Rheinische Friedrich-Wilhelms-Universität Bonn
Insitiut für Informatik VI
Autonomous Intelligent Systems

Diploma Thesis

Stairs Detection and Modelling Using RGB-D Images

Author: Moataz Elmasry
First Reviewer: Prof. Dr. Sven Behnke
Second Reviewer: Dr. Volker Steinhage
Tutor: Dirk Holz

March 28, 2013

Abstract

It is a great challenge for mobile robots to be able to overcome stairs. This could prove very useful in "Urban Search and Rescue" *(USAR)* scenarios, domestic service robots and other similar scenarios. This diploma thesis aims to present a method to detect stairs in an urban environment, describes them precisely in terms of numbers and dimensions and finds out the camera's parameters like pose, height and distance to stairs.
In the domain of stair detection, our approach provides information for four types of situations with respect to stairs: (i) Approaching the first step. (ii) traversing the first step. (iii) traversing the Flight-of-Stairs. (iv) Landing. In this thesis we present an approach to detect and model stairs using 2D and 3D computer vision algorithms that process a stream of inputs of an RGB-D camera to detect stairs and to construct and maintain an incremental global stairs model from successive stairs observations.

Acknowledgement

My profound gratitude goes to Professor Dr. Behnke und Dirk Holz, my mentor and diploma thesis advisor. I thank them for the constant support, brilliant ideas and interest in the research.
I wrote this thesis at the Rheinische Friedrich-Wilhelms-Universitaet Bonn and I'm grateful for the work space I've been offered. I also wrote this thesis at Fraunhofer IAIS Institut that offered me a work place and the financial mean, for which I'm very grateful. The diploma thesis has been funded by the EU FP7 NIFTi project (Project 247870). Last but not least I thank with all my heart two amazing ladies in my life, my mother, Eaman Elsalawy, and my girlfriend, Ieva Zuozaite, who have both been always there for me and offered nothing less than great attention and support.

Contents

Chapter 1

Introduction

Consider a domestic service robot with high mobility that should be able to traverse a multiple story building. This robot must then be able to overcome stairs to achieve its tasks on different floors. In this case a robot might try to blindly climb stairs like in the work of Nishiwaki et al. [33] who developed the humanoid H7. This robot has the ability to climb single steps after manually positioning the robots in front of them and did not employ any kind of sensors, i.e., it did not employ stair detection at all. So a question arises here: "Is it necessary to detect stairs at all?. Can stairs climbing procedure be achieved only mechanically with no vision sensors and no stair detection?". The answer to the later question is yes and no. Yes it is possible and has been done as seen in Nishiwaki et al. [33] or in the work of Ben-Tzvi et al. [11] who used a reconfigurable tracked robot to climb stairs mainly by placing the robot's flippers on the stairs and simply drive forward depending on the robot's stable design to traverse stairs. The problem of blind climbing, as we shall call it, is that it is prone to wall hitting, flipping over of a step if it is too high or to tip-over the stairs if they have no bounding walls [11]. In this case stair detection is necessary.

Employing a human operator to drive the robot would not be a suitable choice for a domestic service robot, since the owner expects the robot to fulfill his assignments autonomously. Further, in the best circumstances it is difficult for the human operator to intercept information about the robot's state and surrounding environment through the robot's perception. Furthermore, the act of remote controlling a robot in overcoming an obstacle causes high cognitive load of the operator, especially if the obstacle or the robotic system requires multiple well-timed manoeuvres or manipulations to accomplish the task [42]. This load is even getting higher if the human operator has no direct line of view to the robot and is forced to use the sensory equipment of the robot, which only supports him with a limited situational awareness [48, 30, 18].

3D data Before we commence our discussion we shall define few terms related to 3D data that we shall use throughout the thesis:

1. *RGB-D data*: Alternatively this is also called a point cloud. This is nothing more than a set of 3D points, where each point contains (x, y, z) components and z is the depth component. Another synonym for 3D image is the depth image. In addition, each point contains color information in an RGB field of the point. So RGB-D image is an acronym for *("RedBlueGreen-Depth image")* .

2. *Organized Point Cloud*: Is also a point cloud whose points are organized. There are sensors and 3D cameras that are able to produce organized point clouds like the Microsoft Kinect. Organized means that the points are organized into a 2D table, where each point belongs to a table cell with (u, v) coordinates. This is not to be confused with the 3D coordinates of each point, (u, v) coordinates are simply the point's indices in the table. This organized structure is exploited by several algorithms to find out the points' neighbours where the neighbourhood knowledge is vital to many 3D image algorithms.

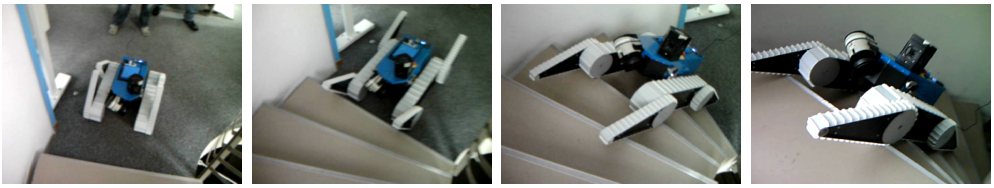
In this thesis we address the domain of stair detection and modelling using only an RGB-D camera before, and during climbing the stairs and be able to update our knowledge about the detected flight of stairs, which is a synonym for a staircase or a series of consecutive steps. We set our goals as follows in a top down manner:

1. Provide information regarding the stairs.
   a) Detect the position of the camera related to the ground and the stairs.
   b) Model the stairs through a set of parameters.
      i. Detection and modelling of stairs.
      ii. Update the stairs model regularly.

We consider standard undamaged house stairs which is on average 25 *cm* depth and 18 *cm* height as defined by the German institute for norms in a section defining the German building standard including the house staircase [20]. we should also differentiate between two cases: ascending and descending the stairs. In each case there are four possible scenarios/situations that our approach should handle:

- Approaching the first step
- Traversing the first step.
- Traversing the Flight of stairs.
- Landing

Also it is important to differentiate between going upstairs and going downstairs as in figure: 1.3.

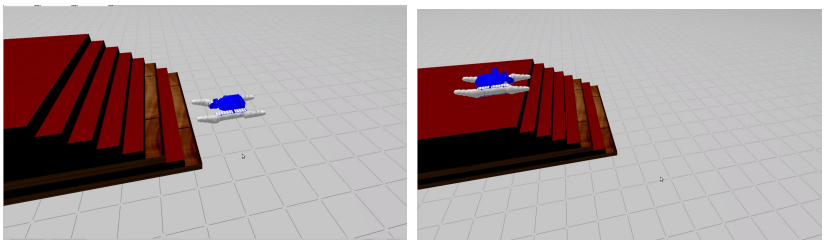


(a) Approaching the first step (b) Climbing the first step. (c) Climbing the Flight of stairs (d) Landing

Figure 1.1: The robot in the image is a production of the company Bluebotics [2] in Lausanne, Switzerland and is produced as part of the EU research project Nifti [4]. The robot is equipped with a SICK laser scanner [7] and 8 RGB cameras (ladybug). The robot is equipped with two front and two back flippers as well as a differential which enable the robot to climb stairs in a steady and stable fashion.

(a) Approaching the first step (b) Climbing the first step. (c) Climbing the Flight of stairs (d) Landing

Figure 1.2: The four scenarios of climbing the stairs as explained in fig1.1 through the camera's view. The images have been taken using a Microsoft Kinect [3] with a resolution of 320 x 240.



(a) Going Up (b) Going Down

Figure 1.3: Going upstairs and downstairs. Images are from an implemented ground vehicle like the one from Bluebotics in 1.1 in gazebo simulator in ROS [6]. This simulator uses a markup description language called URDF and the simulated robot in this figure has been written from scratch using URDF. Later in the experiment phase 3D images from gazebo will be generated while driving up and down the stairs and we shall try to detect the stairs from these images and verify the result. This should prove advantageous since the ground truth data of the simulator are available.

1.1 Stair definition

A flight of stairs is defined through a set of attributes as explained in fig. 1.4.

- Flight of stairs: A set of consecutive steps constitutes a staircase.
- Step: contains two parts, the horizontal plane (Tread) and the vertical plane (Riser).
- Edge: The meeting line between a riser and tread. There are convex edges between each tread and riser and concave edges between each riser and tread, they occur on the staircase in an interchangeable manner. An edge can belong to at most two planes: one is a riser the other is a tread.
- Landing: The large horizontal part at the start and end of stairs. This part is used as a rest place and to change staircase direction.

Each plane has the following attributes:

1. Normal vector to the plane.
2. Centre.
3. Whether the plane is horizontal, vertical or none.
4. Width and depth of horizontal planes and width and height of vertical planes.

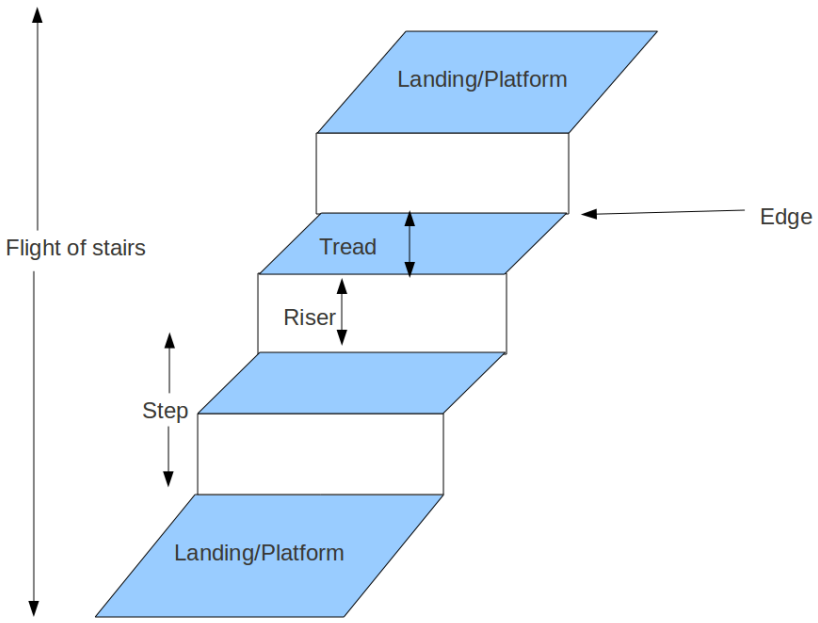


Figure 1.4: The parameters used to define a staircase or flight of stairs. A staircase consists of a series of stairs with a platform/landing at the beginning and end of a staircase to change staircase direction or take a pause.

The rest of this thesis is organized as follows: in chapter 2 we disucss the state of the art in this field. We present then our method in chapter 3 where we detect planes through a region growing algorithm that uses RGB-D images as input. We shall also use two different edge detection approaches to detect edges between horizontal and vertical planes. Then we combine the detected edges and planes to build an initial stairs model as explained in the model building chapter 4. Furthermore, we shall increment and correct an accumulated model for the whole staircase using an *Iterative Closest Point* (ICP) approach as explained in chapter 5. The results in chapter 6 shows the results of initial and incremental models constructions.

Chapter 2

Related Work

In this section we shall present the state of the art for stair detection algorithms used by the corresponding robots to climb stairs.

2.1 Edge detection

Fig. 2.1 is a an example of a tracked mobile robot [28] with two front flippers that could be controlled separately. The robot includes a stereo camera for obstacle detection in an outdoor environment and a laser range finder for indoor environemnt.

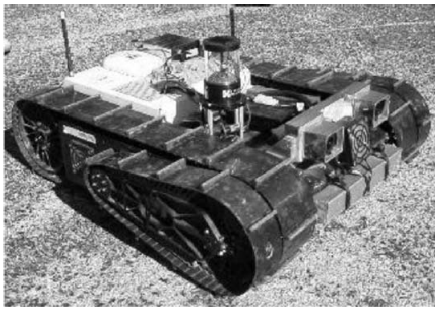
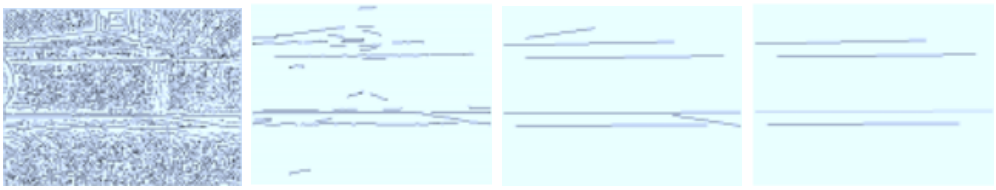


Figure 2.1: Urban II by IS Robotics [28]. This robot is equipped with two front flippers to allow more mobility and overcoming of obstacles. It is also equipped with a laser scanner for indoor perception and an RGB camera for outdoor perception. The paper mentioned that the stair detection experiments have been conducted only using the RGB camera in an outdoor environment (source: [28]).

The paper presents a method for the robot using the flippers to actively and autonomously climb stairs. The robot is able to detect obstacles through a stereo camera. Its detection algorithm is applied directly to the disparity image map using different thresholds. Differentiation between three types of pixels is made according

to pixel height: (1)no obstacle, (2)traversable obstacle and (3)non traversable obstacle. These labelled pixels are then projected onto the ground plane to construct a local occupancy grid map. When the robot detects an obstacle and the pilot decides that these are stairs and could be climbed, then a climbing process is initiated. The robot then drives to the first step using a wall following similar approach and adapts itself parallel to the first step. The pilot also gives in advance the number of steps to climb. Detection of stairs takes place through canny edge detection [14], so that the first edge is the step start, second edge with the same height is step end, then the third edge with a higher magnitude is the start of the next step and so on. The vision algorithm determines on each detected step the centre point. The robot then should always follow this center point, i.e., stays parallel to the step. Through such logic the robot avoids tipping over or hitting the wall by remaining in the center as much as possible. The robot climbs the number of steps he is instructed to. Through the inclinometer the algorithm reports landing on the ground. Through a combination of center point following and gyroscope reading as well as the number of steps climbed the robot tries to maintain awareness of the stairs.

The vision algorithm based on disparity map is strongly affected through shadows and lights. This leads to over or under detection of steps, so that for instance two steps close to each other are detected as one. Also the algorithm does not address stair descending problem. Furthermore, it describes the start and end of each step, but it fails to provide information, whether a given step is good to climb, for example there are no gaps on the step. Additionally there is no modelling of the stairs and so the information about the detected edges and steps are lost. The edge detection result of this algorithm can be seen in fig. 2.2.



(a) Input image (b) Raw edge detection (c) Result after dominant orientation filter (d) Result after short lines removal and merge neighbouring lines

Figure 2.2: Extracting edges from steps. Raw image (a). Raw result after detecting all edges using canny edge detection(b). After filtering the lines not belonging to dominant orientation in a histogram, noisy and too short lines as well as lines whose orientation is far from the horizontal axis and can not be part of an edge of a step(c). After merging the neighbouring lines and removing the short lines (d). (source: [28]).

2.2 RANSAC Plane Fitting and Polygonalization

Another approach is proposed by [29] within the project *Leaving Flatland*, that uses point clouds to perceive stairs using an RHex robot [50]. The robot is six legged robot with manoeuvre ability, that is developed by Boston Dynamics. It can move with high speed on flat ground and has the ability to traverse rocky terrains and climb stairs. It is equipped with a stereo camera that builds a 3D map of the environment and updates it as the RHex moves. As a part of the project Leaving Flatland the robot is expected to apply navigation algorithms to build a map of the ground floor as well as of higher floors. This project handles many challenges such as perception, localization and navigation in a fully 3D indoor and outdoor environment.

Figure 2.3: RHex robot developed by Boston Dynamics. As can be seen it is a six legged robot with a stereo camera at the front (source: [50]).

The algorithm receives point clouds data as inputs. An octree is then constructed from the given data, where points that are spatially related are assigned to the same tree cell. After that a plane fitting is applied to the points using RANSAC to extract at most one plane from each cell . Finally the inliers are projected onto that plane and a polygon is calculated. As a post-processing step, polygons lying on the same plane are merged and small planes are removed. As a final step, polygon labelling takes place.
There are four types of labels:

- Ground level polygon.
- Level polygons, almost parallel but higher level polygons.
- Vertical polygons.
- Steps: Consecutive (Ground) level and vertical polygons.

2.3 Stair climbing using Humanoid

Osswald et al. [38, 37] present a work using the humanoid robot 'Nao' to climb stairs. The robot is capable of climbing straight as well as spiral stairs [37]. The robot is equipped with a laser scanner, which is used for plane detection from far distance and model building as explained later, and with an RGB camera for edge detection from close distance.

