-space point X that we are looking for. Though this is not exactly the point in 3-space in the original scene, due to errors of projection of x and x' to the images. In real use cases the points are evaluated with an error cost function to achieve a more likely point X for the two image points [RH04]. We will not delve into this subject in this thesis, since it would divert us too much from the main topic, but it was worth mentioning how projection errors are fixed.

2.1.7 N-view reconstruction

N-view reconstruction is the process of using a batch of images to do scene reconstruction. In the book "Multiple View Geometry in computer vision" by R. Hartley and A. Zisserman [RH04] it is shown how one can move from a relation of $x \leftrightarrow x'$ to a three image relation of $x \leftrightarrow x' \leftrightarrow x''$ [RH04]. And from there it is possible to go one further and move from a three image relation to a four image relation $x \leftrightarrow x' \leftrightarrow x'' \leftrightarrow x'''$. This four image relation can be used in a general purpose way for an N image relation. While increasing the the accuracy of the calculated 3-space points the N-view reconstruction adds a new

valuable property to reconstruction called auto-calibration. Auto-calibration is the process of finding the internal camera parameters (required by the essential matrix) using multiple uncalibrated camera images. This property allows for the creation of a metric reconstruction, without any previous knowledge about the cameras used or any of their parameters. [RH04]

2.2 Scene reconstruction

Using the methodology described in the last chapters one can create a representation of the scene depicted in the input images given to the algorithm. The representation will be a metric reconstruction and very accurate, but it is still not a full reconstruction, because it does not form a polygon mesh. For now the reconstruction is a point cloud with varying degrees of point density throughout the reconstruction, depending on the amount of feature points found and matched on the input images. In order to get a proper polygon mesh a scene reconstruction algorithm is required. Unlike multi-view reconstruction, scene reconstruction is not a single algorithm, but rather a pipeline of actions that are performed sequentially in order to create a polygon mesh. Many algorithms have been devised for this purpose, but we will focus on the algorithms that deal with man-made constructions (houses, highrises, rooms).

2.2.1 Manhattan world-stereo

Manhattan world-stereo is a Multi-view stereo approach that promises good scene reconstruction results on surfaces with few to no textures [YF09]. This is a step forward, because usually multi-view stereo is very good, but it relies on features in order to produce a point cloud reconstruction. Since man-made architecture usually has very little to no texture (mono colored walls) it is complicated to get a dense enough point cloud in order to use conventional polygon mesh construction algorithms on the point cloud. There is also the consideration that while in nature scenes angles and lengths may not have to be very accurate, we can not make the same assumption about constructions like houses and rooms. Just a small error in the calculations and you might end up with a slanted or crooked wall, which is unacceptable in case the mesh has to be used later for some general purpose, since it no longer depicts the scene. [YF09] The manhattan-world stereo algorithm pipeline has three steps (illustrated by 13):

1. Find dominant axes
2. Generating hypothesis planes
3. Reconstruction by connecting hypothesis planes to pixels

The manhattan-world stereo pipeline starts by first trying to estimate the dominant axes of the scene. This is required since we can not trust the point

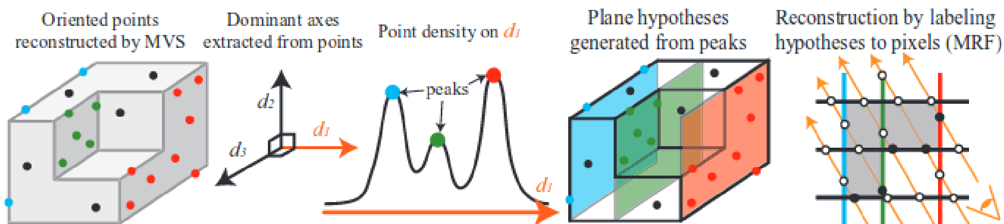

Figure 13: Manhattan-world stereo reconstruction pipeline. [YF09]

cloud to conform to any regular axes like (X, Y, Z). Most often the actual axes of the reconstruction will be subject to some sort of rotation. The dominant axes of the scene are estimated using the oriented points generated during the multi-view stereo(MVS) algorithm. The orientation value, also called normal, is a vector value usually used in conjunction with faces or polygons, to signify which way the polygon is facing (Which side is the front). While for points the normal value has no real significance, since points really can not face in any direction. This property is still useful in the reconstruction process, since the value of many points can be used to determine surfaces that those points might be part of. The dominant axes are found using a hemispherical histogram (figure 14) [YF09]. The principal used is simple yet brilliant. If we assume that the scene has many straight walls that are intersected by other perpendicular walls (rooms, houses), then we can assume that a large portion of the points in the point cloud are facing the same way as the wall. If we now take the normals of all the points and convert them to an unsigned value (we don't care which way the normal vector is pointing but only on which line it is), then we can sort all the normals into a hemispherical histogram with $\sim 10°$ slots, and check which lines are the most common. The three most common lines that are all almost perpendicular to each other are the dominant axes $\{\vec{d_1}, \vec{d_2}, \vec{d_3}\}$ for this scene. According to the researchers in the manhattan-world stereo paper, in a proper manhattan scene the three axes found usually differ from being totally perpendicular only by a few degrees. [YF09]

In the second step of the algorithm the axis aligned hypothesis planes are generated. For every reconstructed point P_i a plane can be defined with a normal equal to an axis direction $\vec{d_k}$ that passes through the point. This plane will have an offset of $\vec{d_k} * P_i$. This is due to the plane having an equation of $\vec{d_k} * X = \vec{d_k} * P_i$. If we now calculate the set of offsets $\{\vec{d_k} * P_i\}$ for each axis direction $\vec{d_k}$ then we can apply the 1-dimensional mean shift clustering algorithm on these sets to extract the clusters and peaks of the sets. The candidate planes are then chosen from the peak offsets of the clusters (where the most

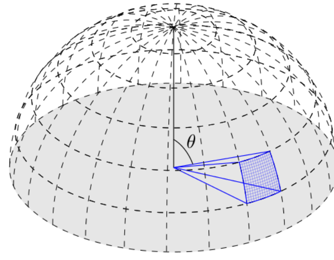

Figure 14: Hemispherical historgram example [uhs].

points were on the plane). Clusters with small sample sizes are excluded. The researchers wanted to note, that this method is used to reconstruct oriented planes. This means that they distinguish between the front and back side of the planes. Thus they always include two planes, one with the face normal pointing along it's corresponding dominant axis and one with the normal pointing the opposite direction. [YF09]
In the third step, the reasearchers use the original input images and the hypothesis planes that were created, to recover a depth map for the images. They assign one hypothesis plane to every pixel of an image and then formulate the problem as a Markov Random Field (illustrated in figure 15). The Markov Random Field is a set of variables having a markov property and is described by an undirected graph. It is sometimes used in the fields of computer vision and surface reconstruction in cases where problems have to deal with energy minimization. [YF09, RK80]