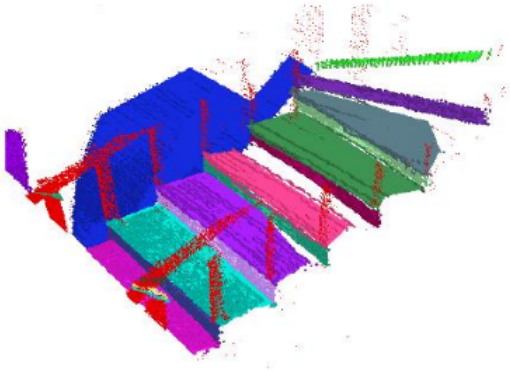
Plane detection For plane detection the authors use a scan line grouping algorithm [27, 26]. A range-image is built accumulatively through waving the robot's head left and right and build the range image from the different observations. The range image is organized into pixels and so the neighbourhood relationship is known. Line segments are then constructed by means of region growing of line segments through least square optimization function.

Next step is to build planes from line segments in an accumulative manner. First they search for three neighbouring line segments and construct a plane seed region from them. Neighbour lines means that the line segments belong to neighbouring scans in the range image. These line segments must also overlap in their starting and end points. To consider these lines as a seed region for a plane they have to pass a plane check test: the standard deviation of the plane fit of all points in these lines (using least square error) shall not exceeds a factor of two (2) of the standard deviation of each line. After that the neighbouring line segments are added incrementally as long as the standard deviation rule holds. When the plane can not be grown further, then a new seed region is searched using the mentioned approach. When no more seed regions could be found then the algorithm terminates.

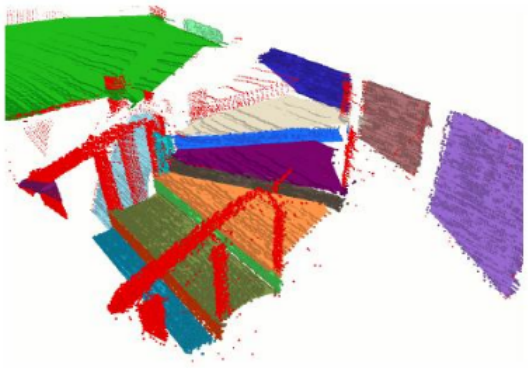
After that a post-processing step is applied where border lines are compared to see if they better fit the neighbouring planes. Also neighbouring planes with the same orientation are merged together.

Model building A model is built from the detected planes. This is a straight forward approach where the intersection lines between the horizontal and vertical planes are found as well as the boundary lines of the planes using a convex hull algorithm. The endpoints of the parallel lines are then connected to form the plane model.

After that a canny edge detection algorithm[14] in combination with a probabilistic Hough transform [13] is applied to the RGB image to extract edges from the stairs. The detected edges from the RGB image and the model edges have to be matched together.



(a) Straight stairs



(b) spiral stairs

Figure 2.4: Plane detection for: (a)straight stairs, (b)spiral stairs using scan line grouping technique as suggested by the paper's authors. Each detected plane receives a unique colour. Points not belonging to a plane are marked red. Also it is noted that in (b) the background wall in the spiral stairs have been detected as three separate planes and not as one (source: [38]).

Due to texture and shadows the authors have to handle false detections. For this reason on each observation the robot makes three images, one looking to the left, one to the center and one to the right and find the edges within these three images. After that a RANSAC line fitting algorithm is applied to all lines.

On each new iteration/observation of the camera, a new model is built and the new steps replace the old steps that have similar height. This is how the stair model is updated.

Stair climbing As mentioned in the previous section, the robot detects the planes using the 3D camera and builds the stair model. Then the robot approaches the stairs or is manually placed in front of them. The robot then swings his head left and right to build a scene from three images: left, center and right and update the model using the detected edges. The robot then starts climbing the stairs one step at a time. The climbing process is a pre programmed procedure with a specific leg movement and height. Due to Nao size, the authors have built special stairs for the robot with 7 cm height.

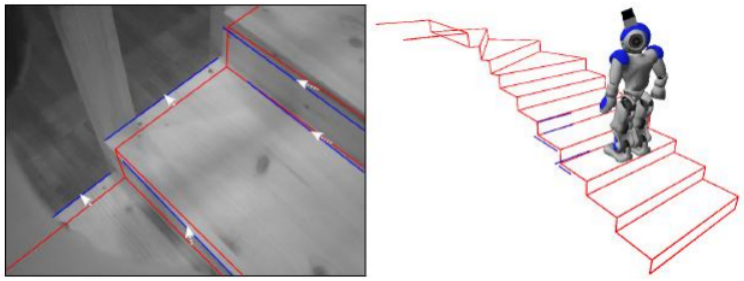


Figure 2.5: Building model with the humanoid robot Nao. Red lines are the model built from plane detection and blue lines are the detected edges. The distance between them is the deviation between model and reality. The model correction approach is simply by building a new model on each iteration and replacing the steps of the old model (source: [38]).

Problems Similar to the approach in section 2.1, this approach also misses the descending of stairs case and also suffers from false detection of edges from the RGB image. Furthermore the authors use stairs with a step height of 7 cm, we in contrast would like to detect standard house stairs with a height of $16 - 18$ cm. Also these is little information available on how the stair model is created, what are the attributes of this model and how is it updated.

2.4 Semantic stairs modelling

Another approach that we find interesting is the work of Schmittwilken and Plümer [44], Schmittwilken et al. [47, 46] which is not related to the domain of robotics but to Geodesy and 3D buildings modelling with concentration among other matters on stairs modelling. The target of the authors research is to be able to describe buildings including their components, facades, stairs etc. using simple entities and attributes. Also they developed specific set of grammar rules for building description.

First of all a straight forward stair model is defined through the following parameters:

- Number of steps
- One origin point
- Length,Depth,Height

Then the stairs must comply to context free grammar rules as following:

$$\begin{aligned} P_1 &: Flight \rightarrow \ \ Riser \ \ Tread \ \ Flight \\ P_2 &: Flight \rightarrow Riser \ \ Tread \end{aligned} \tag{2.1}$$

Also define the following atributes to steps:

$$\begin{aligned} &Flight.steps \\ &Riser.rise \\ &Riser.width \\ &Tread.depth \\ &Tread.width \end{aligned} \tag{2.2}$$

As seen in equation 2.1 the authors are defining the stairs in a recursive manner, so that each flight contains a step plus another flight and so on until the last flight that only contains one step.

To understand the concept of modelling the stairs through grammar, consider the example as shown in fig. 2.7 with a single flight of stairs. The grammar describing this set of stairs is as follows:

$$\begin{aligned} P_1 &: Entrance \rightarrow Stair \ \ Door \ \mid \ Door \\ P_2 &: Stair \rightarrow Flight \\ P_3 &: Flight \rightarrow riser \ \ tread \ \ Flight \ \mid \ riser \ \ tread \end{aligned} \tag{2.3}$$

Huge data sets for whole building facades, flight of stairs and corridors are available. The target is to match the grammar to the data and to reconstruct a stair model. First a RANSAC plane fitting algorithm is applied to the whole data set of points. Further, the close planes are compared to find a so called "meets Perpendicular rule" where a tread meets a riser. Furthermore, steps are found using the

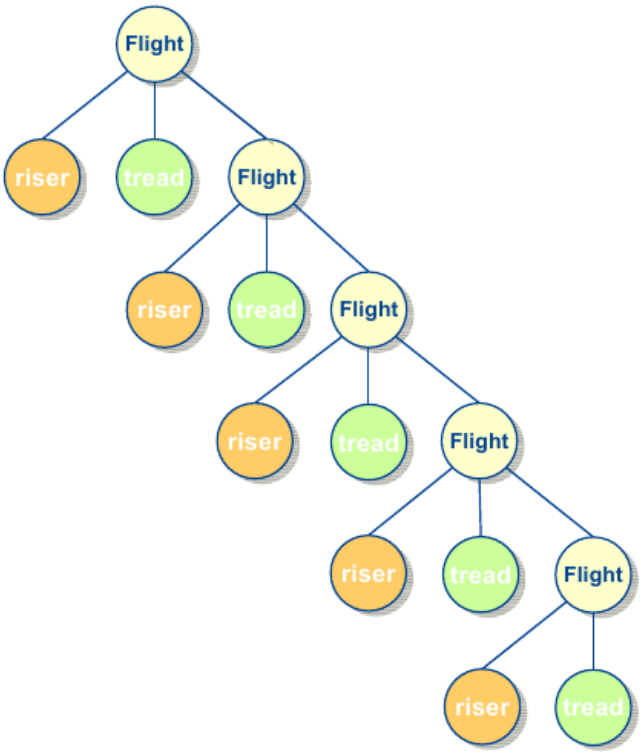


Figure 2.6: Attribute grammar for a flight of stairs as described by Plümer [39]. As seen in the diagram the grammar tree is defined in a recursive fashion where each flight consists of a $tread + riser + flight$ recursively until the last step contains only a tread and a riser (source: [39]).

same approach assuming that the found steps comply with equation: 2.3.

This approach could be problematic if used in our work. First, there is a performance problem since RANSAC plane fitting must be applied to a huge point cloud several times, which shows a very slow performance [43]. But this approach has been extended and enhanced to use a so called "model based estimation" to extract the stair model directly from the set of points without using RANSAC and instead using stochastic methods that investigate the points' distribution on the height and depth axis. This is done in Schmittwilken and Plümer [45], Schmittwilken [43].
The second problem is that the grammar is defined in a way that assumes that a whole point cloud is present from which a stair model could be extracted. This is not the case in our research. During traversing along the stairs, we only have what a Kinect can percept, which is usually only a limited view of the stairs. For this reason if we want to use this approach, we will have to adapt the grammar to our need to allow the definition of yet non existent steps and unknown entities on the staircase.

Figure 2.7: An example of a simple staircase that consists of a single flight of stairs. We can also notice the reference point at the bottom left corner of the flight of stairs as defined by Plümer which describes the origin point in the stairs model. Using this global reference point and the other attributes mentioned in this section, a stair model is defined as a part of buildings modelling (source: [39]).

2.5 Challenges

As we have seen in this chapter, there are several robots and approaches that solve the problem of stair detection and climbing autonomously. We have also presented some problems of these researches and open issues. Revisiting these problems, we are going now to concretize the challenges that we shall address in this diploma thesis:

- Detect standard straight house stairs.
- Handel the case of going downstairs as well as upstairs.
- Combine the model building approach of semantic stair modelling of section 2.4 and the approach of Osswald et al. [38] explained in section 2.3 concerning stairs detection and modelling to adapt to our needs.
- Try to overcome the RGB false edge detection problem mentioned in sections 2.1 and 2.3.
- Find camera's pose and height.

Chapter 3

Stair Detection

3.1 Chapter organization

As can be seen in figure. 3.1, we shall discuss the different approaches we followed to detect stairs. First a plane detection approach shall be presented using scan line grouping and using a Region growing approach as will be explained in section 3.2. Since the camera has been tilted downwards while acquiring the 3D images, we need in the following step to rotate all planes in a way so that horizontal planes in reality are also horizontal in the image and the same applies for vertical planes. For this reason we need to determine the camera's rotation and rotate all planes, so that the normal vectors of the horizontal planes coincide with the gravity vector. When this has been achieved we apply edge detection mechanisms on the input. On one hand we shall obtain the edges from the normal image in section 3.4.1 and then we obtain edges through depth jumps in section 3.4.2.

3.2 Plane detection

3.2.1 Scan Line Grouping

Georgiev et al. [19] use a scan line approach to detect planes. Using an RGBD-image as input, the algorithm makes two passes on the data. A single row in an organized point cloud is called a sub scan. First, it scans each row in the organized point cloud and searches for neighbouring points that might constitute a line segment. Second, using a similar approach to region growing, the lines are scanned from top to bottom fashion using a line segment as a seed region and trying to find neighbouring line segments in the consecutive scans that together with the seed line(s) can form a plane.

Line segment extraction There are several line extraction techniques in the literature, for instance incremental line extraction as in [31] and [51], which uses a line update technique. That means updating the line's normal and calculating the

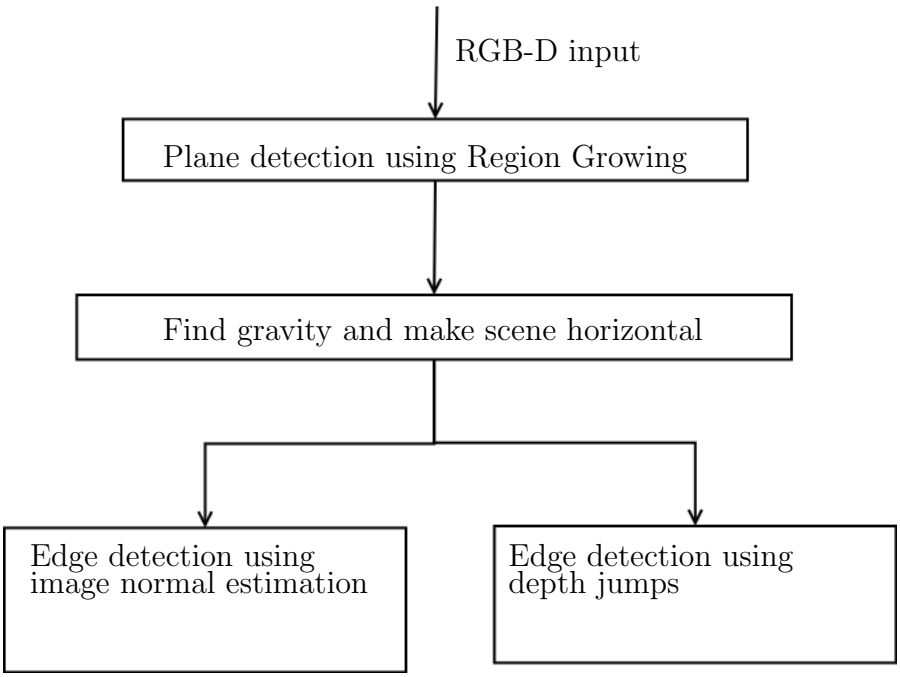


Figure 3.1: The organization of this chapter

Mean Square Error (MSE) of the line segment costs O(1) and hence the algorithm has runtime of O(N). The line segment extraction algorithms works as follows in algorithm 1.

Algorithm 1 Line segments extraction

1: Scan each row of the organized point cloud
2: Find a seed point and create a new line segment with this point
3: Add consecutive points to the line segment as long as they are not too far from one another and still lie on the same line segment
4: If the consecutive point does not belong to the line segment, then stop and go to (2)
5: Repeat until all points in a scan line are consumed, then proceed to the next line

Construct planes from line segments Now that a set of line segments in each subscan (image row) is constructed, the authors use this knowledge to construct planes in a region growing approach as seen in algorithm 2.

Two consecutive line segments from two subscans are coplanar if they:

- do not belong to the same subscan (otherwise they would have been one line instead of two).

Algorithm 2 Planes construction from line segments

1: Iterate over each subscan (image row)
2: Find a line as a seed region and construct a plane
3: Test lines in the consecutive subscan for coplanarity
4: If the line is coplanar then add it to the plane
5: Repeat until there are no more lines to add
6: Go to (2)

- do not intersect.
- lie on the same plane.
- are close enough to each other.

In the algorithm description only the next consecutive line to the plane is considered. In the actual implementation the next 'k' lines are considered for coplanarity, where 'k' is usually less than 10 lines. Due to noise and perception errors in the 3D image, it is not dependable to consider only one neighbouring line.

The implementation of this algorithm has been received from the paper's authors. Although this algorithm is very fast, it is not suitable to detect planes in the case of a staircase as input. The reason is that it fails the collinear and coplanar tests between horizontal and vertical planes of the stairs and detect the vertical plane partially as horizontal plane and vice versa. Figure. 3.2 shows the problem. This erroneous detection leads then to further problems. As a result this algorithm could not be used in this thesis.

3.2.2 Region Growing

As seen in the previous section, the scan line grouping does not achieve our goals. We search for another plane detection algorithm. Holz and Behnke [23] describe an image segmentation and surface reconstruction approach by means of approximate polygon meshing. The algorithm presented aims at surface reconstruction from segmented meshes. What interests us in this approach is the plane segmentation through region growing mechanism. We received the original implementation of this algorithm directly from the paper's authors. The processing pipeline consists of:

1. Calculate mesh approximation from mesh neighbourhood.
2. Use mesh neighbourhood to compute approximate local surface normals.
3. Bilateral filtering to smooth both points and normals.
4. Segmentation based on region growing.

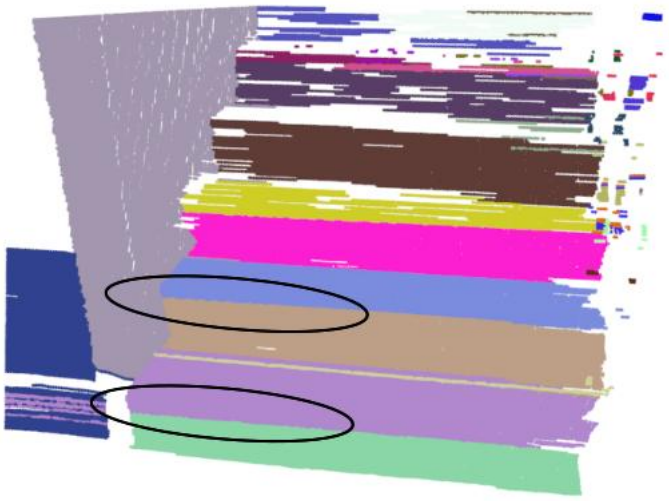


Figure 3.2: As can be seen, each plane takes part of the consecutive plane as its own, which results in imprecise plane detection.