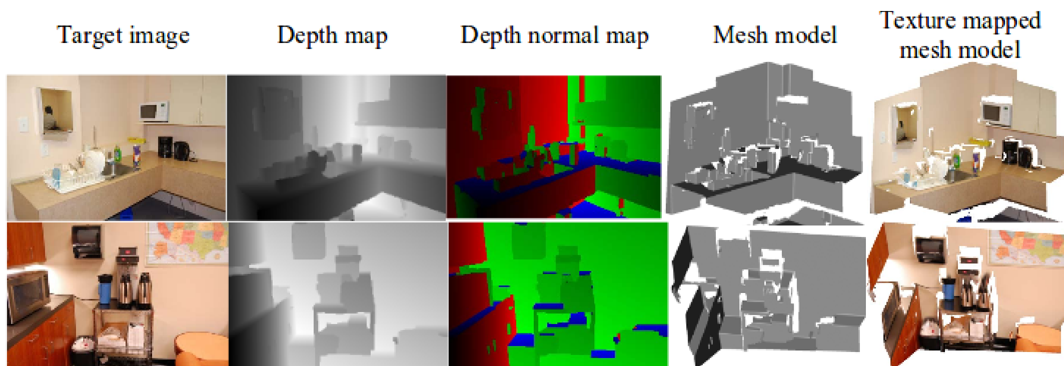

Figure 15: Manhattan-world stereo reconstruction example [YF09].

2.3 Summary

In this chapter we looked into point cloud creation and face extraction state of the art. We described how to create a point cloud using a set of input images

and later convert that point cloud into a polygon mesh. In the next chapter we will hypothesize about a new algorithm that will suit our requirements more precisely.

3 Hypothesis

As shown in the previous chapters, the state of the art in terms of structure from motion and surface reconstruction is very good. Not only will modern techniques allow us to reconstruct a metric transformation point cloud using nothing but a series of pictures taken with an uncalibrated camera, but we also have reconstruction pipelines that will allow us to extract surfaces from this point cloud, even if the pictures have few textures (like manhattan-world views). While the pipeline seems to be complete and working very well with real world data, some of the research questions posed in this thesis have been left unaddressed. As a reminder the questions of this thesis were:

- How to define an algorithm that reconstructs an interior room, without altering its dimensions.
- How to ensure that the algorithm connects up all the faces without leaving gaps.
- Can the above mentioned be achieved without using any further information than the provided point cloud.

Up until now only the first question has been answered. As we saw in chapter 2.1 by using n-view reconstruction we can ensure a metric reconstruction of the input images is created, solving our first problem of how to reconstruct an interior room without altering it's dimensions. However the other questions remain unaddressed at this point, since even manhattan-world stereo cannot promise that all faces are connected properly (as shown in figure 15), if information is missing in certain parts of the image. Also manhattan-world stereo has the dependancy of the input images, which means that if the input images are separated from the point cloud, a reconstruction is no longer possible. Next we will hypothesize of how one might try to get around these restrictions.

3.1 Idea

In order to achieve the goals set in this thesis, it is evident that we will need to develop some sort of algorithm. Much like manhattan-world stereo it will have to be a series of steps rather than one monolith algorithm trying to achieve all goals at once. While the first steps of the algorithm are the same as in manhattan-world stereo, we will try to achieve our goals without using any kind of extra information, other than the point cloud we have generated using

the Multi-view stereo (MVS) algorithm. Next we will outline the pipeline we will use for our reconstruction algorithm:

- Extract dominant axes
- Assign points to axes
- Reduce point cloud size
- Assign points to planes
- Reconstruct faces
- Ensure structural completeness

In the first step, much like in manhattan-world stereo, we extract the dominant axes of the scene, to avoid errors introduced by the rotation or translation of the input point cloud.
Secondly we assign points to axes, by calculating the angle between the vertex normal and the axis vector, choosing the closest axis and ignoring all points that belong to no axis (points that do not face along any dominant axis).
Next we reduce the point cloud size by removing all the points that do not face along a dominant axis. This is useful to reduce the computational strain of the CPU, since iterating over a large quantity of irrelevant points can waste a lot of resources.
Fourthly we assign all remaining points to planes in order to distinguish multiple planes facing in the same direction. It will also prepare us for the next step where we will try to form polygons out of these points.
In the fifth step we will try to reconstruct a ploygon mesh on top of the point sets that define the planes. The result of this step will be a set of irregular rectangles that are not connected.
Finally we wrap up by ensuring that all planes are connected and form a complete polygon mesh. This is achieved by closing the gaps left by the last step by unifying vertices of the polygons created.

3.2 Method

3.2.1 Extracting dominant axes

The first step of our algorithm pipeline is the extracting of the dominant axes. This is important if we take into account the fact that multi-view stereo allows

to reconstruct a scene as a metric transformation, it has no guarantees at which rotation or translation the resulting point cloud will be. Since manhattan-world stereo had good results using a hemispherical histogram, we decided to use the same approach. Next we will introduce a few modifications to the axes extraction process that were not used in the original manhattan-world stereo algorithm.
In the original paper the researchers used a hemispherical bin histogram with 1000 bins. [YF09] This striked the author of this thesis as odd. When we take a spherical space with all possible vector directions and then try to map this onto a hemisphere, then a trivial algorithm can be created to remove the negative values of one axis from the vectors (assuming the center point of the sphere is $(0, 0, 0)$). Effectively switching the direction of some vectors to their direct opposite. Now trying to map these vectors to 1000 bins is unnecessarily complicated, since one axis has lost half of its values (negative) the bins can no longer be divided equally between axes without making the bins of one axis more precise than the others. This can be rectified using an algorithm to ensure equal distribution of values to bins or just using a different bin quantity. The second possibility was chosen, since it was simpler. The chosen bin count was 4000, since this allows us to divide 20 bins to the full axes and 10 bins to the axis that was halved.
The second modification is that instead of finding three dominant axes from the histogram, we find six axes. The original three dominant axes and the three opposite vectors of those axes. This makes the next steps easier, when we start to reduce the point cloud size and also start to classify points to planes with face normals.

3.2.2 Assigning points to axes

After the last step we now have the three dominant vectors $\vec{d_k}$ and their three opposite vectors. We use these vectors to assign points from the point cloud to layers (groups of points identified by an axis vector. The assigning is done by iterating over the point cloud and for each point calculating the angle between the point normal vector and each of the axes vectors. If a match is found where the angle is less $10°$ than the point is assigned to a layer connected to the axis. If no match is found the point is ignored. This method is quick and efficient, since if there is a match it has to be on a single axis and duplicate assignments are not possible, because the dominant axes are all perpendicular to each other as stated as a requirement in the last step, which means that the

same holds for their opposite vectors.