This algorithm requires an organized point cloud structure where a neighbourhood between points is established. If this is not the case, then we can use one of the common approaches to build such a neighbourhood relationship like quadtree [41] or octree [52].

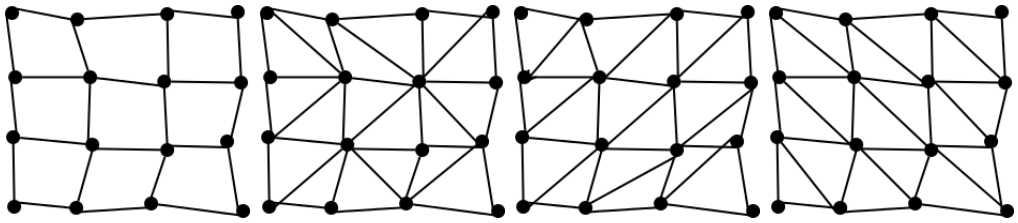
Build a mesh using the neighbourhood structure The target is to calculate the mesh directly from the image neighbourhood structure. So first the algorithm traverses through each point in the organized point cloud (or the constructed neighbourhood) to construct the polygons of the mesh. As seen in fig. 3.3, there are four types of polygons. Let u be the index for rows, v for columns of a point in an organized point cloud:

1. Quad meshes: Constructed by connecting the pixels (u, v) to its direct neighbour in the next row and/or column $p(u, v+1), (u+1, v), (u+1, v+1)$.

2. Fixed left and right cut meshes have the same cuts as quad mesh plus cutting either the quad from top right to bottom left (left cut) or from top left to bottom right (right cut).

3. For adaptive triangulation we make either a left or right cut. Choose the one with the smallest distance between its two end points.

As a post-precessing step remove all vertices that do not belong to a polygon.



(a) Quad Mesh (b) Adaptive triangulation (c) Left cuts only (d) right cuts only

Figure 3.3: Fast approximate meshing of an organized point cloud in the form of a quad mesh (a) and a triangulation (b). Compared to the adaptive approach, triangulations using either only left (c) or only right cuts (d) through the quads can be obtained slightly faster (adapted from: [23]).

Compute surface normal The normal n_i of the point p_i is calculated as the weighted average of normals of the surrounding faces to point p_i.

Bilateral Filtering Usually a point cloud may contain noise due to hardware limitations or environment conditions. This noise might negatively affect obtaining good final results. For this reason a bilateral filter is applied to smooth both points and normals, preserving the neighbourhood structure while doing so. Using the current point structure it is easy to extract the point's neighbourhood from the mesh. A filter is then applied to the point p_i and its normal n_i over its 1-ring neighbourhood, i.e., all direct neighbours to p_i that are connected to p_i by an edge in the mesh in order to smooth each point and its respective normal using its local neighbourhood.

Region Growing-based Segmentation Next the authors present the region growing algorithm as described in several literature. The pseudcode in algorithm 3 describes region growing briefly.

Region Models used for Segmentation Several models have been implemented that behave similarly to to the segmentation algorithms mentioned in the literature like in the work of Harati et al. [22], Cupec et al. [15] and Poppinga et al. [40]. The goal is to check whether a given point belongs to the region model or not. This check is done based on the point's normal deviation from the model and also by checking the euclidean distance to the model. The important parameters of a model are: the minimum cluster size (minimum number of points) and the angle tolerance between points' normals. This is the angle between the new point's normal and some defined normal of the model that varies depending on the model's type. The results of different models are presented in figure 3.4. Here are some model examples:

Algorithm 3 Region Growing Algorithm

```
1: Select a seed point and initialize a region model
2: Add seed points to queue Q
3: while Q is not empty do
4:    P = Q.pop()
5:    if P is comaptible to the region model then
6:       Add P to the region plane
7:       Add P neighbours to Q
8:    end if
9: end while
10: Repeat until no more seed points can be found
```

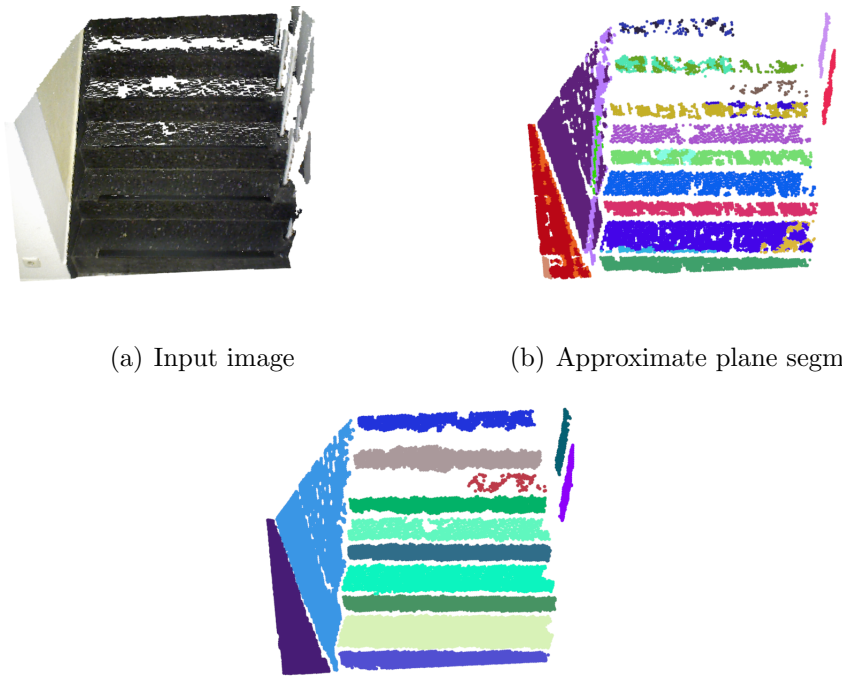
1. Approximate Plane Segmentation: To find out whether a given point p_i belongs to the model or not we check the point's normal n_i against the average normal of the plane for similarity. The average normal of the plane is maintained and every time a new point is added, the average normal is updated. This is much faster than calculating the average normal of all points upon each addition of a new point.

2. Last Normal Segmentation: In this model type, the normal of the given point is checked against the normal of the last added point for similarity. This type of model has shown the best results among the different implemented models (with a wise choice of parameters).

Since last normal segmentation model proved superior, it will be the model type used throughout this research. But regardless of the model segmentation type used, we cannot assume that no oversegmentation will take place in all cases. For this reason we shall see in the next section how a check is done on all planes to find neighbouring planes that have similar orientation and how these planes will be merged together. Please note that the careful choice of parameters for the segmentation model would allow in the worst case for oversegmentation but never for undersegmentation, i.e., no two planes with different orientation will be detected as one. The reason for that is because it is easy to find similar planes and merge planes, but it is much more difficult to find under segmented planes and to separate them from one another. Figure 3.5 shows the result of applying the region algorithm[23] to a 3D image

3.3 Using region growing algorithm to detect stairs

Make the camera horizontal After applying the region growing algorithm [23], we obtain a set of planes of different sizes and orientations. On the first place we are interested in those planes that can constitute a staircase, in other words horizontal and vertical planes. A horizontal plane in reality is detected as horizontal plane if



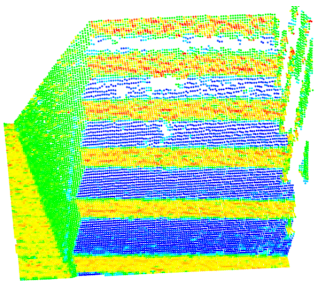
(a) Input image

(b) Approximate plane segmentation

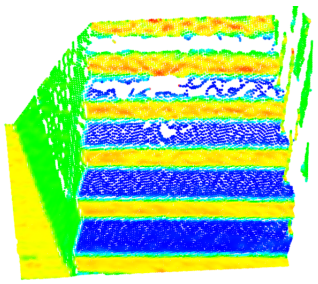
(c) Last normal segmentation

Figure 3.4: Showing two different results for region segmentation models: approximate plane segmentation and last normal segmentation. As can be seen the raw result of last normal segmentation is better than approximate plane segmentation, as it did not over segment the planes and recognized patches with the same orientation as one plane.

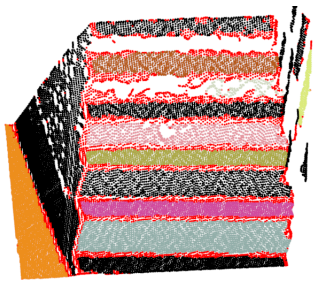
(a) Input image



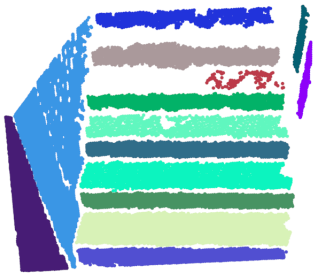
(b) Raw mesh



(c) Smoothed mesh



(d) Initial segmentation



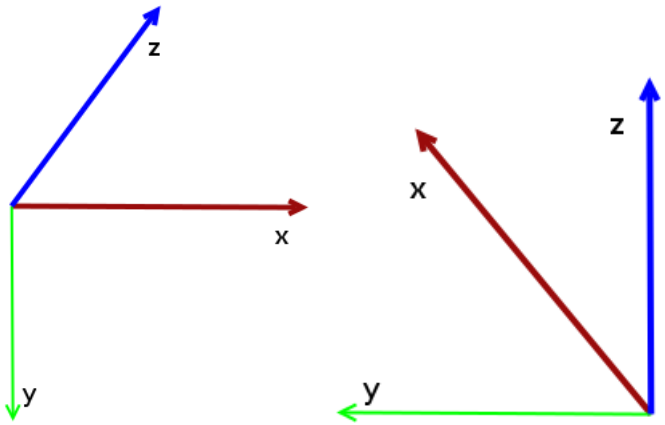
(e) Segmentation without common points

Figure 3.5: The figure shows the result of plane segmentation using a region growing approach. The input image(a) is used to construct the raw mesh(b) which is then smoothed in (c) and finally plane segmentation is applied(d), where the red points are those that belong to multiple planes and (e) are the planes removing these points.

and only if the camera is held in a horizontal position, i.e., not tilted downwards. This is of course not practical as most of the horizontal parts (treads) of the steps won't be seen, and so a camera has to be tilted downwards. For this reason we need to find the camera rotation and rotate all planes in the opposite direction so that the normals of the horizontal planes coincides with the gravity vector.

It would be useful at this stage to calculate some extra attributes of the planes. The normal to the plane is calculated as the average normal of all point normals on the plane. The center of mass point is also calculated as the average point of all points on the plane.

Before we start we will explain some technicalities about the camera and world coordinates. ROS and PCL use what we shall call a world coordinate system while the Kinect camera uses a camera coordinate system. Both are right hand coordinate systems. The camera coordinate system is where the x-axis points to the right, y-axis downwards and z-axis forward. A world coordinate system is where the x-axis is forwards, y-axis to the left and z-axis upwards. The point cloud received is in the camera coordinate system and so we rotate it to the world coordinate system as shown in figure 3.6.



(a) Camera coordinate system (b) World coordinate system

Figure 3.6: Transformation from camera coordinate system in figure. 3.6(a) in which the points obtained from Kinect are represented, to the world coordinate system in figure. 3.6(b).

Below is the algorithm on how to detect the rotation of the camera:

Filter planes We are mainly interested in the planes that could be candidates for a staircase, primarily horizontal and vertical planes. For this reason, planes that are

Algorithm 4 Make the scene horizontal

1: Calculate the angle between each plane's normal and the normal to the ground plane, that is $(0, 0, 1)$
2: Find the normal with the smallest angle to normal $(0, 0, 1)$
3: Find the normal(s) with rotation within 10% from the smallest normal, i.e., planes with similar rotations
4: Calculate the weighted average normal and weighted average angle, where normals close to the minimum have high weight and vice versa
5: Calculate the rotation quaternion from the calculated weighted average normal and angle. This is equal to the camera's rotation
6: Rotate all planes using the calculated quaternion

neither horizontal nor vertical are removed. Furthermore planes with small number of points (less than 100 points) are also removed, since they could be a source of errors in future steps.

Calculate bounding box Another interesting attribute that the plane should have is the dimension of the planes. For horizontal planes this is (Length x Depth). For vertical planes this is (Length x Height). First we should calculate the convex hull of the plane. There are several algorithms that solve the convex hull problem, for instance the Graham Scan [21] and Chan's algorithm [32].

Now that a convex hull is available, the next step would be to find the bounding box of the plane. The bounding box consists of four points: top left, top right, bottom left and bottom right. We are interested only in calculating the bounding box of the horizontal and vertical planes. When the bounding box points are available, it is then easy to calculate the length, depth and height of planes. Algorithm 5 explains as an example how to find top left and bottom left points of a plane.

This algorithm is analogous for top right and bottom right points except that we use 20% of the sorted points with the lowest y-value.

Calculate the planes' dimensions The plane length is calculated as:

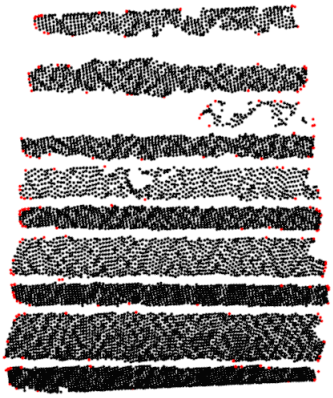
$$Length = max(\ (topLeft.y - topRight.y), (bottomLeft.y - bottomRight.y)\ ) \quad (3.1)$$

If the plane is a riser we calculate the height as follows:

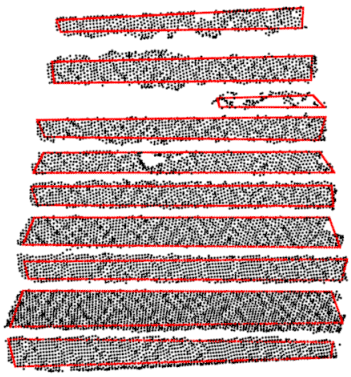
$$Height = max(\ (topLeft.z - bottomLeft.z), (topRight.z - bottomRight.z)\ ) \quad (3.2)$$

On the other hand if the plane is a tread then we calculate the depth:

$$Depth = max(\ (topLeft.x - bottomLeft.x), (topRight.x - bottomRight.x)\ ) \quad (3.3)$$



(a) Convex hull



(b) Bounding box

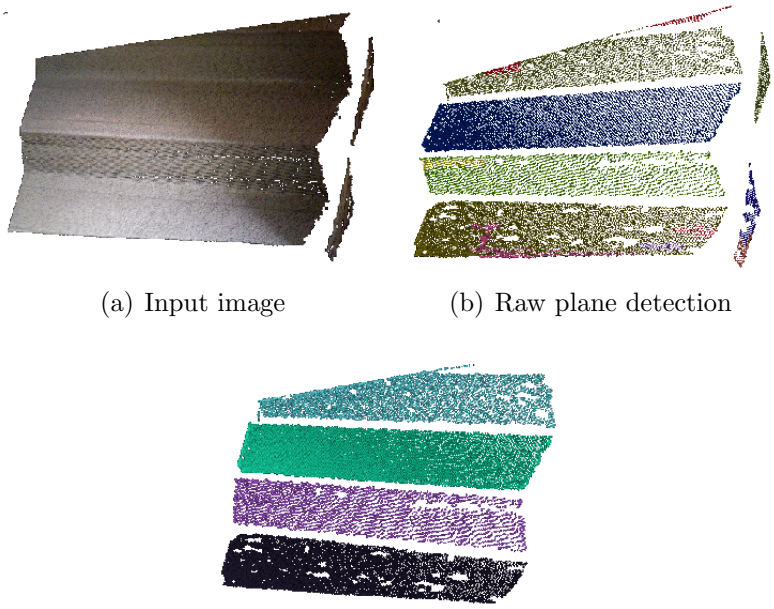
Figure 3.7: The result of applying convex hull to each riser and tread (a) and the bounding box calculated from the convex hull. The points between the four bounding box points are only generated for visualization purposes but they are not needed by the detection algorithm.