3.2.3 Reducing point cloud size

The next step in our algorithm aims to reduce the size of the point cloud. This is important for a number of reasons, like filtering out the points we are interested in and of course increasing the speed of any algorithm that has to be run on the point cloud after this step.
The filtering is done by removing all points from the point cloud which do not belong to an axis layer. This removes points that do not face along a dominant axis, and therefore increase entropy in our scene, since our assumption is a manhattan-world type scene.

3.2.4 Assigning points to planes

The third step in our algorithm is to divide the points in each layer to planes. Up until now we have categorized the points we are interested in by the way these points are faced. Now we have to find a way to differentiate between different faces along a certain axis for each of our six layers that we extracted in the last step. The manhattan-world stereo algorithm solved this problem by defining hypothesis planes and reducing the problem to a one dimensional axis where mean shift clustering could be used to assign points to planes [YF09]. Our approach to this problem is somewhat more basic and time consuming, but since we have not set a time constraint to our algorithm and the result is at least as good as the manhattan-world approach, then our solution is equally valid. Our approach for distinguishing planes from one another is a simple $O(n^2)$ algorithm of taking a point and finding all points that are within a certain distance, and then recursively repeating the process until no more points are found. The assigning of points is continued until all points of the layer have been assigned to planes. Since there is a certain amount of entropy to be expected from the point cloud, all planes with less than 1000 points in them should be discarded as outliers.
Although we decided in favor of this algorithm, does not mean that this is the only solution for this problem. Any algorithm that can distinguish between planes that may be defined along the same line are equally valid for our pipeline.

3.2.5 Reconstructing faces

After the planes have been extracted from the layers we need to make these point groups into polygon mesh faces. This is not an easy task, as it has been pointed out many times, the dominant axes of the scene usually do not coincide with the (x, y, z) coordinate system. Also the points of any face rarely align on a clearly defined line, but are rather scattered in an orgranized way on the plane. There are many polygon mesh reconstruction algorithms available, like poisson surface reconstruction and the marching cube algorithm just to name a few [MK06, WEL87]. These algorithms usually work well, but are intended for much more complicated scenes than our rectangular room. This is also the reason why most of these algorithms are quite complex and hard to implement. So instead of using a very complicated algorithm we will instead be guided by the occam's razor principle, where one interpretation states that the simplest solution is usually the best one.
The reconstruction step will consist of four parts, during which we will achieve a polygon face representation of the point group associated with a plane. Also we will be able to reduce the amount of points in our point cloud even further. The four parts of the reconstruction step are:

- Rotate points to be flat along the Y axis
- Flatten the points by removing the Y value
- Use a 2D convex hull reconstruction algorithm to build the face
- Remove unnecessary points and rotate the points back

First we need to rotate all the points of the plane in such a manner, that we can eliminate one of three axes. Since we assume all planes to be flat surfaces, this can be achieved. In order to rotate the points, we need to translate the points to the zero point first. This can be done by calculating the center point of the plane (the average values over all axes) and subtracting this center point from every point in the plane. After the points have been translated a rotation can be applied in order to rotate the points to be flat along the Y axis. A single two axis rotation in 3-space in complicated, so it is easier to split the rotation part into two separate rotations. First we will rotate the points to face along the X axis, using a Y axis rotation. Then we make the points face along the Y axis using a Z axis rotation. The corresponding angles can be calculated easily using the average vertex normal, which can be calculated along side the

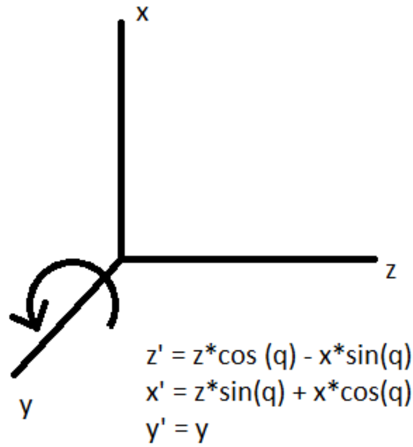

Figure 16: Y axis rotation in 3D space.

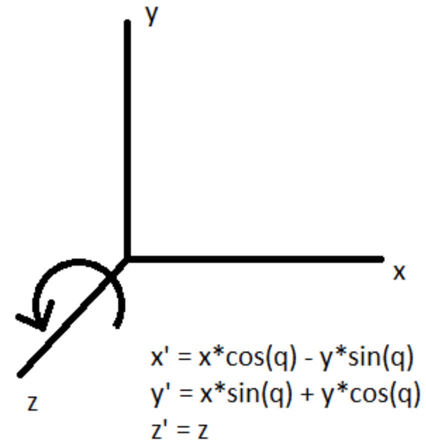

Figure 17: Z axis rotation in 3D space.

center point (illustrated in figures 16 and 17). The angle of the first rotation is the angle between the vector $(1,0,0)$ (X axis vector) and $(x,0,z)$, where x and z are the corresponding values of the average vertex normal. The second rotation angle is the angle between the vector $(0,1,0)$ (Y axis vector) and (x,y,z) where x, y and z are the corresponding values of the average vertex normal.

Since the points that constitute the face that we are trying to build are now flat along the Y axis, we can set the Y value of every point to 0. This will flatten our face and force all the points on a single plane.

Now the resulting plane can be thought of as a two dimensional object, since we have rotated and translated the plane points in such a manner, that the Y axis no longer influences it. Building a polygon from points in a two dimensional space is a very well studied field and therefore there are many algorithms available for achieving a wide variety of different results. Usually these algorithms are much more simple and straightforward than all the algorithms designed for three dimensional space. This is beneficial for us, for cutting down on implementation complexity and also algorithm runtime. The choice of the two dimensional face reconstruction algorithm is here left open for the implementation to choose whichever algorithm suits it the most. In any case, the final result should not be influenced too much by the choice of the algorithm.

Once the face reconstruction algorithm is finished, the point cloud can be once again filtered for unnecessary points that are not included in the polygon. This step should drastically reduce the number of points in the point cloud and again increase the performance of any future calculations. When the un-

necessary points are cleared, the remaining points of the plane are rotated back to their original positions by applying the same two rotations in the opposite order with negative angle values. Once the points are rotated to their proper rotation, the final operation is to translate the points back, to their original position by adding the calculated center point to each point in the plane.
By applying these four steps to each plane of the point cloud, we are left with a non connected polygon mesh of the original scene.

3.2.6 Ensuring structural completeness

The last task is to ensure structural completeness of the polygon mesh. This involves trying to connect all the planes into one single mesh of the room. Since we specified that we only want a rough estimation of the room dimensions and the floor plan, then the reconstruction does not have to be very accurate in relation to room interiors (furniture and other objects). This means that objects in the room can be replaced by cubes that stretch the entire room height, effetively marking areas of the room that are definately navigatable. This approach assumes that no scene reconstruction is under a bigger rotation than 45 degrees (the scene is not reconstructed in a manner where the floor cannot be determined). We will try to achieve this by first limiting our room space. For this we need to find the (min, max) values for all dominant axes in the scene. After the values have been found we can construct a bounding box around our planes, to encapsulate our room. This bounding box represents the boundaries of our room. This means no plane can extend beyond these walls. Next we will try to fill our room with cubes representing the objects in the original scene. Applying the following two steps to every plane created in section 3.2.5 will finish our reconstruction process:

- Make plane into a rectangle.
- Resize the plane if necessary.
- Extend the plane opposite the face normal to another plane with the same opposite face normal or to the room bounding box wall.

In order to create a rectangle from a plane all we need are the (min, max) values of the plane along the dominant axes. Once these values are found we can construct a bounding rectangle for the plane that is aligned along the dominant axes.
Next we need to check if resizing of the bounding rectangle is necessary. The

bounding rectangle should be resized to stretch from the scene floor to the scene ceiling if we have no information about the possible actual height of the object (I.e another plane with a face normal along the "up" dominant axis of the scene intersecting the plane, showing the "top" of the object).
Once the plane is of a proper size and shape the last step is to extend the plane from 2D to 3D along the face normal opposite vector, until one of two conditions holds. Either the plane hits another plane with roughly the same bounding box and a face normal that matches the vector along which the original plane is being extended (I.e we found the other side of the same object) or the extendable plane hits one of the room bounding box walls.
Applying these steps to all planes, will cause some overlapping cubes, but when rendered by a graphical program, it will be still very usable and esthetically pleasing, since the overlapping faces are not rendered.

3.3 Summary

In this chapter we hypothesized about a new algorithm that is more precisely tailored to our research questions. We described in length how different parts of the algorithm help in achieving our goals and how all the steps are necessary for it to work. In the next chapter we will take a look at the demo application that was implemented alongside the development of the algorithm.

4 Implementation

The algorithm proposed in the hypothesis chapter 3 is implemented as a simple application using the programming language Go [gol]. Go is a statically typed, compiled language developed by Google employees Robert Griesemer, Rob Pike, and Ken Thompson in 2007. Like C it is not object oriented, but Go does have memory management and garbage collection making programming more simple than languages with manual memory management (like C and C++). Go was chosen by the author of this thesis because of its simple and understandable syntax.

4.1 Structure storage format

In this thesis we are dealing with large point cloud data quantities, which can range to millions of points for one scene reconstruction. For this we will need a file format that can be easily created and parsed, but it must also be able handle the large amounts of data that we use.
In order to deal with large point cloud files a file format PLY (Polygon File Format) has been created [ply]. It was desgined to store 3D data from scanners. Since our data does not differ from the data of a 3D scanner we can also use this file format to store and manipulate point clouds in our implementation. The polygon file format has two different versions for storing data: ASCII and binary. While the ASCII version takes more room in the file, it is also more readable. The opposite goes for the binary version, which is more compact, but less readable. Next we will introduce the various parts of the polygon file format and explain in depth how they are constructed.
The polygon file format is split into two parts, the header and the body. The header always starts with the keyword:

```
ply
```

After the identifying keyword the version of the polygon file format is declared. It is usually one of three values:

```
format ascii 1.0
format binary_little_endian 1.0
format binary_big_endian 1.0
```

The version defines what to expect while parsing the body. Whether the data is ASCII or binary, and what is the endianess of the bytes. For our implementation we will use *format ascii 1.0*.

Next there are the definitions of the primitives described in the file. They are defined using two tags: *element* and *property*. The keyword *element* starts the definition of a new primitive and the keyword *property* allows to add new properties to the last element defined in the file so far. Next a small example of a vertex element:

```
element vertex 4
property float x
property float y
property float z
```

The first line of the example defines a new primitive of type *vertex*. The number 4 at the end of the line declares, that the body must contain 4 vertices. The three *property* lines after the vertex definition add properties to the vertex. In this example a vertex is defined with three float values x, y and z.
The file format can also be used to store polygons and meshes. This is achieved using a *list* of vertex indices. Next an example:

```
element face 1
property list uchar int vertex_indices
```

In this example a new *face* type primitive is defined. The line also declares that the body of the file will contain 1 face. Next a property is attached to the element, but rather than being a single value, it is defined as a list which can hold at most the maximum value of an unsigned *char* elements and where each element is an integer index pointing to a vertex in the file. Objects in the body of the file are indexed starting from 0. Lastly the header can contain any number of comments using the keyword *comment*:

```
comment This is a comment!
```

After the definition of all primitives, the header of the file always ends with the keyword:

```
end_header
```

After the end of the header the body of the file starts. The body follows the header rigidly. If the *vertex* element was defined before the *face* element, then all vertices will be declared before all faces in the body. Also the body must contain exactly the amount of objects declared in the header. Effectively the body is just a list of values that can be decoded using the header. Next there is an example of the body that would match our example header:

```
0 0 0
0 0 1
1 0 1
1 0 0
4 0 1 2 3
```

The first 4 lines in the body are vertices, that are laid out in a flat rectangle shape on the y-axis and the last line is a face with 4 vertex indices $\{0, 1, 2, 3\}$
The PLY format header that we used in our implementation is as follows:

```
ply
format ascii 1.0
element vertex <vertex_count>
property float x
property float y
property float z
property float nx
property float ny
property float nz
property uchar red
property uchar green
property uchar blue
property uchar alpha
element face <face_count>
property list uchar int vertex_indices
end_header
```

The only change to the example header is that we have added extra vertex properties for color and face normals.

4.2 Extract dominant axes

The implementation of the extraction of dominant axes follows the guidelines set by the manhattan-world stereo paper and the hypothesis section of this thesis [YF09]. In this chapter we will focus a bit more on some of the details of the implementation and explain why these choices were made.
The hemispherical histogram needed to complete the extraction was implemented as a simple array of point structures. This allows for quick access of the elements (constant time $O(1)$). As stated in the hypothesis chapter an

array with 4000 elements was used. All bins define a 0.1 size range on all axes in the following ranges:

- X-axis: from -1.0 to 1.0 (20 bins)
- Y-axis: from 0.0 to 1.0 (10 bins)
- Z-axis: from -1.0 to 1.0 (20 bins)

This means that when we take a vertex normal and convert it into an unsigned form (map it to a hemispherical space), then we can use an index calculation formula, to find the correct bin. But before we can use the formula, we need to convert the vertex normal vector *float* values to more managable *int* values. For this we use a simple algorithm, which also respects the size ranges of the bins:

```
// remove negative values by adding
// the minimum axis value
normal_value -= min(axis)

// increase float precision enough to create n bins
// and convert the value to integer
int_value := int((normal_value) / step)

// Limit values to n bins
if int_value == int(max(axis) / step):
  int_value = int(max(axis) / step) - 1
```