Algorithm 5 Calculate Bounding Box

1: Sort the points in ascending order along the y-axis (left/right)
2: Take 20% of the points with the highest y-value(most left points)
3: If the plane is horizontal then sort the points ascendingly along the x-axis (depth), or if it is vertical then sort points along the z-axis (height) into an array and call it pointsZAxisLeft. For this example we will assume it is a vertical plane
4: To find the top left point we consider the last point in the array pointsZAxisLeft (highest z-value)
5: If the difference in y-component between this point and the left most point is less than 1 *cm* then we have found the top left point, otherwise consider the next point in the array
6: Do the same for the bottom left point but start with the first point in the array pointsZAxisLeft (the one with the lowest z-value)

Merge similar planes The region growing algorithm does not always produce perfect planes. In some cases we find planes that are close to each other, have the same orientation and should definitely be the same plane but they are not, the goal here is to merge them and for that we shall use a histogram.
We shall create two histograms, one for horizontal and one for vertical planes. A histogram has several bins that spans from the minimum to maximum height for horizontal planes, and minimum to maximum depth for vertical planes. We start by adding planes to histograms, for example a horizontal plane is added to a bin in the histogram, where the plane's height lies between min and max range of the bin; a bin size is set to 5 *cm*, i.e., height or depth between two neighbouring treads or risers in order to be merged has to be less than 5 *cm*. Next step is to merge planes in the same bin if and only if distance between their borders is small (less than 50 *cm*). We can use the bounding box of each plane to find the distance between them.

In some cases a draw back emerges from histogram, when two close planes are assigned to two neighbouring bins due to discretization. For example consider the bins with range $[20, 25[$, $[25, 30[$ and two planes with height 24.5 and 25.5 *cm* respectively. These planes belong to two different bins but the height between them is less than 5 *cm* and should be merged, so we merge them.
Such an approach is good in solving the discretization limitation of the histogram but it imposes another problem. If we start merging planes from neighbouring bins we might accidentally cause a series of merges of consecutive bins as in a domino effect. To avoid that we set a rule that a plane from a given bin can be merged with a plane from a neighbouring bin exactly once. After that a plane cannot be further merged. Plane merging can be seen in figure 3.8.



(a) Input image (b) Raw plane detection

(c) Merged planes

Figure 3.8: Raw RGB-D image(a). Detected planes after applying region growing with some small and noise planes(b). Planes after filtering, merging planes and removing planes with two few points(c).

Calculating camera's height So far we know the camera's rotation and we have the detected planes as well. Our goal is to find the camera's height. We sort all planes according to depth and take the plane with the smallest depth value, i.e., the nearest plane. We consider this depth value as the camera's distance to the stairs or as side 'C' in figure. 3.9. The distance 'C' can be found using the Hessian Normal Form as follows:

$$D = \hat{n}.x_0 + p \tag{3.4}$$

Where in equation 3.4 'D' is the distance scaler value, x_0 is a point on the plane, $\hat{n}$ is the normal to that point and '.' is the dot product. The p is the vector to the origin point, which is the camera in our case. The camera has always the coordinates $(0, 0, 0)$ and so the Hessian Form in our case becomes:

$$D = \hat{n}.x_0 \tag{3.5}$$

Calculating the camera height (side 'B' in figure. 3.9) is then straight forward using right angle triangle equation to calculate side B:

$$B = C \times \cos(a) \tag{3.6}$$

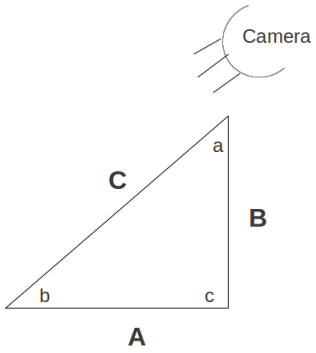


Figure 3.9: The triangle shown is used to calculate the camera's height. The camera is located at angle where sides 'B' and 'C' intersects and the angle between them is calculated as 'a', which is the camera's rotation. The length of 'C' is the camera's distance to the first detected plane. The camera is represented as side 'B'. The Hessian Normal form using these information is used to find the camera's height.

3.4 Edge detection

The state of the art has shown that stair detection through plane detection might not be enough. In some cases it happens that planes are not detected at all or detected but the size is incorrect. Also in case of going downstairs, no vertical planes can be detected and also horizontal planes are not detected correctly. This is due to camera perspective related to the stairs. Using edge detection could introduce extra information to stair detection.

The simplest approach is to apply one of the common edge detection algorithms ,like canny edge detection, to the RGB image. Unfortunately edge detection on RGB images does not yield good results. Experiments have shown that, while tweaking the parameters for canny edge detection algorithm, either too many edges are detected or too few, and hence this leads either to too many planes or too few planes. The main reasons for that are: the steps we tested have the same color or they are dark. Under these conditions it is very difficult to detect edges. Using the organized point cloud we can apply two other approaches to exploit this organized structure:

1. Edge detection from the Normal image.

2. Edge detection through Depth jumps.

3.4.1 Edge detection from normal image

Calculating normals to each point of the organized point cloud would enable us to create an RGB image where similar points' normals have similar color. This way we obtain an RGB image with various colors from which an edge detection algorithm could be applied as shown in algorithm 6. The calculated normal image can be seen in figure 3.12(b).

The algorithm is as follows:

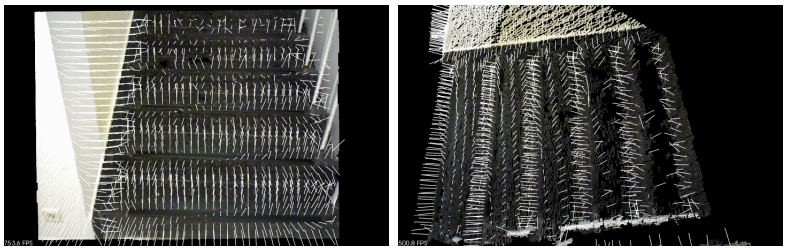
Algorithm 6 Edge detection from normal image

1: Using the organized point cloud, find the normal vector to each point using an image normal estimation algorithm
2: Project the organized point cloud with normals to an RGB image
3: Apply Canny Edge detection algorithm
4: Find contours
5: Apply Hough transform to the contours
6: Filter lines whose orientation varies significantly from the dominant orientation
7: Calculate distance to the origin point and the orientation of each line segment
8: Merge lines that are close and with similar orientation
9: Remove short and non horizontal lines

Normal Estimation

In an organized point cloud each point p_i is assigned to a pixel (u, v) in an organized structure. There are several algorithms in the literature that exploit this organized structure to calculate the local normal of each point depending the neighbourhood of that point. The approaches presented in [24, 25] shows how to calculate the normal to the surface on which each point lie using the local neighbourhood of this point. There are two possible kinds of neighbourhoods: either a k-nearest neighbours, where for each point the nearest k neighbouring points are calculated, or points lying within radius 'r' from the given points. To find the neighbours of a point, one could use a search method like constructing a kd-tree from the point cloud or to use the organized structure of the point cloud. The first method is accurate but lacks speed, where the second method is less accurate but very fast since no search structure need to be constructed. The result of applying the mentioned normal estimation method is shown in fig 3.10. We used the algorithm implementation offered by the PCL library.

Project the organized point cloud with normals back to an RGB image In an organized point cloud each point has an index (column, row). The goal of this step is to calculate an RGB color for each point based on the three components of the normal vector of each point. Each component of the 3D normal vector is



(a) (b)

Figure 3.10: Frontal(a) and top view(b) to the estimated normals of a point cloud to the stairs.

between $[0, 1]$ and is translated to a value between 0 and 255. Equation 3.7 explains how to translate a value in an older range to a new value in another range. This way we obtain an RGB color vector with three components representing the color of the point. It is obvious that points lying on the same surface would have almost similar normals and though would yield almost a similar color as seen in fig 3.12(b). To transform each value from $[0, 1]$ to $[0, 255]$ we use the following equation:

$$newValue = \frac{(OldValue - OldMin) \times (NewMax - NewMin)}{OldMax - OldMin} + newMin \quad (3.7)$$

Hough Transform

The hough transform technique is used to find primitive shapes that can be expressed by a mathematical expressions, like lines, curves and ellipses in binary images [16]. A binary image is a black/white image. For one white pixel in an image there can be infinite lines passing through this pixel. Now if one or more of these lines also pass through other white pixels, this is an indication that a line might exist in the image that pass through the selected white pixel and other white pixels. This is the principal of Hough Transform of finding straight lines. The hough algorithm implementation that we used can be found in OpenCv.

A line segment in 2D space can be expressed mathematically by the following equation:

$$y = a \cdot x + b \quad (3.8)$$

where 'a' is the slope and 'b' is where the line intersects the y-axis and (a, b) are the coordinates of any point lying on the line segment, i.e., this equation applies to all points of the line segment. The problem with that representation is that it is an infinite line, i.e., points in an infinite space will still be valid for this line. For this reason it is desirable to find another expression to describe a line segment with

boundaries. To achieve this an angle and a distance parameters are used to describe a line segment instead of a slope and an intersection.

The parameters 'ρ' (also called rho) which is the distance to the origin point (0,0) and 'θ' (theta) which is the angle between the x-axis and the 'ρ' vector. Having this in mind we can reformulate the line segment equation in 3.8 to be:

$$y = -\frac{\cos(\theta)}{\sin(\theta)} \times x + \frac{\rho}{\sin(\theta)}$$

$$\texttt{which can be rearranged to} \tag{3.9}$$

$$\rho = x \times \cos(\theta) + y \times \sin(\theta)$$

In contrast to the equation 3.8 the parameters are finite where $\theta \in [0, 180]$ and $\rho \in [-D, D]$ where D is the diagonal of the image. All points with (x, y) coordinates must fulfill this equation. Using this knowledge we can transform each line into a single point in the parameter space using the pair (θ, ρ). This is called the hough space.

Typically this procedure is applied on the result image of canny edge detection or similar binary image. Canny edge detection is explained in the literature in [14] since then the original algorithm has been subjected to several enhancements. We use the implementation for this algorithm from the OpenCV library.

Hough transform is then applied to the white edges in the edge detected images as in fig 3.12(c) after applying hough transform on the "detected edges" image the lines that go through the maximum number of white pixels can be found as seen in fig. 3.11. The result of the hough transform is stored in a 2D matrix, with one dimension representing the distance (ρ) and the other dimension representing the theta (θ) and each entry in the matrix is holding a value to tell how many lines lie on the corresponding (θ, ρ) pair. In this case it is easy to find the entries that hold the highest values and these would be the hough lines we wanted to detect.

Detect edges from normal image

Although applying canny edge detection on normal image obviously yields much better result than on a simple RGB image, there are usually a lot of noise and small edges that lead to faulty hough line transformation calculations. There are is no unified set of tweaked parameters that would suite all inputs to produce an acceptable result. An approach that showed an enhancement was to use a "find contour" procedure using a border following mechanism [49].

Finding contours helps smoothing the input to hough line transform by joining edges close to each other and closing the gaps between neighbouring lines that should have been detected as one line. This step is not always essential but it is recommended to enhance the input to hough transform.

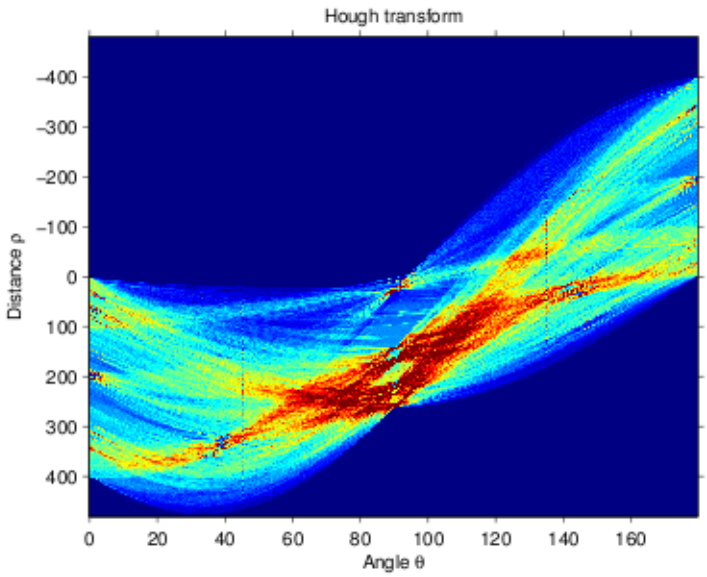


Figure 3.11: Sinusoidal curves passing through the white pixels. Red points are the pixels with the highest number of lines passing through them (source: [1]).

Calculate Distance to origin point and orientation The randomized hough transform as implemented by openCV delivers lines as two points in 2D space representing the start and end point of a line segment. Our goal is to transform each set of points into a line with the pair (ρ, θ) denoted as the polar coordinates. To do that we first need to calculate some attributes of the lines mainly the center point as well as the normal vector. For visualisation reasons we are also going to generate a set of points representing the line segment between the start and end point.

Let us start by generating points between the two end points of the line, which we will denoted as $P_1(x_1, y_1)$, $P_2(x_2, y_2)$ as seen in algorithm 7.

Next step is to calculate the centroid and normal:

$$\vec{c} = \left(\frac{maxX + minX}{2}, \frac{maxY + minY}{2} \right). \tag{3.10}$$

In order to find the normal to the line segment, we first have to find the slope:

$$\begin{array}{rcl} \texttt{Let } dx & = & x_2 - x_1, dy = y_2 - y_1 \\ normals & = & (dx, -dy) \texttt{ and } (-dx, dy) \end{array} \tag{3.11}$$

After acquiring the pair: the center and normal vector to the line segment in a 2D space it is time to calculate the pair (ρ, θ). Let us start by calculating θ in degree

Algorithm 7 Sample line segment from start and end points

1: add (start,end) points to line model
2: find minX, maxX, minY, maxY from (start,end) points
3: distance=$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
4: xStepSize=$\frac{(maxX - minX)}{distance}$, yStepSize=$\frac{(maxY - minY)}{distance}$
5: point=P_1
6: **while** point != P_2 **do**
7: point.x += xStepSize
8: point.y += yStepSize
9: add point to line model
10: **end while**

as the angle of rotation of a line segment. The perpendicular normal to a horizontal line segment in 2D is $(1, 0)$ and is denoted as 'd' and the normal of a given line segment as $n = (n_x, n_y)$.

$$\theta = \cos^{-1}\left(\frac{d.n}{||d|| \times ||n||}\right) \tag{3.12}$$

To calculate 'ρ' as the distance to the origin point, we use the hough line transform as seen in equation 3.9.

Line filter As mentioned earlier we use an image normal estimation algorithm [24] to generate the normal image. Unfortunately it suffers from inaccuracies when calculating points normals on a far distance. This is clearly seen in fig. 3.12(b). It is then inevitable that false/faulty edges will be detected and will produce a partially faulty output. For this reason we decided that edges further than a given threshold from the camera, will be discarded as seen in fig. 3.12(f). This is done using the 'r ' component of the line segment. Non horizontal lines are also removed. i.e., lines with rotation $\theta \geq 45°$.

Merge line that are close and with similar orientation The idea of this approach is to allocate all lines into a histogram according to two criterion: (1) distance to the origin and (2) line orientation; in hough space terms this is the pair (ρ, θ). A histogram is designed as a two dimensional array (Matrix), One dimension for holding the distance 'ρ' component and the other dimension for holding the orientation θ. Each bin has a minimum and maximum 'ρ' and θ.

To create the histogram, we iterate through the available line segments and search for the minimum and maximum ρ and θ. The rows and columns range will span across the minimum and maximum ρ and θ. Where the numbers of rows and columns are defined in the expression 3.13:

$$\begin{aligned} \texttt{number of rows} &= \frac{max(\theta)-min(\theta)}{\theta\_threshold} + 1 \\ \texttt{number of columns} &= \frac{max(\rho)-min(\rho)}{\rho\_threshold} + 1 \end{aligned} \tag{3.13}$$

Where $\theta\_threshold$ is the maximum orientation deviation allowed between two lines and $\rho\_threshold$ is the distance threshold. After testing and tweaking a 'θ' value of 3° and an 'ρ' value of 5 pixels showed good results. In general small threshold values for both parameters proved superiority over large values.

It should be expected that the set of edges that constitutes a staircase should have similar orientation. For this reason we check the number of lines in each column along the orientation axis, and mark the column with the largest number of lines as the the dominant orientation of lines. We are mainly interested in histogram column with the dominant orientation and its close neighbours.
The next step is to merge the lines in each bin together This works fine as lines with similar orientation and similar distance to origin are probably the same line. The next step is to check whether lines from neighbouring bins could also be merged together. Similar thresholds are applied when merging lines from the same bin to merging lines from neighbouring bins. For example if the rotation span in each bin is 5°, then two lines from two neighbouring bins can only be merged if their absolute θ difference is less than 5°. To prevent a line segment being involved in a chain of merges to the consecutive neighbouring bins we set a rule that a line segment shall be involved in at most one merge operation with lines from neighbouring bins. "Locking" line segments in this way prevents the merging propagation of unrelated line segments. The result of extracting edges from the normal image can be found in figure 3.8.

Now that edges are detected in a 2D image, it is time to project the detected edges back to the 3D organized point cloud. This is trivial since it is already known which line segments are there in 2D images as well as the corresponding image pixels to that line. Projecting the line segments from 2D to 3D requires obtaining each point from the 3D whose row and column indices corresponds to the pixel's row and column in the 2D image. This way we project the the detected edges to sets of 3D points.