This algorithm maps any float value to the integer part of the precision required. In our case it has the following mapping (x-axis):

1. -1.0 - -0.91 -> 0
2. -0.9 - -0.81 -> 1
3. ...
4. 0.8 - 0.89 -> 18
5. 0.9 - 1.0 -> 19

The formula for calculating the index of the bin the point belongs to is as follows:

```
index := x * y_bins * z_bins + y * z_bins + z
```

Where x, y, z are the vertex normal vector values and y_bins, z_bins are the amount of bins assigned to the axis. Since the amount of bins for our application has been predetermined the formula can be simplified to:

```
index := x * 200 + y * 20 + z
```

After the histogram has been populated and sorted, the three most common perpendicular axes have to be found. For calculating angles between vectors we use the formula:

$$cos\theta = \frac{(\vec{u} * \vec{v})}{(|\vec{u}| * |\vec{v}|)}$$

Figure 18: The calculation of an angle between two vectors

Where $(\vec{u}*\vec{v})$ is the dot product of two vectors and $(|\vec{u}|*|\vec{v}|)$ is the multiplication of the vector lengths. The vector lengths can be found using the Pythagorean formula $c = \sqrt{x^2 + y^2 + z^2}$

4.3 Assigning points to axes

The step of assigning points to axis layers has no real suprises in its implementation. Nevertheless we will point out some of the techniques that were used in order create the sample application of our algorithm.

The algorithm follows the hypothesis chapter to the letter by iterating over all the points in the point cloud and trying to establish to which of the axes the point should belong to if any. Inside the algorithm we use the same vector angle calculation formula used in the previous chapter (See figure 18):

```
for (vertex, vertex_normal) in point_cloud:
  for (axis, axis_vector) in axes:
    if angle(vertex_normal, axis_vector) < 10:
      axis.add(vertex)
      break
```

The algorithm searches for the correct axis for the given vertex, cutting the shortly after a solution is found and jumping to the next vertex. Since the amount of axes is limited to a specific number which is small, then the time complexity of this algorithm is $O(n)$, which is quick even with millions of points.

4.4 Reducing point cloud size

The algorithm used for point cloud reduction is naive. A simple $O(n)$ time complexity cycle to iterate over all point cloud points, removing the ones, that do not belong to any axis layer,

4.5 Assigning points to planes

Assigning points to planes is the step where our approach diverges greatly from the techniques used in manhattan-world stereo. Instead of using a fast algorithm that tries to match points to hypothetical planes and then uses a clustering algorithm to estimate the most likely plane positions, we instead use a bit slower algorithm which differentiates between different planes just as well, while being much simpler and a bit more time costly. Since we have set no limit on running time for our algorithm, then this was a trade-off the author was willing to make for the implementation of the algorithm example application.

The assigning of the points in the implementation works by using a similar approach to breadth-first search. Unfortunately the points in our axes layers do not form a graph of any sort and creating this graph would take about the same amount of time as our algorithm in total. So the solution is to introduce a costly second cycle inside the first one, iterating over all points once again, to find neighbouring points that are closer than a certain distance. This brings the time complexity of our algorithm up to $O(n^2)$, but reduces the complexity of the code considerably. Also this algorithm works really well for trying to distinguish between different planes that would be mapped to a single hypothetical plane in the manhattan-world stereo algorithm (E.g A wall with an extruding support column. The manhattan-world stereo algorithm would only extract two planes, while our approch would extract three).

The pseudocode for extracting a single plane from a layer is as follows:

```
// add random point to stack
stack.add(layer[rand])
layer.remove(stack[0])
while len(stack) != 0:
  for point in layer:
    if distance(point, stack[0]) <= maxDistance:
      stack.add(point)
      layer.remove(point)
```

```
    plane.add(stack[0])
```

This algorithm is then repeated, until the layer contains no more points. As a result of this process a lot of planes will be created, most of them will be very small with only a few hundred points. This is why we discard all the planes that have fewer than 1000 points in them. It is important to do this additional filtering as it cleans up the image more by removing additional noise from the input data, leaving a nice clean data set to work with in our next step.

4.6 Reconstructing faces

The face reconstruction step is simple in its approach. It only requires basic knowledge of geometry. Both of the rotations can be implemented directly using the information from figures 16 and 17. The removing of the Y value from each point in the process is also a trivial task.
The two dimensional convex hull algorithm chosen in the demo implementation of this thesis is the gift wrapping algorithm [Jar73]. The gift wrapping algorithm follows a simple principle of starting on the left-most point in the point set given and moving from point to point along the left most edge of the point set. The check of whether a point is aligned more left-side than another is done by using the vector cross product between the pairs: the current point and the first test point and the current point and the second test point. If the cross product is larger than zero then the second test point is on the left side of the first. The points are followed along the left edge, until the original starting point is found, during which the algorithm finishes. The gift wrapping algorithm is fast and primitive. It does not expect any strcture in the point layout. Next we will show the pseudocode for the implementation of the gift wrapping algorithm used in the demo application:

```
// Find left most point
leftMostPoint := points[leftMost]

// Gift wrap
pointOnHull := leftMostIdx
j := 0
P := []
for {
  P = append(P, pointOnHull)
  endPoint := points[0]
```

```
  for k := range points {
    if (endPoint == pointOnHull) || (points[k] left of P
        [j]) {
      endPoint = points[k]
    }
  }
  j += 1
  pointOnHull = endPoint
  if endPoint == P[0] {
    break
  }
}
return P
```

After the algorithm finishes the point cloud is filtered to remove excess points that are not needed to represent the polygon that was constructed. Finally the points are rotated and translated back to their original position.

4.7 Ensuring structural completness

The final step of the reconstruction pipeline is the connecting of planes in order to create one solid mesh wtihout structural errors. This step requires no complicated mathematical knowledge. It rather tries to use common logic and reasoning in order to complete the task we have set.
The implementation of the required steps described in chapter 3 section 3.2.6 does not quite follow the guidelines set. Next we will explain the peculiarities of the implmentation compared to the theoretical algorithm.
Firstly the theoretical algorithm for ensuring structural completness does not require the dominant axes to line up with the more common (x, y, z) axes. This however would require some linear equation solving in order to calculate bounding boxes for planes not aligned along these axes. So in order to decrease the amount of work that is needed to be done it is beneficial to first rotate the planes so that the dominant axes line up with the regular axes of Cartesian coordinate system. The required rotations are the same that they were in 4.6 only this time the rotation is applied to all planes at once, using the point cloud center point as the pivot. There is also an additional rotation on the Y axis to rotate the dominant axis with the highest histogram value to be the new X axis. Since the theoretical algorithm already contains the constraint

that the reconstructed point cloud cannot be rotated more than 45 degrees (a floor cannot be determined), then finding the appropriate "up" axis among the dominant axes is easy and therefore we can rotate the planes to reduce the computational complexity for the rest of the step.
Secondly the implementation does not feature the requirement to connect opposite facing planes with each other, since it was noticed, that the test point clouds did not have any of those features in them. This decision was purely based on time constraints on the thesis scope, but for an algorithm that works on all point clouds this feature should be added.
To complete the implementation a generalized algorithm was designed that could retrieve the boundaries of a plane on each axis using a helper function that was passed as an argument. This is where the programming language choice was beneficial, since in Go all functions are first class objects, which means they can be constructed at runtime and passed as arguments among other properties. Next we will show the pseudocode for this function.

```
Find-Min-Max(axisGetter):
  min = axisGetter(points[0])
  max = axisGetter(points[0])

  var val float32
  for each point in points:
    val = axisGetter(point)

    if val < min:
      min = val
    if val > max:
      max = val

  return min, max
```

The type axisGetter is a shorthand for a function that takes a point argument and returns one of the axes of the point.
Once the boundaries of the plane are known all that is needed to be done, is to extend the plane and use the known values to add 8 points of a cube into the pointcloud and define 6 new planes. Then the original plane may be deleted from memory. Doing this with every plane retrieved from chapter 4.6 gives us our full point cloud reconstruction.

4.8 Summary

In this chapter we looked at the demo application that was implemented during the development of our scene reconstruction algorithm. We delved into implementation specifics and argumented why certain things were done differently in the practical application compared to the theory. In the next chapter we will be looking at the results of our reconstruction algorithm and assessing its effectiveness.

5 Results

In this chapter we will present the results of our hypothesized algorithm step by step, and give an assessment to the results of each part of the algorithm. All images in this chapter are created using the open source graphics program Meshlab [mes].

In order to have a good comparison of whether our hypothesized algorithm worked well or not, we need something to compare it to. Next we will show the test point cloud unaltered visualization image and after that the results of the algorithm in different steps in the pipeline. The test point cloud is created by the UT distributed systems group.

Figure 19: Original point cloud created from images by the UT distributed systems group

As can be seen in figure 19, the pointcloud has a lot of points. Some parts of the cloud are more densly filled with points while others have very little to no points. This the very same problem discussed in chapter 2.2, where parts of the scene with lots of texture will have also a lot of points while monotone stretches of wall may appear to have holes in them. It is also worth to note the clumps of gray points in the middle-left and middle-right of the image, which is actually noise introduced by the reconstruction process and doesn't actually appear in the original scene at all.

The first task of our algorithm was to estimate the dominant axes in our scene. The main goal was to fix rotation or translation issues introduced by the reconstruction process, by providing the three axes that have the most points facing along them. Since the assumption was that of a manhattan-world scene, then those three axes would be dominant axes for the scene. This

step did not deal with filtering the point cloud, so all axis unassigned points are still visible.

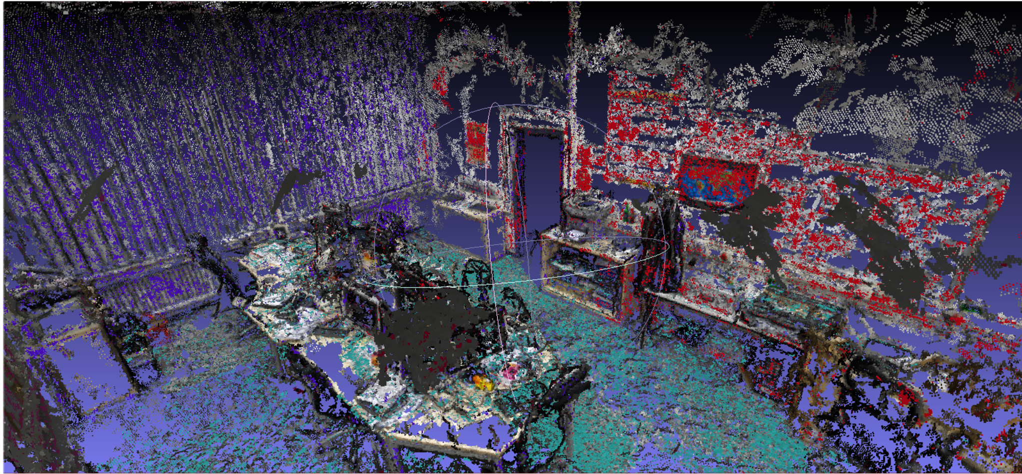

Figure 20: Point cloud with dominant axes found (blue, green and red in the picture) and all suiting points attached to the correct axes

While figure 20 remains still noisy it is visible from the picture, that the extraction of dominant axes was a success, since each wall of the room is a different color and there seems to be little to no overlapping of the axes colors. This is exactly what we wanted achieve in this step. By this grouping alone we have reduced the amount point to be processed significantly.
The next step in our pipeline is the filtering of axis unassigned points from the point cloud. This step not only releases the memory used up by these structures, but also increases iterating speed and saves valuable computing time.

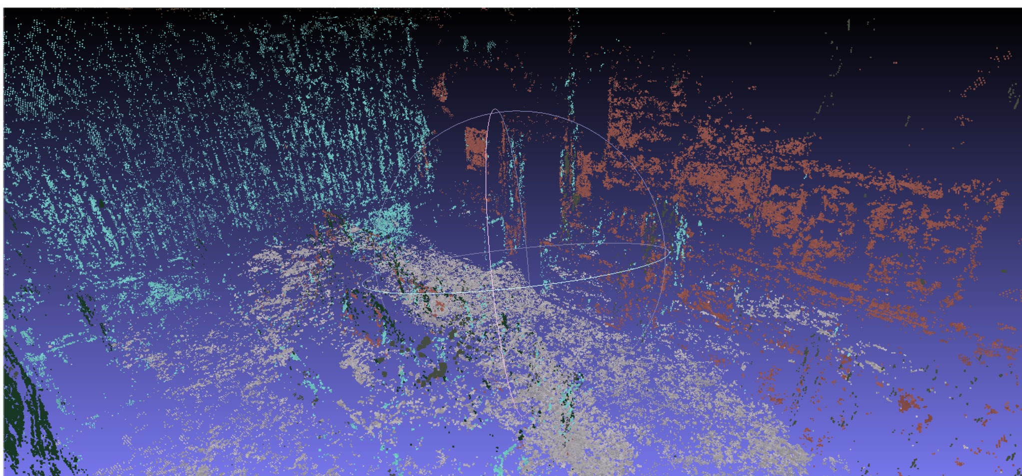

Figure 21: Filtered point cloud with only the points assigned to dominant axes left

In figure 21 a very large amount of points have now been removed from this picture, but while usually losing information is bad, then in our case it is actually good. The points that were removed were not facing along any axis, and were therefore not of much use to us, since they would have made the next steps only more complicated, by introducing unnecessary noise. Now what is left, even if it may seem sparse in comparison to the original, is actually the essence of the manhattan-world scene that we are trying to reconstruct.
While now the scene may still seem a bit vague, with some clouds of points in only three different colors it will become much more defined and structured once we apply the next step of our reconstruction pipeline on our point cloud.