3.4.2 Edge detection through depth jumps

Let us in the beginning tell the motivation behind this technique. As we mentioned earlier that we are detecting edges from a normal image since edge detection from RGB images does not yield good results due to shadow, textures, etc.
The advantage of a normal image is that it makes use of the normal deviation between points that lie on different orientation planes to detect edges, i.e., as a pre

requirement we must have changes in normals, for example the presence of horizontal and vertical planes. This assumption works well when going upstairs, as both tread and riser are present. But this approach fails when going down the stairs as the riser part (the vertical plane) can not be seen and as a result of the absence of the vertical planes, a normal image is not a good option. In driving downstairs an interesting feature in the depth image is the "*depth jumps*". A depth jump simply means a significant change in the depth value between two neighbouring points in an organized point cloud. For example take two points that are neighbours in the organized point cloud and lie on the same plane then most probably they have small depth difference. On the other hand take two neighbouring points in an organized cloud, which belong to two different planes, e.g. edge points. These two points will have significant change in height, depth values or both. A good example to that case is going downstairs where occlusion occurs for vertical planes and there are depth jumps between neighbouring points.

In the case of going downstairs, part of each tread is covered by the previous tread due to the camera perspective. That means if each tread has two edges one near and one far, then the near edge will always be further than it should be in reality, since it is partially covered by the previous tread. This situation can be seen in figure. 3.13.

Points in an organized point cloud are organized into a matrix. The idea of the algorithm is to go through all points and compare each point to its three next direct neighbours, so the point $p(i, j)$ is compared to $p(i + 1, j), p(i, j + 1), p(i + 1, j + 1)$. If the depth or height change between the two points is large, then this is a depth jump and the two points are marked. The next step is to extract edges from the depth jump points. We do this as follows:

Algorithm 8 Extract line segments from depth jumps cloud using RANSAC line fitting

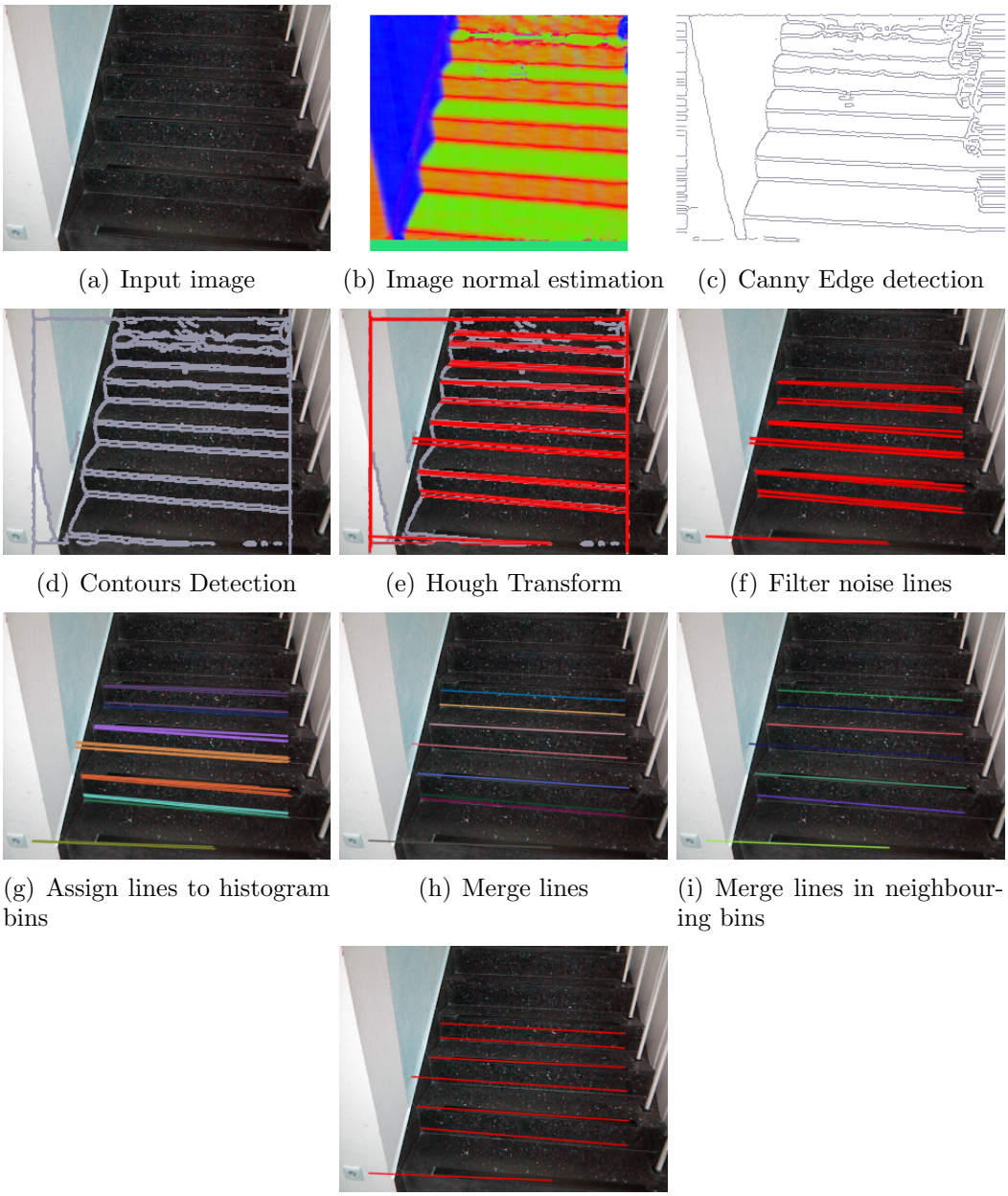
1: **while** depth jumps cloud is not empty **do**
2: Apply RANSAC line fitting to the depth jumps cloud to extract a line segment
3: Subtract the points of the line segment from the cloud
4: **end while**

Also here as with the normal image, the extracted lines should be filtered, where non horizontal lines as well as too short lines are removed. Furthermore using a histogram, lines with similar orientation that are collinear are merged together. Fig. 3.14 shows each step of this algorithm.

RANSAC RANSAC has been proposed by Fischler and Bolles [17] in 1981 as an approach to find the parameters of a geometrical model M described through a mathematical expression using a set of data that contains a large number of outliers. To understand the meaning of outliers imagine that we have a point cloud of a plane and we have a mathematical formula that describes this plane. In this

case all points in the point cloud that are close enough (within a given threshold) will be part of the plane and hence considered as inliers. While all points that are outside the model are called outliers. The target of this algorithm is to estimate the parameters that best describe the data, i.e., a model with the maximum number of inliers. The algorithm contains mainly two steps: hypothesize and test. These two steps are repeated in an iterative fashion to find the best model. RANSAC stands for *RAN*dom *SA*mple And *C*onsensus which would mean something like consensus through selecting random samples.

In the hypothesis step minimal sample sets (MSSs) are selected randomly from the input point cloud and the model parameters are computed through elements of MSS. The cardinality of MSS is the smallest set sufficient to find out the model's parameters.
In the test step the entire dataset is checked to see which elements are consistent with the model that has been calculated through the parameters estimated in the first step (Hypothesis). The set that constitutes the consistent elements is called the consensus set (CS).
The algorithm terminates when it is most unexpected that better elements will be found than the existing ones. The result of applying RANSAC line fitting to the depth jumps cloud can be seen in figure 3.14(c).



(a) Input image (b) Image normal estimation (c) Canny Edge detection

(d) Contours Detection (e) Hough Transform (f) Filter noise lines

(g) Assign lines to histogram bins (h) Merge lines (i) Merge lines in neighbouring bins

(j) Final result

Figure 3.12: Raw RGB image (a). Normal image,where the colour of each pixel is calculated from the normal of the point in the organized point cloud (c). Result of applying contour detection (d). Apply hough transform to contour lines (e). Filter out the vertical lines, lines whose orientation is far from the dominant orientation of most lines, short lines and around 15% of the upper lines since they are often faulty (f). Assign lines to histogram bins based on the similarity of two parameters: 1- orientation, 2- distance from the origin point $(0, 0)$. (g). Merge lines of of the same bin if and only if they are closer to each other than to another line segment in a neighbouring bin. (h). Merging lines of neighbouring bins (i). Final result of edge detection using image normal as an input(j).

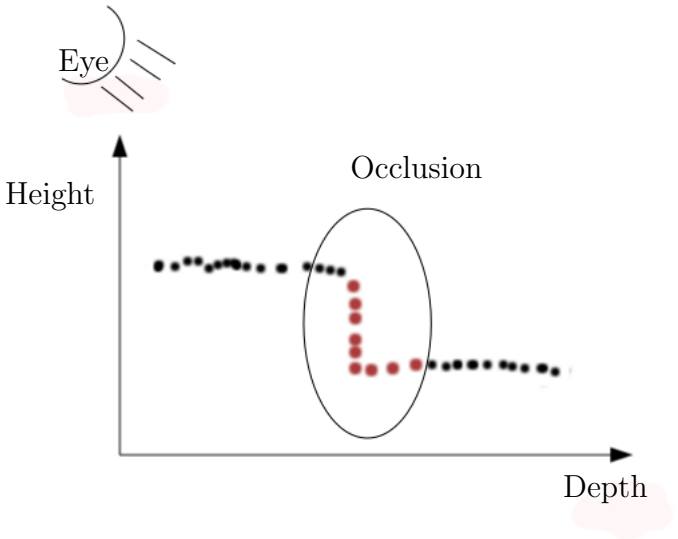
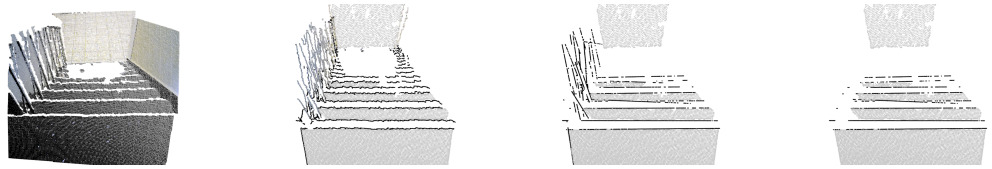


Figure 3.13: A side perspective to the stairs when going downstairs. Black points are what is seen by the camera, while the red points represent the hidden part. As can be seen there is a depth jump between the last point on the higher step and the first point on the lower step, although in an organized point cloud they are neighbouring points (an organized point cloud means that each point belongs to a row and a column (i, j) and hence a neighbouring point means a point in a neighbouring row or column not a neighbour in the sense of "small euclidean distance"). Near edge on the lower tread gets very close depth value to the far edge on the higher tread.



(a) Input image (b) Depth jumps cloud (c) Ransac line fitting (d) Filtered lines

Figure 3.14: The input image (a) is given, where the depth jumps points are extracted and coloured in black (b). RANSAC line fitting is then applied to the depth jumps cloud (c) to extract line segments and finally a filter is applied to remove vertical lines and short lines (d).

Chapter 4

Model creation

4.1 Chapter Organization

As seen in the previous chapter 3, planes and edges have been detected. Now it is time to make use of these detection in order to build a meaningful model that describes the stairs as mentioned in chapter 1. This chapter is organized as in figure 4.1. First we take a set of edges and create planes out of them. Second a plane merge between the planes obtained through plane detection algorithm and the newly created planes take place. After that the steps are constructed from the set of available planes. Finally the steps are organized in a meaningful way to construct an initial stairs model.

We shall make some assumptions about the stairs before trying to model them. The stairs are:

1. horizontal,
2. standard undamaged house stairs and
3. at least partially seen.

In the state of the art section 2.4 we mentioned the disadvantages of Plümer's approach . Mainly the grammar assumes the presence of the complete point cloud. In our case it is not possible to have such assumption and actually it is most probably that the point cloud is incomplete. Also we need to take into account, when defining the models, the perspective of the camera and consider the cases of "going upstairs" and "going downstairs". So let us redefine and introduce some entities to Plümer's grammar to adapt it to incomplete point cloud.

As mentioned earlier we are interested in using the stairs modelling approach as mentioned in section 2.4 but with some extensions so that the grammar is also capable of describing incomplete data sets. Let us describe the two possible camera perspectives as seen in figure. 4.2 in the case of incomplete point clouds.

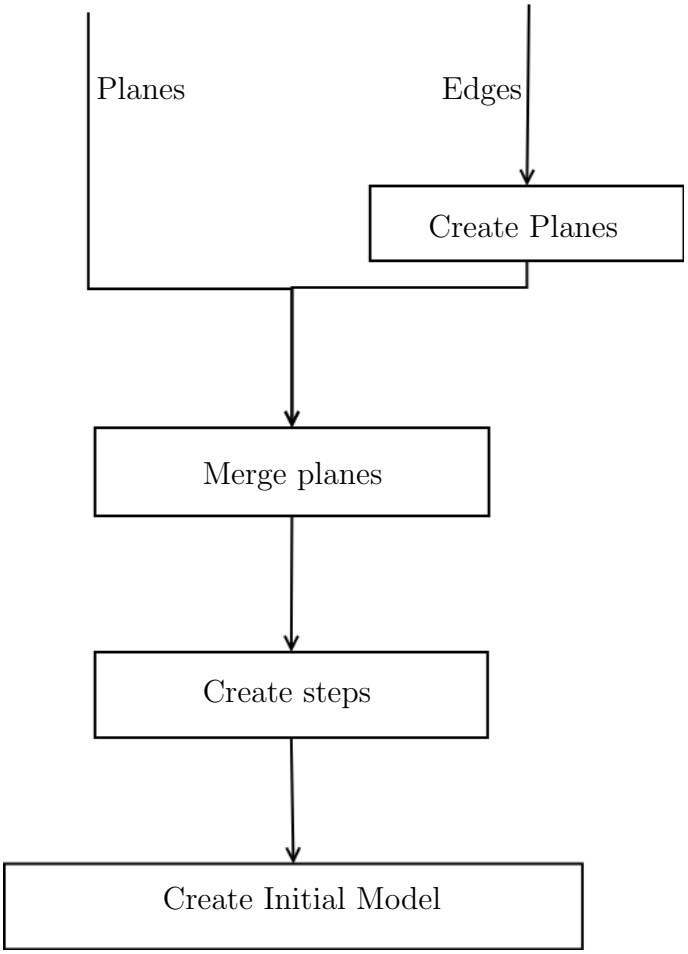


Figure 4.1: The organization of this chapter

(a) Going upstairs (b) Going Downstairs

Figure 4.2: Camera perspective of going upstairs (a) and downstairs (b).

1. Case: Going upstairs: If the camera is moving upstairs, it is expected that vertical planes facing the camera will be easily detected while few or no horizontal planes will be detected, depending on the camera's tilting and its height.
   A step is considered as so if and only if it has a predecessor step or landing, i.e., it is reachable. A step could also be connected to either a successor step or landing.
   An intermediate step is a possible step that has a predecessor. It could be a normal step or a landing. It is now yet known.
   The distance between the step in question and its predecessor should lie beneath a threshold in terms of height and depth.

2. Case: Going downstairs: While going downstairs, there's no possibility to detect vertical planes, but only horizontal. In this domain we define a step as a horizontal plane that has a predecessor, which is either a horizontal plane or a landing and a successor, which is either a horizontal plane, a landing or an intermediate step.

4.2 Stairs production rules

A flight of stairs has the following entities:

1. Step: A step consists of a tread and a riser. Small gaps between them is tolerated (due to camera position, movement, ill results, etc.). In case of going upstairs it is possible to see both the tread and the riser, or only the the riser for the higher steps. In case of going downstairs it is only possible to see the tread as the riser is hidden.

2. Predecessor: Each step has a predecessor. This predecessor is the entity prior to the current step and it could be a step a landing or a step.

3. Sucessor: A successor to the step is either a landing a step or the so called intermediate step.

4. Intermediate step: Is used to denote an unknown step. An intermediate step could be a step or a landing, but the camera is yet not so far ahead to decide which one of the two is the right one.

After defining the entities that constitutes a flight of stairs, we are ready to build the production rules that define the relationship between these entities in the grammar shown in figure. 4.3 to adapt to our needs. The drawbacks of the original version of the grammar have been discussed in section 2.4.

P1: Flight → Landing Flight Landing | Step Flight Landing | Step Landing

P2: Step → Predecessor Riser Tread Successor | Predecessor Tread Riser Successor

P3: Predecessor → Landing | Step

P4: Successor → Landing | Step | Intermediate step

P5: Intermediate step → Landing | Step

Figure 4.3: Our own defined grammar. This grammar is modified from the original definition of Plümer's grammar in section 2.4 to include Landing (P1) and predecessor and successor (P2), definition of predecessor (P3) and successor (P4) and definition of an intermediate step (P5).

4.3 Create steps

Having the knowledge in mind about the modified grammar, we would like to define the so called local model and global model, this could also be called initial model and incremental model. What is meant is that we use the first observation of stairs as the initial model and we update that model along while moving with the camera and receiving new input 3D images.
The first goal we have is to build an initial model using the knowledge we gathered regarding the detected edges and planes. To combine the knowledge from planes and edges, these two entities should be compared together. To do that we need first to build planes out of the detected edges to make a comparison between planes.