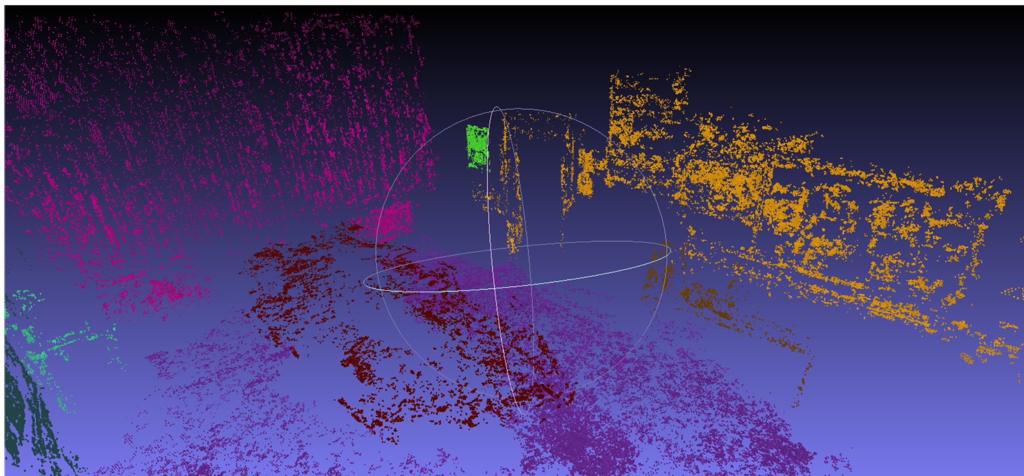

Figure 22: Point cloud with points assigned to specific planes

In figure 22 all of the remaining points are assigned to planes. Now it is already quite well visible how different parts of the picture are of different color (floor, table, walls, shelf). A poster on the wall has also been colored in a different color than the rest of the wall. This is due to the fact that the wall is mono colored and the poster was secludedly in the middle part of the wall. In any case this will not stop our reconstruction process.
The fifth step in our pipeline was to reconstruct the polygon faces of the scene using the extracted planes as input. All the planes are reconstructed using the two dimensional gift wrapping algorithm explained in detail in chapter 4.6.
As we can see in figure 23 all of the planes extracted in the last step have been flattened and covered by a simple convex mesh, created by the gift wrapping algorithm. All of the major faces of the scene have been reconstructed successfully and there seem to be no defects in the reconstruction process. The amount of vertices in the point cloud has also decreased. Dropping from over two hundred thousand points to a mere two hundred and fifty points in figure

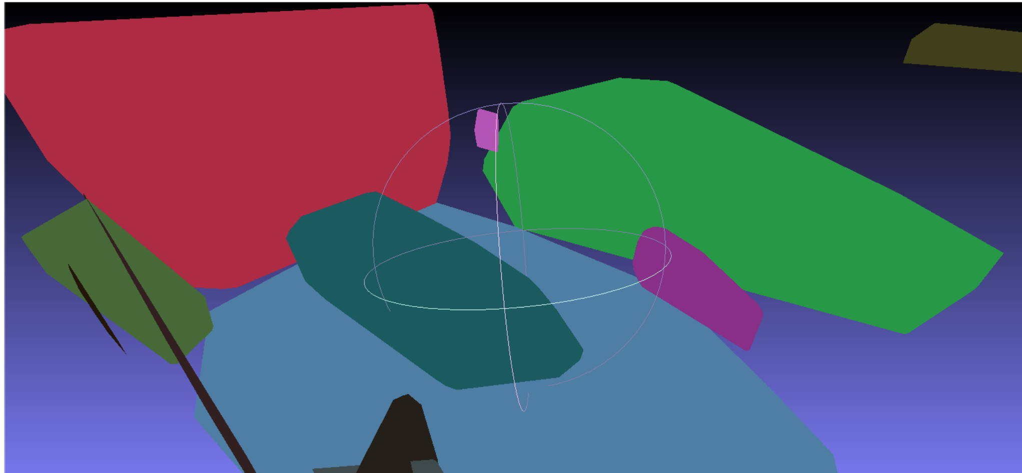

Figure 23: Point cloud with planes reconstructed to faces

23. This result sets us up for the last step in the pipeline, where we try to assure structural completeness of the scene, which requires us to connect our independant polygon faces into one continuous structure.
In the final step (figure 24) we built a bounding box for our room and populated it with cube objects using our planes as a guideline according to section 3.2.6.

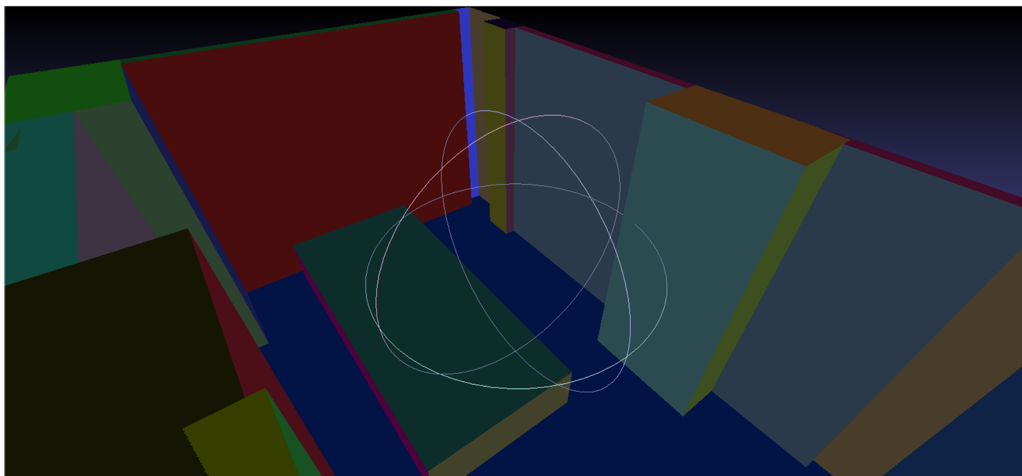

Figure 24: Point cloud with structural completeness.

This completes our reconstruction pipeline. The result is satisfactory. Due to lack of points in certain areas of the point cloud the reconstruction does not feature some of the objects in the original scene. It does however give a rough estimation of the room dimensions and floor plan. It is also structurally complete due to the way the reconstruction was done, which means that there can never be incomplete features(holes) in the floor or walls.

6 Conclusion

In this chapter we will be assessing the results of the thesis and whether or not it fulfills all the goals set in 1.2. First lets reiterate what our goals are were.