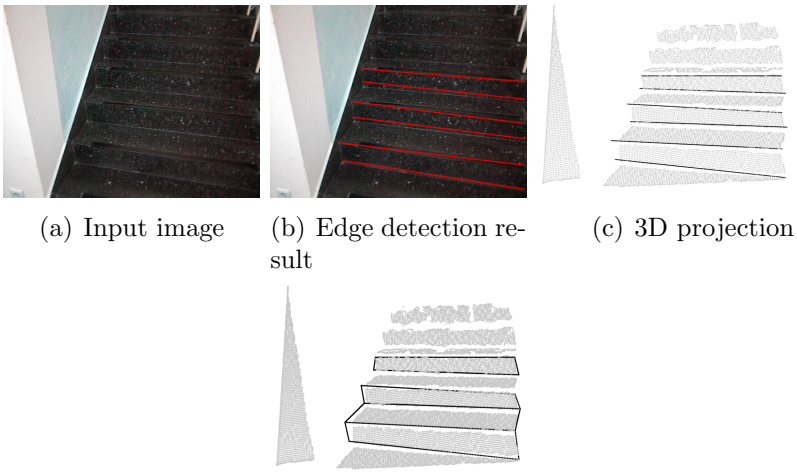
Create planes from edges From the set of edges extracted at the edge detection step in section 3.4, planes should be created and merged to the existing planes. To create planes from line segments we first sort the line segments according to height. Next, we iterate through lines, compare each line to its closest neighbour line segments for possible horizontal or vertical plane existance. Two line segments form a horizontal or vertical plane, if:

- their end points overlap along the horizontal axis (y-axis in world coordinates),
- they do not intersect,
- they have almost similar height (horizontal plane) or similar depth (vertical plane), threshold allowed between lines is $5cm$ and

- they are not vertical (steps contain only horizontal edges).

Also obviously if two line segments form a horizontal or a vertical plane, then one of them is a concave edge and the other is a convex edge. Specifically in a horizontal plane, the far line is concave and the near line is convex; while in a vertical plane the higher line is convex and the lower line is concave. Each line, when used to create a plane is marked as either concave or convex. A line segment can be an edge of at most two planes, where one of them is horizontal and the other is vertical.

Each line segment contains two extreme points, start and end point of a line segment. A total of four extreme points are used to find the bounding box of the plane. We call these created planes "*plane models*" to differentiate them from planes that contain 3D points, where planes created from edges do not contain any points but only a bounding box.

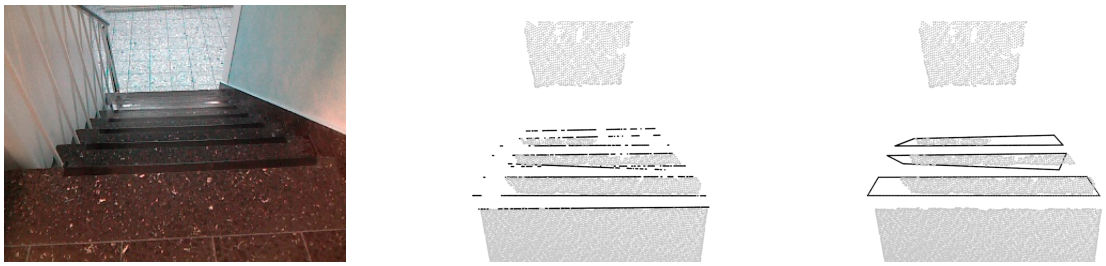


(a) Input image (b) Edge detection result (c) 3D projection

(d) Plane reconstruction

Figure 4.4: Creating planes from detected edges. The stairs in (a) are given as an input, where the edges in the 2D image are calculated in (b). Then each pixel of the lines is reprojected as a 3D point in the organized point cloud in (c) and finally in (d) the planes are reconstructed from edges. The stairs in the background of (c) and (d) are the result of plane detection the correspond edges are added to each image as described previously.

Please note that in fig. 4.4(d) there is one missing plane that should have been constructed otherwise. But due to noise in the input image it is not always the case, that the detected edges are perfectly aligned to each other on the horizontal or vertical plane, but rather on different heights and depths. The solution here follows a pessimistic approach and discards planes that should have been otherwise reconstructed. It is feasible to discard weak planes in order to solve the false edge/plane



(a) Input image (b) Extracted lines (c) Constructed planes

Figure 4.5: Planes construction using edges as input in a going down perspective. From the input depth image(a) the lines are extracted as in (b) and planes are constructed from these lines (c).

detection problem as presented in section 2.1.

Merge planes Further we would like to combine the newly created planes and the detected planes. We create two histograms, one for horizontal planes that categorizes the planes into bins according to height and a second histogram for vertical planes that categorizes the planes according to depth into bins. Then we merge planes that lie into the same bin together, i.e., planes that are close in height or in depth.

Create steps from planes Next step is to create steps from the planes. Our approach is first to create steps from only the horizontal planes and then try to fit the vertical planes in between or subsequently create vertical planes that would fit between horizontal planes. A step is the building block of the stairs model. First we remove the small planes. We can define a small plane as that with the length,Depth and height less than a given threshold. After that we sort all planes according to depth ascendingly. We further assign each horizontal plane to a step. Adding risers to the model will be discussed while creating the model in the next section 4.4.

4.4 Create initial model

Before creating the initial model we shall define some rules concerning this model:

1. A step consists of $Riser + Tread$, not the opposite.
2. A small gap between the tread and the riser is tolerated (less than 10 cm).

3. Each step has an origin point. This is the center point on the edge where the riser meets the tread.

4. If the depth of a tread exceeds a large threshold (larger than 50 cm), then it is marked as a landing.

5. Putting in mind partially seen planes, a tread must have a minimum depth and a riser must have a minimum height.

After setting some pre requirements we shall now build the stairs model. First we sort all the steps according to depth in an ascending manner. Second, we iterate through each step and compare the euclidean distance between the border of the new step and the last step in the model. We calculate the euclidean distance between the top border of the last step in the model and the bottom border of the new step. Assuming the coordinates (x, y, z) represents depth, width, height respectively as per ROS right hand coordinate system we calculate the euclidean distance only from the x and z coordinates. It suffices us to know that the two steps overlaps on the y-axis. The distance between the two steps has to be below an acceptable threshold. According to the German Institute for standards [20] a riser should be between 14 $cm - 20$ cm while a tread should be between 23 $cm - 37$ cm. That said, the institute further defines an ideal step expression:

$$59 \texttt{ cm} \leq 2 \times \texttt{Riser Height} + \texttt{Tread depth} \leq 65 \texttt{ cm} \tag{4.1}$$

whereas the ideal value for this expression is 63 cm. A step is accepted if it is not smaller or larger than these thresholds by 40% which is a generous threshold taking into account limited camera visibility, camera angle and distance , camera errors as well as plane and edge detection errors. If the new step conforms to these rules and is not far away from the last step in the model then it is added to the model.

We now know that adding a new step is dependant on the existing steps in the model. Now imagine a scenario where for some reasons a faulty step has been added to the model. This could happen if it is detected as the first step in the model or for some other reason it passed the tests for adding it to the model. A problem will then evolve as future steps might be compared against this faulty step and get rejected, although they are correctly detected and constructed steps, since they will not pass the conditions of a faulty step. This will lead to an incomplete faulty stair model. To overcome this problem we create a list of models. When a new step is checked, it is compared against all models and we search for a best match within all models. If such a model is found then the stair is added to that model, else a new model containing this step is created. Finally the model with the largest number of steps is selected as the stair model.

After creating the stair model from individual steps which in turn is created from horizontal planes (Treads), it is time to fit and/or reconstruct vertical planes (Risers). Again euclidean distance is calculated only from the (x, z) coordinates of the

planes, while ensuring that the step and the riser overlaps on the y-axis. This euclidean distance again must be under a given threshold to accept the riser. If the step that represents the best match already contains a riser, then we look for a second best match and so on while the threshold conditions hold. In the going upstairs scenario and due to the camera perspective usually some vertical planes are detected that have no horizontal planes. Steps containing only the risers are created and added to the stair model accordingly in the same manner as shown in the previous paragraph.

After iterating through all risers we iterate through the steps in the model and find the steps that still have no riser to construct it. A riser is constructed from the top edge of the previous tread and the bottom edge of the current tread. In the same fashion a tread is constructed from the top edge of the riser Riser and the bottom edge of the next riser. The result of this process can be seen in figure 4.6.

(a) Going upstairs (b) Going downstairs

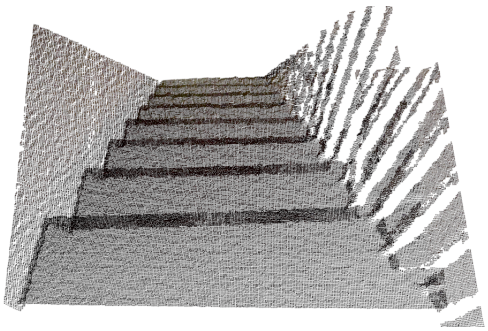
Figure 4.6: The local model constructed from a going upstairs perspective (a). Each consecutive riser and tread are given the same color to indicate, that they belong to the same step in the stairs model. Some steps are missing a tread, this is because due to the nature of this perspective a horizontal plane could not be detected, neither through plane detection nor edge detection. The initial model constructed from a downstairs perspective(b) shows only treads, since vertical planes are occluded.

4.5 Reconstructing Occluded steps

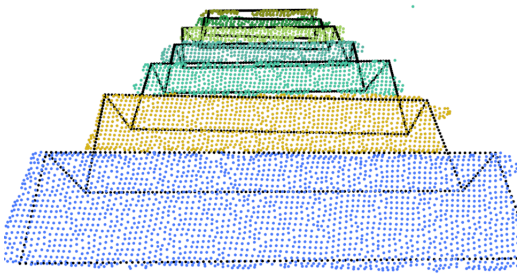
Due to camera perspective usually several planes cannot be seen. As a clear example is while going downstairs, the vertical planes cannot be seen. Also parts of the horizontal planes, that are close to the concave edges, are occluded as can be seen in figure. 3.13. Logically and according to grammar definition a step consists of a riser and a tread and that a small gap in between is allowed or no gap so that there is a corner in between. Using this knowledge we are going to reconstruct the missing

parts of the steps.

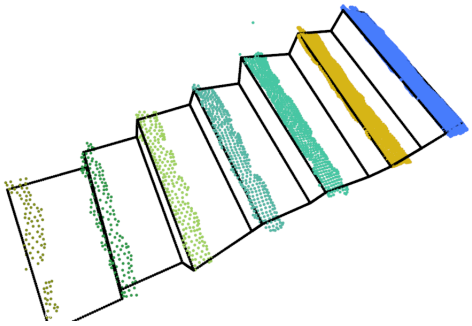
We notice from figure. 4.7(a) that the risers can not be seen in a going downstairs perspective and we also notice that the nearer treads have realistic depth, while the further treads have smaller depth. Since the nearer treads are more reliable than the further treads, we can use this knowledge to reconstruct the missing risers as well as correct the depth of the far treads. For each plane, either tread or riser we define two edges: top edge and bottom edge. For a tread top edge is the far edge in depth while bottom edge is the closer edge. For a riser the top edge is the higher edge of the plane and bottom edge is the lower edge. Now that the initial model is available, we start with the first step (the closest step) and check to see whether the riser is missing. If this is the case then we take the top edge of the current tread and the bottom edge of the next tread. We use these two edges to construct and fit a riser between them. But since the next tread is partially occluded by the previous tread (figure 3.13) we need to move the bottom edge backwards to have the same depth as the top tread. We simply assign the depth values of the top edge to the bottom edge, so that they are parallel and have the same depth. This way we can reconstruct a riser that is really vertical. It remains to correct the depth of the tread. We use the bottom edge of the next tread that we just adjusted and the existing top edge of the next tread to reconstruct the bounding box of the next tread and correct its depth. This procedure is done for each two consecutive steps. In an analogue fashion we can reconstruct missing and occluded planes while going upstairs. Figure 4.7 shows the result of reconstructing occluded and missing planes.



(a) Input image



(b) Planes reconstruction result from frontal perspective



(c) Planes reconstruction result from top perspective

Figure 4.7: Reconstructing the occluded planes in an initial model that is constructed from (a). Frontal and top view perspective for the reconstructed planes in (b) and (c). Each riser and tread in a step are reconstructed using information from the predecessor step.

Chapter 5

Model update

5.1 Chapter Organization

In the previous chapter 4 we have seen how to create a stairs model using a set of planes and a set of edges. The 3D camera produces consecutive point clouds, from each, an initial stairs model could be built. It is the task at hand in this chapter to update the model with each new point cloud input.

When the first point cloud is received and the stairs model is constructed from it, we consider this as the seed to the model and call it initial model as it contains only one point cloud. With each new point cloud the model is incremented with the new data and so, we shall call this stair model the incremental model. On each iteration the incremental model shall be updated by an input initial model. To find the transformation between the incremental model and a newly added initial model, we shall use an iterative closest point approach.

Our approach consists of finding certain features in each model, for instance convex, concave edges, treads or risers etc., and try to find the transformation between the two models using the selected features. Further, a transformation is performed on the initial model to the coordinate system of the incremental model. After that a merge takes place between initial and incremental model. Finally an incremental transformation matrix is maintained and updated on each frame, so that whenever an input initial model is added, it is first transformed to the coordinate system of the incremental model and then an ICP approach is applied on the two models.

5.2 Iterative Closest Point (ICP)

Several initial inputs shall be merged in the model update step. Merging simply mean that the points constituting the models will merged. It is feasible before merging different initial models together is to bring them all into the same coordinate system then merge them, this is called point cloud registration. To do the

registration we shall use an Iterative Closest Point algorithm (*ICP*) to achieve this goal. Point clouds registration using ICP is proposed by Besl and McKay [12].

To register two points clouds, we shall call them model M (model set, $|M| = N_m$) and D (data set $|D| = D_n$), we should find the transformation between the two clouds, i.e., the rotation R and the translation t that minimizes the cost function:

$$E(R,t) = \sum_{i=1}^{N_m} \sum_{j=1}^{N_d} w_{ij} ||m_i - (Rd_j + t)||^2 \tag{5.1}$$

In eq. 5.1 $w_{i,j}$ is assigned to 1 if the i-th point in cloud M matches the j-th point in cloud D. In this case we say that that both the i-th and the j-th describe the same point. Otherwise $w_{i,j}$ is assigned to 0. Next step is to find all the corresponding points and we shall simply call them the correspondences. The ICP algorithm is then able to calculate the correspondences in an iterative manner, where on each iteration closest pair of points are calculated. Based on the correspondences, the transformation (R,t) should be found that minimizes the cost function $E(R,t)$. Other than the original paper the algorithm is well explained in [36, 34, 35]. What interests us in this algorithm is that it is able to find the correspondences as well as the transformation between two point clouds. For this diploma thesis we used the ICP implementation offered by the PCL library. Figure 5.1 shows the result of applying ICP approach to find correspondences between two clouds.

Apply ICP between two clouds To remind the reader of an important term discussed in the previous section: "transformation". When we speak of transformation between initial model and incremental model we mean that we use ICP to find the treansformation between the two models and transform the initial model to the same coordinate system as the incremental model. The transformation is simply a transformation matrix that has translation and rotation components. We apply the transformation matrix to each point and each normal in the initial model to align it to the incremental model. Let us assume we have an "input cloud" that we would like to transform to be aligned to a "target cloud". We start by assigning a large distance threshold between the two clouds, lets say 1 meter. Apply ICP for several iterations and on each iteration, we find the transformation between the two clouds. Further we transform the input cloud using the transformation matrix calculated using ICP. After that We make the distance threshold smaller (circa half the previous threshold) then repeat the whole process and so on until distance threshold is as small as 1 *cm* or no more correspondence points could be found.

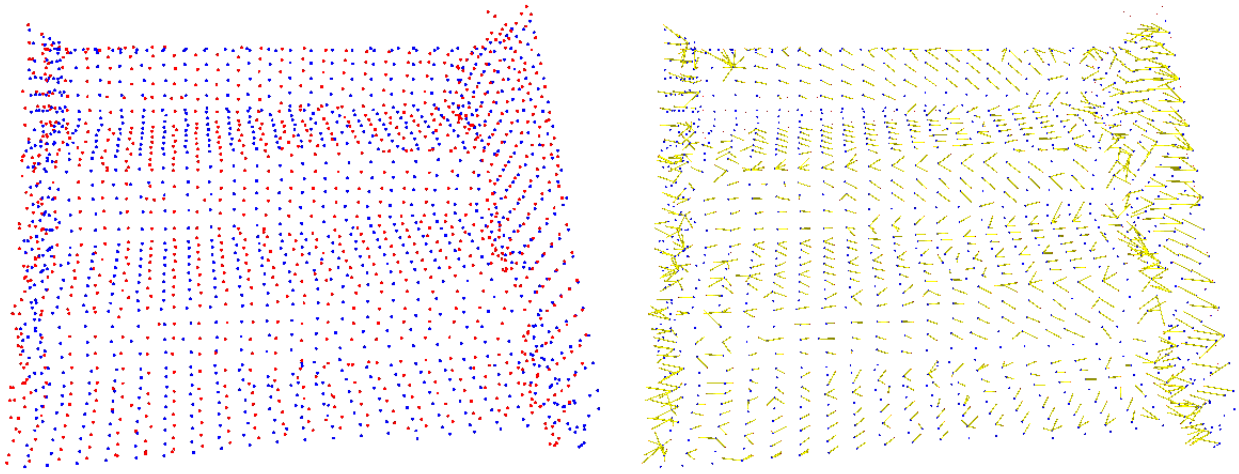


Figure 5.1: Showing on the left hand side two point clouds from two consecutive 3D images in red and blue points. On the right hand side we show the result of registration where the yellow lines binds each two corresponding points.

5.3 Find features

We should take advantage of the stairs model structure we have at hand and try to find suitable features within this structure to feed to the ICP. The idea here is to find different features in two input models, then apply ICP to these features separately. After that we obtain the correspondence points of all point features and apply ICP one last time to the correspondence points to find the transformation. We tried ICP on three different kinds of features:

1. Edges features (six features) which include:
   a) Convex edges.
   b) Concave edges.
   c) Left,right tread edges.
   d) Left, right riser edges.
2. convex hull features of treads and risers (two features).
3. Points of treads and risers (two features).

In the actual implementation of ICP on convex hull it is usual that the number of points of a convex hull of a plane is too small. For this reason we sampled between each two points of the convex hull a line segment that consists of points separated by one 1 *cm* each. Otherwise there are simply too few points for the ICP to function correctly. For this reason choosing to apply ICP on treads and risers points as our features yields better results than Applying ICP to edges features or convex hulls features. The number of points in the previous two cases seems to be too few. An error emerges after adding several point clouds where points are not added correctly. This is an incremental error that takes place when the transformation matrix between two models is not calculated precisely on each frame and so the incremental matrix errors keeps growing. An example that shows the result of both using convex hulls and planes points as an input to find the transformation can be found in figure 5.2. For this reason we shall use ICP with treads and risers points features as an input to calculate the transformation.

The highlights of this approach is presented in algorithm 9.

5.4 Merge models

After the initial model has been received, it will be transformed using the incremental transformation matrix. Furthermore ICP is then applied to find the local transformation between the initial and the incremental models. After that, the initial model is transformed again using the local transformation calculated in this step.

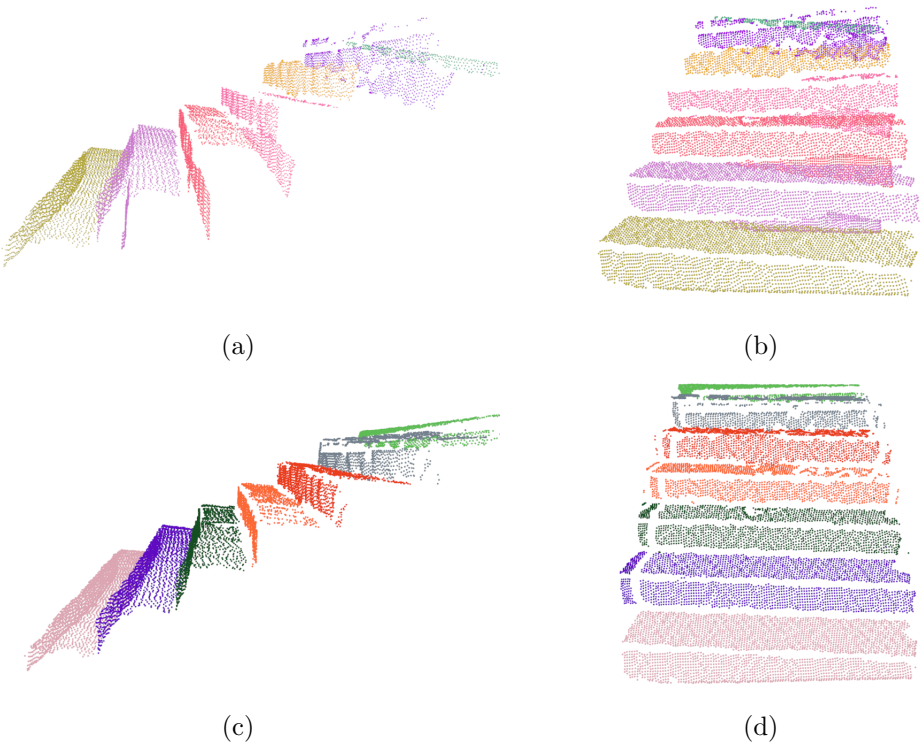


Figure 5.2: A faulty model update shown in that uses convex hull points to find the transformation between models in a side view(a) and a front view (b). It can be clearly seen from the side view that the points of risers are not correctly aligned to one another. While (c) and (d) show a correctly constructed incremental model that uses treads and risers points as features to find transformation.

Algorithm 9 Align initial model to incremental model

```
transform the initial model using the incremental transformation matrix
model_1 = initialModel, model_2 = incrementalModel
Find the features in model_1 and model_2
for all features do
    correspondances = icp_get_correspondances(feature_in_model1,feature_in_model2)
    allCorrespondances.add(correspondances)
end for
transformation = icp(allCorrespondances)
transform model_1 to model_2
update incremental transformation matrix with the last transformation
```

Now that the initial model is aligned to the incremental model, it is time to merge them. For each step in the initial model we try to find a matching step in the incremental model, using the following set of rules:

- If the two steps have treads, they should have similar heights.
- If the two steps have risers, they should have similar depth.
- If one step has a riser and the other has a tread, then by comparing the bounding boxes of the two planes, they should be aligned to each other or have only a small gap in between.

If a step in the initial model has no match in the global model, then a new step must be added to the incremental model. Next we present the rules for adding a new step:

- If the incremental model is empty, then add the initial model step as the first step.
- If the incremental model is not empty, then we find whether the incremental model of stairs is of a going upstairs or going downstairs perspective, then:
  - If we are going upstairs, we search for the nearest step in the incremental model that matches the step from the initial model. Then we check whether the new step should be added before or after the incremental model step based on the depth and height values. Also we must make sure that the position for adding the new step is not occupied already. Furthermore the two steps must overlap along the horizontal axis.
  - If we are going downstairs, an analogue check is done to see where to place the new step.

Figure 5.3 shows two models where the initial model has been transformed to the coordinate system of the incremental model before the two models are being merged. Each model is assigned a different colour.

Finding the camera's perspective related to stairs We must also explain how we find out whether the incremental model is from going up or down perspective. First the steps of the model are sorted ascendingly according to the depth. A model is taken from a "going upstairs perspective" if each step is lower in height and depth value than the following step and is from a going upstairs perspective if each step is higher than the following step but closer to the camera. If non of these rules apply the model is assumed to be from a "going upstairs perspective" for safety reasons. For example when there is only one step to be seen. The direction of travelling (upwards/downwards) has a weight, i.e., if the last 'n' initial models were of type "going upstairs perspective" while the last initial model is of type "going downstairs perspective", then the model remains of type "going upstairs" with a

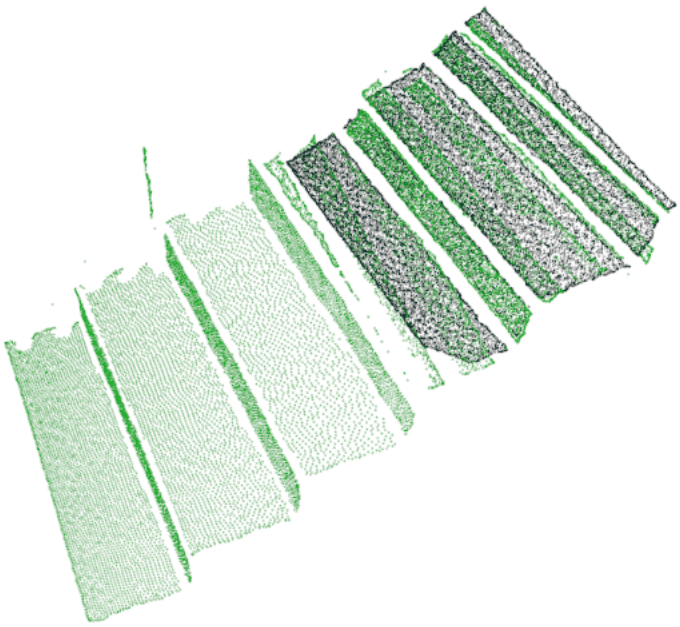


Figure 5.3: An image showing merging initial model to incremental model. The two models are assigned different colours. The initial model has been transformed to the coordinate system of the incremental model. The incremental model is the one with the larger number of steps.

little less weight. Figures 5.4 and 5.5 show how the stairs model is incremented through several consecutive inputs. As can be seen with each iteration, new points are added to the incremental model.

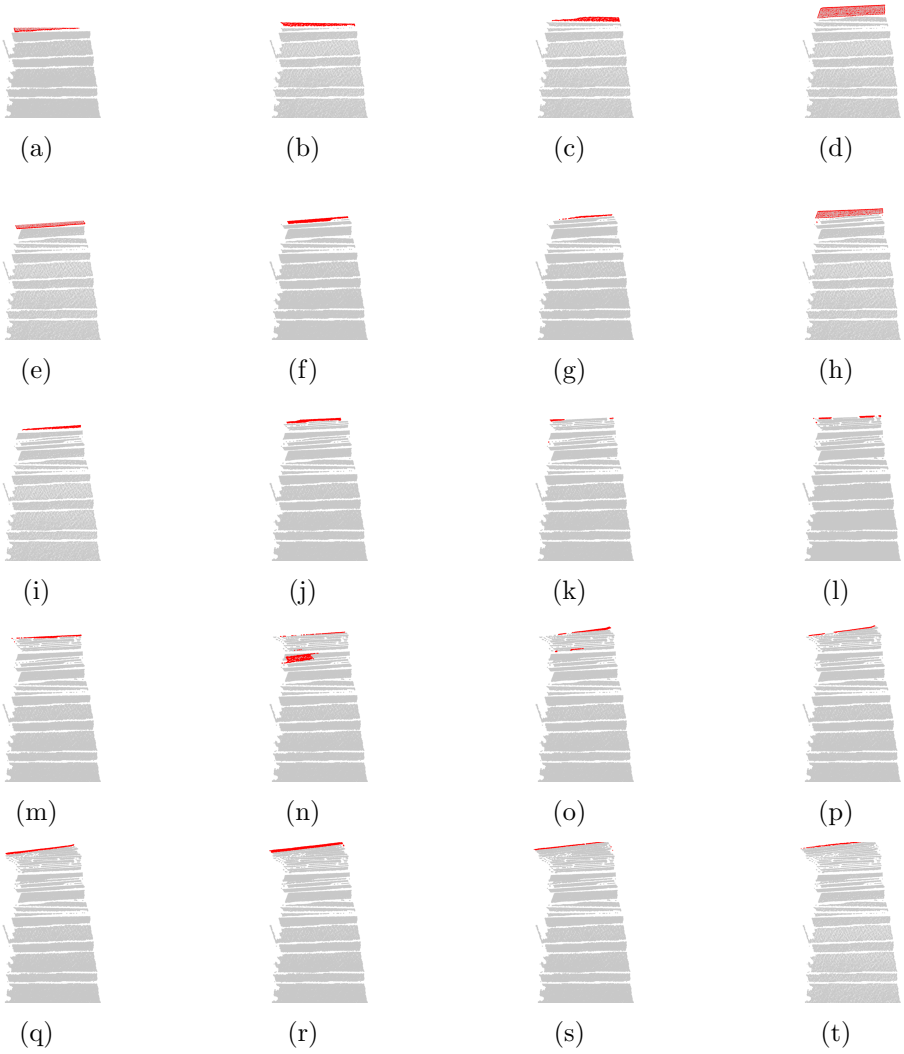


Figure 5.4: Updating the incremental model in a going upstairs perspective. The red points represent the newly added points. In reality building each of these models requires 100-120 input clouds but for the sake of visualization only images with the highest number of added points are shown.

Merging two steps Next step is to explain how actually two steps from two models are merged together. As we have seen in the previous steps in this chapter, consecutive initial models are added to the incremental model. So upon adding each new initial model, it is first registered and aligned against the incremental model to come

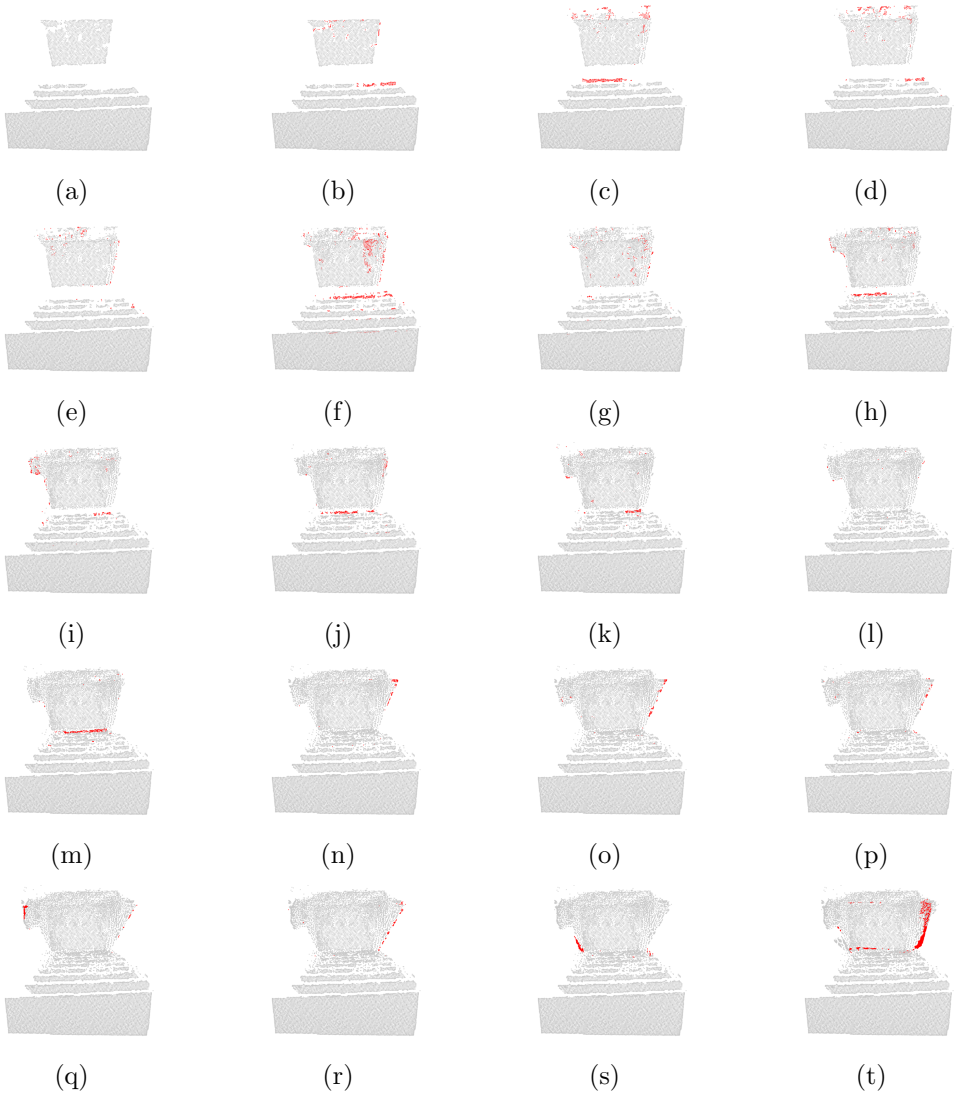


Figure 5.5: Updating the incremental model in a going downstairs perspective. The red points represent the newly added points.

as close as possible in terms of distance and rotation. After that we try to find for each step in the initial model a matching step in the incremental model. If a match is found, then we merge the two steps together, else a new step is added to the model. Now we discuss how two steps are merged together. We name the entities from the initial model as the input step/plane etc., and those of the incremental model as the target step/plane etc.