- How to define an algorithm that reconstructs an interior room, without altering its dimensions.
- How to ensure that the algorithm connects up all the faces without leaving gaps.
- Can the above mentioned be achieved without using any further information than the provided point cloud.

Starting from the top we have the question of how to create an algorithm that does not alter the dimensions of the room and gives us correct distances and angles for the reconstruction. As we learned in chapter 2.1 modern structure from motion techniques gives us a metric transformation, meaning that all lengths and angles are preserved during the reconstruction process. Since our alogrithm only works with data generated by the N-view geometry algorithm and does not alter point information in a significant way, we can be assured that our reconstruction pipeline also cannot change room dimensions in a significant way.

The next question to be answered was about structural completeness. We wanted that the result of our algorithm could have a practical application and this requires that there are no holes or odd geometry inside the mesh. This problem was addressed in the final step of our reconstruction process. The bounding box of the room combined with the addition of only cubes extended to other faces creates an impossibility for erroneous geometry to form during the reconstruction process.

The last question that needed to be answered was whether all this could be achieved without any additional information like original pictures the point cloud was generated from. During the pipeline no other information than the input point cloud is used, meaning that indeed we have created an algorithm that only relies on point coordinate and point normal values.

As we have shown, all of the questions have not only been answered, but also proven to be possible. Since we have successfully answered the questions we set out to research and created an algorithm and an implementation of said algorithm in the process we can conclude that the thesis has been successful.

7 Future work

This chapter outlines possible ways the research done in this thesis can be expanded upon by highlighting certain parts that could be improved. These are parts that were left unaltered due to improvement complexity and time constraints set on the thesis scope. This does not mean that the current solution is not sufficient or adequate, but rather that the current solution is not as efficient as it could be.

There are three places where the thesis can be improved immediately. Next we will name the shortcomings and give a brief overview on how one might go about improving them.

- Assigning points to planes
- Better convex hull method
- Floor shape used in reconstruction

Currently the assigning of points to planes is done using a less than efficient algorithm that has a time complexity of $O(n^2)$. With hundreds of thousands of points this becomes an issue, since it takes n^2 amount of time for any amount of n points that we identify to be useful in the scene. With larger scenes this could literally mean hours of waiting for the algorithm to finish. The current algorithm is a possible candidate for parallelization which would improve the amount of time the algorithm needs, but the converting of the algorithm into a parallel one is a problem that is less than straight forward. Due to the breadth-first search like algorithm that we use, we would need to start handling complex data races in the stack construct. A much more logical approach to the problem would be to use the same algorithm used by the researchers of the Manhattan-world stereo article [YF09]. Their combination of mapping axis points to a 1 dimensional line and using the mean shift clustering algorithm would improve our own algorithm greatly. This however would still not be sufficient, since their approach does not give us the property to distinguish two planes on the same layer. For this we would still recommend to run the slow $O(n^2)$ algorithm, but now not on an entire axis of points but merely on a layer. This would improve computation times due to the divide and conquer like approach to the problem and could easily be run in parallel.

The next improvement to introduce is the final result of the algorithm. currently we are reshaping all of our planes into rectangles. This is partly because it is easier to work with rectangles, but partly also beacause most convex hull

algorithms only envelope the points rather than try to find the shape the points represent. If an algorithm were used that also found the original shape of the points we could make the final results much more detailed than just cubes reaching from the floor to a plane or the ceiling. By using a convex hull method that maps the plane to a more precise polygon we can create much more accurate representations of the objects inside the room when we are extending the planes. Instead of the regular rectangular shapes we could have complex polygons for tables, shelves and chairs.
The last improvement to the algorithm we propose is the usage of the floor plane in order to eliminate some errors created by the lack of planes extracted by the algorithm in the early steps. The idea is that usually the floor of a room is a very textured surface and therefore has a lot of points on it. Also the floor usually reflects very well the objects that are positioned in the room. Since our goal was to get a rough estimation of the room and its floor plan we could use the floor plane in order to find possible objects that may not show up otherwise due to the lack points in the scene. All that would be needed is to exclude the actual floor plan from the bounding box to obtain the objects visible from the floor plan. Later we can still add more planes and extend them as per the algorithm, but in case some planes were not extracted the floor extension could act as a backup.

References

[Fau92] Olivier D. Faugeras. What can be seen in three dimensions with an uncalibrated stereo rig? *Proceedings of European Conference on Computer Vision.*, 1992.

[fea] Feature detection and description. `http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_table_of_contents_feature2d/py_table_of_contents_feature2d.html` Last visited 17.05.2017.

[gol] The go programming language. `https://golang.org/` Last visited 17.05.2017.

[Har92] Richard I. Hartley. Estimation of relative camera positions for uncalibrated cameras. *Proceedings of European Conference on Computer Vision.*, 1992.

[Jar73] R. A Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 1973.

[KB07] Joan Davies Ken Binmore. *Calculus Concepts and Methods.* Cambridge University Press, 2007. p. 190.

[LH81] H. Christopher Longuet-Higgins. A computer algorithm for reconstructing a scene from two projections. *letters to nature*, 1981.

[Low99] David G. Lowe. Object recognition from local scale-invariant features. *Proceedings of the International Conference on Computer Vision*, 1999.

[mes] Meshlab homepage. `http://www.meshlab.net/`.

[MK06] Hugues Hoppe Michael Kazhdan, Matthew Bolitho. Poisson surface reconstruction. *Eurographics Symposium on Geometry Processing*, 2006.

[ply] Ply - polygon file format. `http://paulbourke.net/dataformats/ply`.

[RH04] Andrew Zisserman Richard Hartley. *Multiple View Geometry in computer vision.* Cambridge University Press, 2 edition, 2004.

[RK80] Laurie J. Snell Ross Kindermann. *Markov Random Fields and Their Applications.* American Mathematical Society, 1980.

[sif] Sift: Theory and practice. `http://aishack.in/tutorials/sift-scale-invariant-feature-transform-introduction` Last visited 17.05.2017.

[uhs] Unit hemisphere image. `http://i.imgur.com/9gxCkK9.png` Last visited 17.05.2017.

[WEL87] Harvey E. Cline W. E. Lorensen. Marching cubes: A high resolution 3d surface construction algorithm. *ACM Computer Graphics*, 1987.

[Wit83] A. P. Witkin. Scale-space filtering. *Proc. 8th Int. Joint Conf. Art. Intell.*, 1983.

[YF09] Steven M. Seitz Yasutaka Furukawa, Brian Curless. Manhattan-world stereo. *Computer Vision and Pattern Recognition IEEE Conference*, 2009.

Non-exclusive licence to reproduce thesis and make thesis public

I, **Sander Tiganik**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

   of my thesis

   Rough estimation of interior dimensions using structure from motion techniques

   supervised by Artjom Lind

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 15.05.2017