When merging two steps We should differentiate between four different cases:

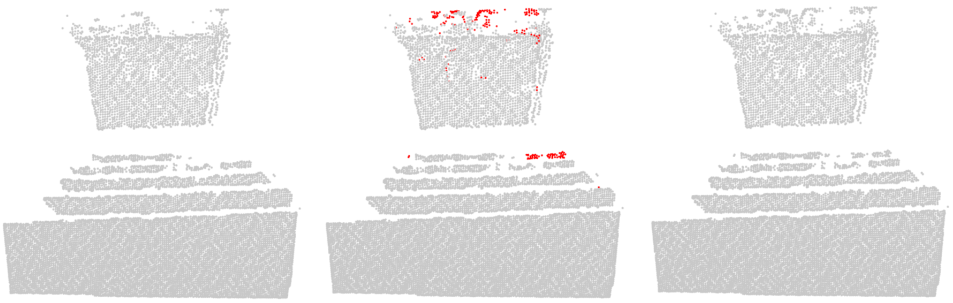
1. The input step has a tread and the target step has a tread → Merge treads.
2. The input step has a riser and the target step has a riser → Merge risers.
3. The input step has a tread and the target step has no tread → Add new tread.
4. The input step has a riser and the target step has no riser → Add new riser.

It remains to explain how two planes are merged together. When merging two treads or two risers a differentiation between three scenarios is made:

1. The two planes are of type "plane model", i.e., they have no points, only bounding boxes. In this case the average of each of the four points is calculated, i.e., average point of top left corner of the two planes is calculated and the same for the other corners. The reason that we calculate the average point and not the maxima point is to avoid the case where a large faulty plane with wrong dimensions is added and leads to wrong overall dimension of the plane.
2. One plane is of type "plane" with points and the other is of type "plane model" with only a bounding box. An analogous approach is followed as in the previous case.
3. The two planes are of type "plane" with points inside. First we update the bounding box as explained in the first case, then we merge the containing points. To do so first the input plane's points are registered and aligned to the target's plane's points using ICP. Further we search for correspondences between the two planes. Only those points in the input plane that have no correspondence in the target cloud are added. This can be seen in figure 5.6 where only the points with no correspondences, we call them rejections are added.

5.5 Summary

We have seen in this chapter a pipeline of operations performed to update the incremental model on each iteration using an input initial model. The pipeline consists of the following operations whenever an initial model is added:



(a) Initial model (b) Adding rejected points (c) Model after update

Figure 5.6: The initial model as seen in (a) is updated using the points marked with red in (b) and the updated model is shown in (c).

1. Transform the initial model to the coordinate system of the incremental model using an incremental transformation matrix.
2. Choose features in the two models that will serve as inputs to ICP.
3. Apply ICP on the features set to find the set of correspondence points.
4. Apply ICP in a second run only on the correspondence points to find the local transformation between the two models.
5. Transform the initial model using the local transformation calculated in the previous step.
6. Update the incremental transformation matrix using the local transformation.
7. Merge initial to incremental model.

Chapter 6

Results

The result of this thesis shall be evaluated from two perspectives. The quality of the initial models and the quality of the incremental model. To test our approach several recordings have been made using a Kinect camera to obtain streams of 3D images while going up and down the stairs. The camera has been hand operated to make the recordings. For some recordings the movement with the camera were relatively slow, i.e., around 10-15 seconds to traverse the staircase and for other recordings the movement has been extremely slow (circa 50 seconds). Other than operating the camera by hand to record an input for our algorithm, a gazebo simulator has been used to make recordings of a robot with a 3D camera traversing up and down the stairs. For more information about gazebo please consult section 8.1. For each initial model as well as the final incremental models we have to evaluate the results of the following parameters:

1. Number of steps detected.
2. Dimensions of each step.

We shall start with the initial models. Each input has the following format "place-name direction-of-travel sample-number", where place name is where the sample has been recorded, direction of travel is either going upstairs or down stairs and the sample number is simply a counter. An input consists of a stream of 3D images, usually between 60-120 images, which is the usual number of images taken while traversing a staircase.

We first show the result for the constructed initial model in comparison to the real stairs in figure 6.1 where the left column shows several input images from up and down stairs perspective where the left column shows the result of building an initial model from the corresponding input. Notice that in going upstairs perspective treads and risers could be detected while in going downstairs perspective, only treads could be detected.

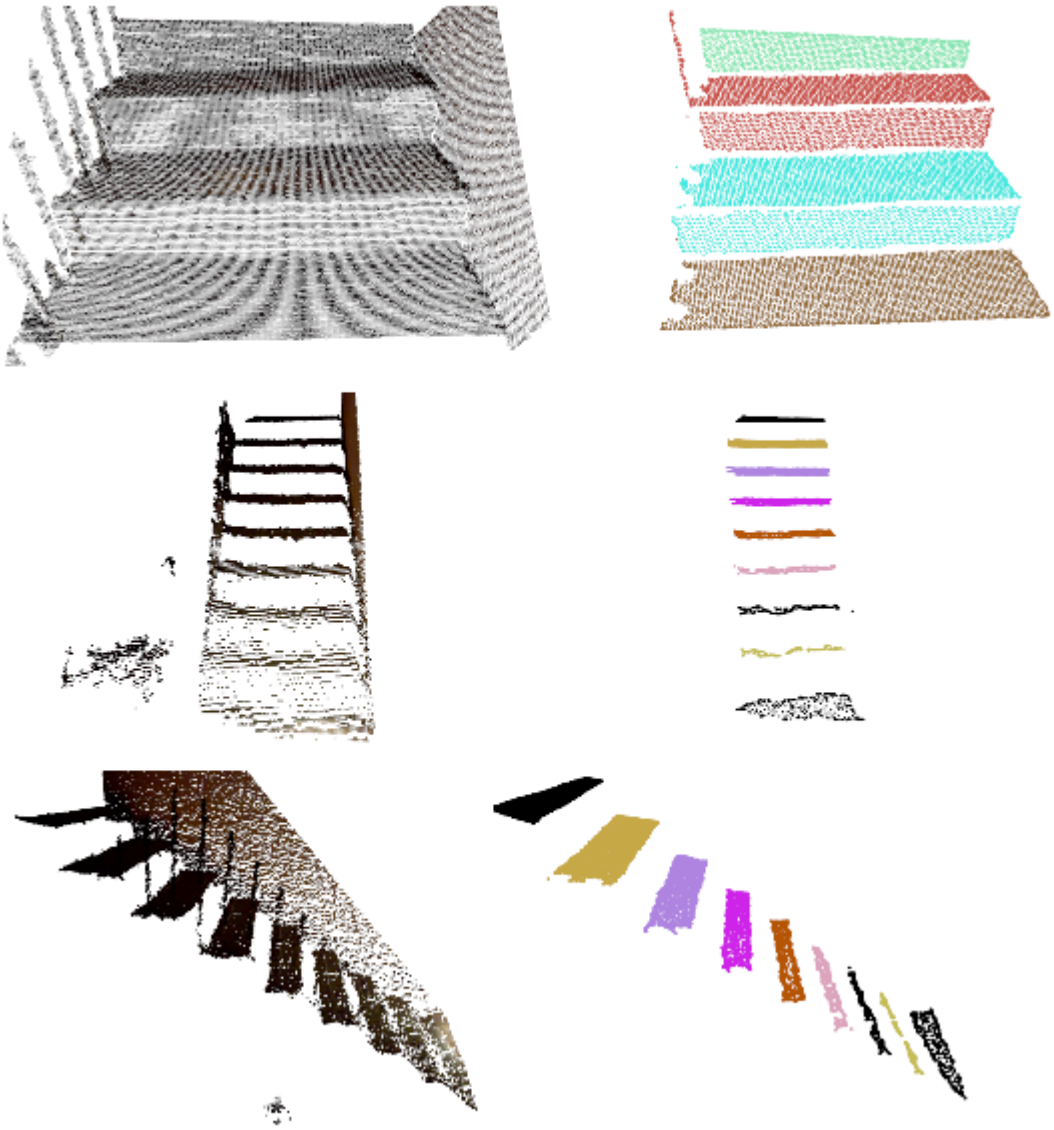


Figure 6.1: The input images of stairs are present on the left column with the initial models built from these inputs in the right column. The first row is for a set of stairs from a going upstairs perspective. The second and third rows shows stairs from a going downstairs perspective rotated in different views.

Further using several initial models as input an incremental model has been built, where new models have been registered and aligned to the existing incremental model so that only new, non matched points are added. The result of this process is shown in figure 6.2 for stairs models in going upstairs and downstairs perspective. Also notice that in the incremental model in figure 6.2(a) and (b) that the last step's tread and riser are marked with different colours. This is to indicate that the last tread is a landing because its depth is large enough to be considered a landing in the stairs model.

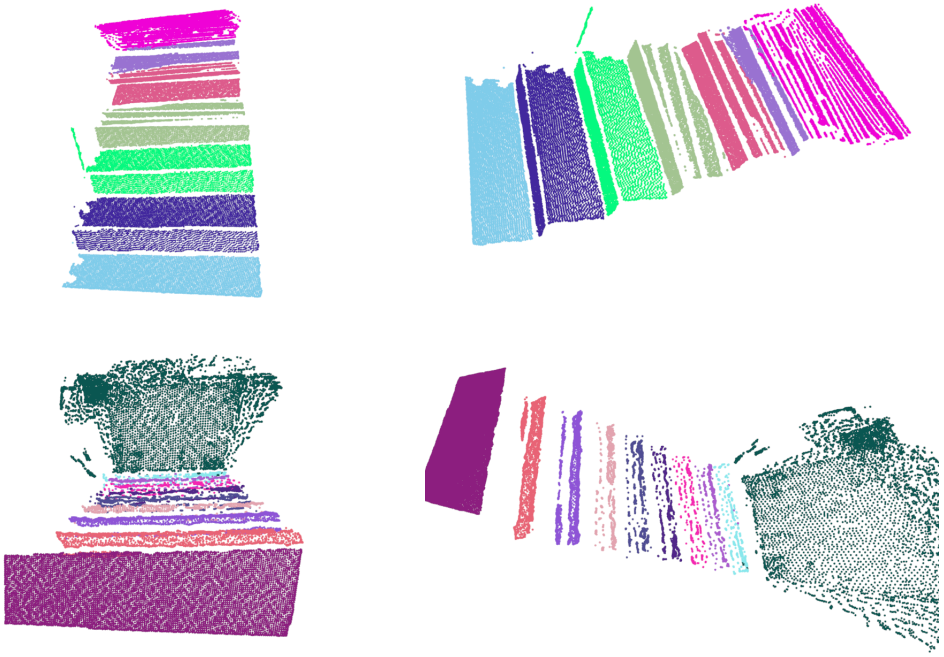


Figure 6.2: Incremental model result for a staircase in a going upstairs perspective in (a) and (b), and in a going downstairs perspective in (c) and (d). The highest step in (a) and (b) is marked as a landing, as well as the lowest step in (c) and (d).

After we tried to visualize the detected stairs for the reader, we shall introduce our findings in detecting stairs, specifically the quality of our detection and modelling approach compared to the reality. We start by the results of the initial models as seen in fig. 6.3. For the case of going downstairs only treads are detected, and hence only tread results are shown. The tests have been conducted so that for each point cloud in each stream of depth images, the number of steps, treads and risers have been counted, also the depth and height of the steps were measured in reality. Step's Length has been ignored since a step is usually much wider than the camera's focal width, so that the whole width cannot be captured by one image from such a short distance. The camera has been operated manually while making the recordings and

the author tried to maintain a camera's height of 1 meter.

Further the number of detected steps and their corresponding dimensions in cm have been calculated and the deviation from the ground truth is shown in histograms in figures: 6.3, 6.4 and 6.5. When treads for instance lie in the column with a deviation of -1 cm this means they have a deviation between $[-2, -1[$ cm from ground truth. On the other hand the last column for example with < -5 cm indicates all planes with deviation of less than 5 cm. The deviation means the difference of depths of treads and of heights of risers from the ground truth data. The percentage on the vertical axis indicates the percentage of planes that lie in a given bin compared to the total number number of planes (risers or treads).

We can see in the samples in fig. 6.3 that several planes have somewhat significant deviation from the ground truth of several centimetres. We should notice that the initial detection is done on single images and so in several of these input images the stairs are only partially seen, the images are distorted etc. Regarding these limitations, it is tolerated that single image might produce significant errors mainly because we are interested in the final incremental models. Also notice that in the sample "FHG going down 1 initial result" the percentage of treads with an error higher than -8 cm is significantly higher. This is because the camera was somehow less rotated while recording the images than in the other samples. So the camera's pose greatly affects the results. A tilting angle downwards between $30\,^\circ$ and $40\,^\circ$ usually generates acceptable results.

Next we show in table 6.1 the percentage of the total steps detected versus the number of steps in reality in initial model. Usually the result is above 90% and it is slightly higher in the case of going downstairs where only treads are detected.

Number of steps Further we inspect the result of the incremental model that has been built from several initial models. The number of steps in each incremental model is 6-10 steps. First we show the result without reconstructing the occluded parts of the stairs as explained in section 4.5. Then we show the result of incremental model with the reconstruction step and we examine the result of the two approaches.

Fig. 6.4 shows the results of the incremental models after adding several initial models. The number of inputs of each stream, from which an incremental model is constructed is on average 80-90 images. We can see that the result in most cases is on average around -3 cm where the worst average is -4.4 cm. The depth of the landing has been ignored from this result as there is no mean to compare how much of the landing is actually seen on the camera to how much has been detected. Also in some of these streams of images a step or more is only partially seen in reality and hence cannot be detected correctly.

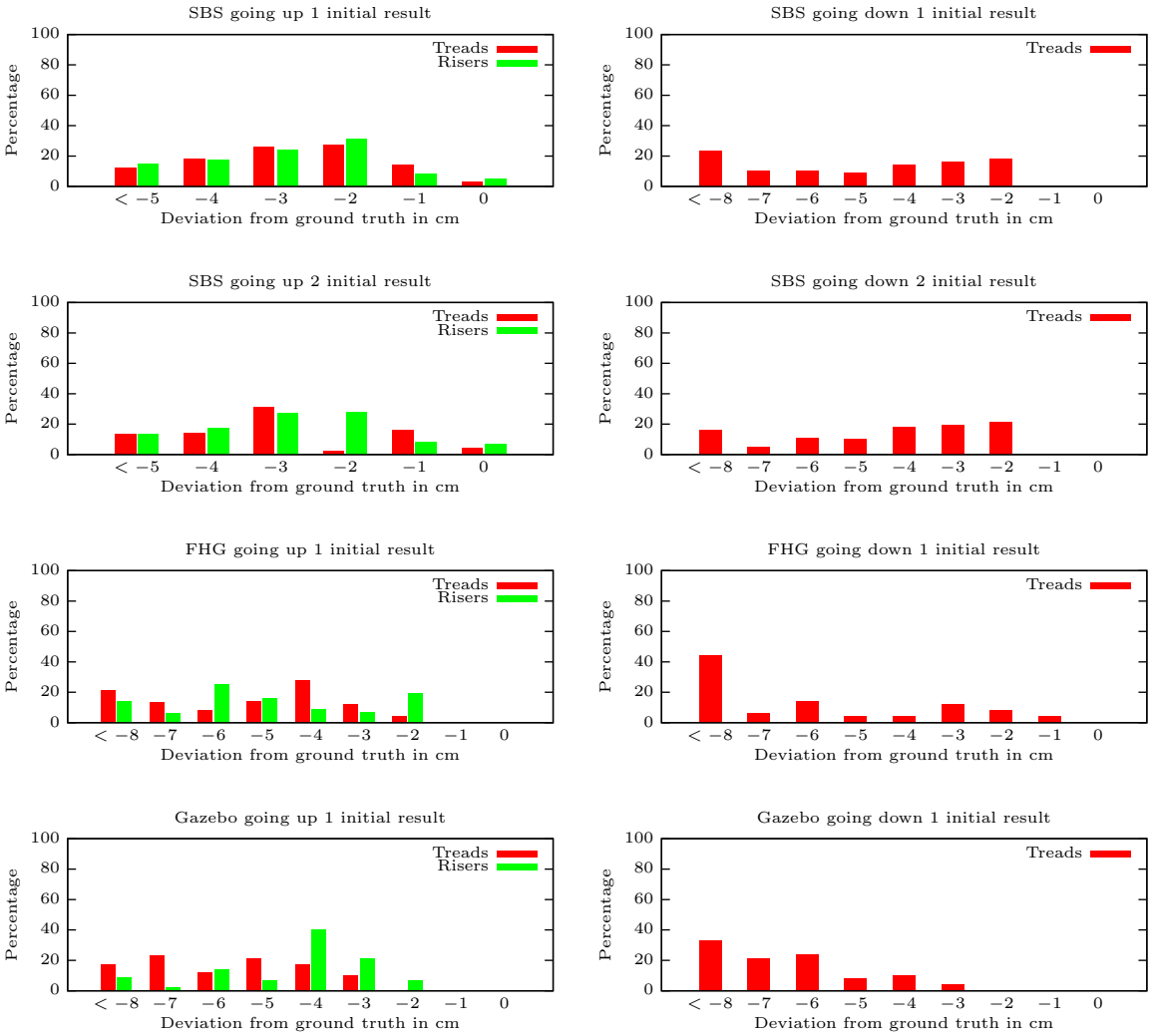


Figure 6.3: Initial model results of several samples from a going upstairs and downstairs perspective.

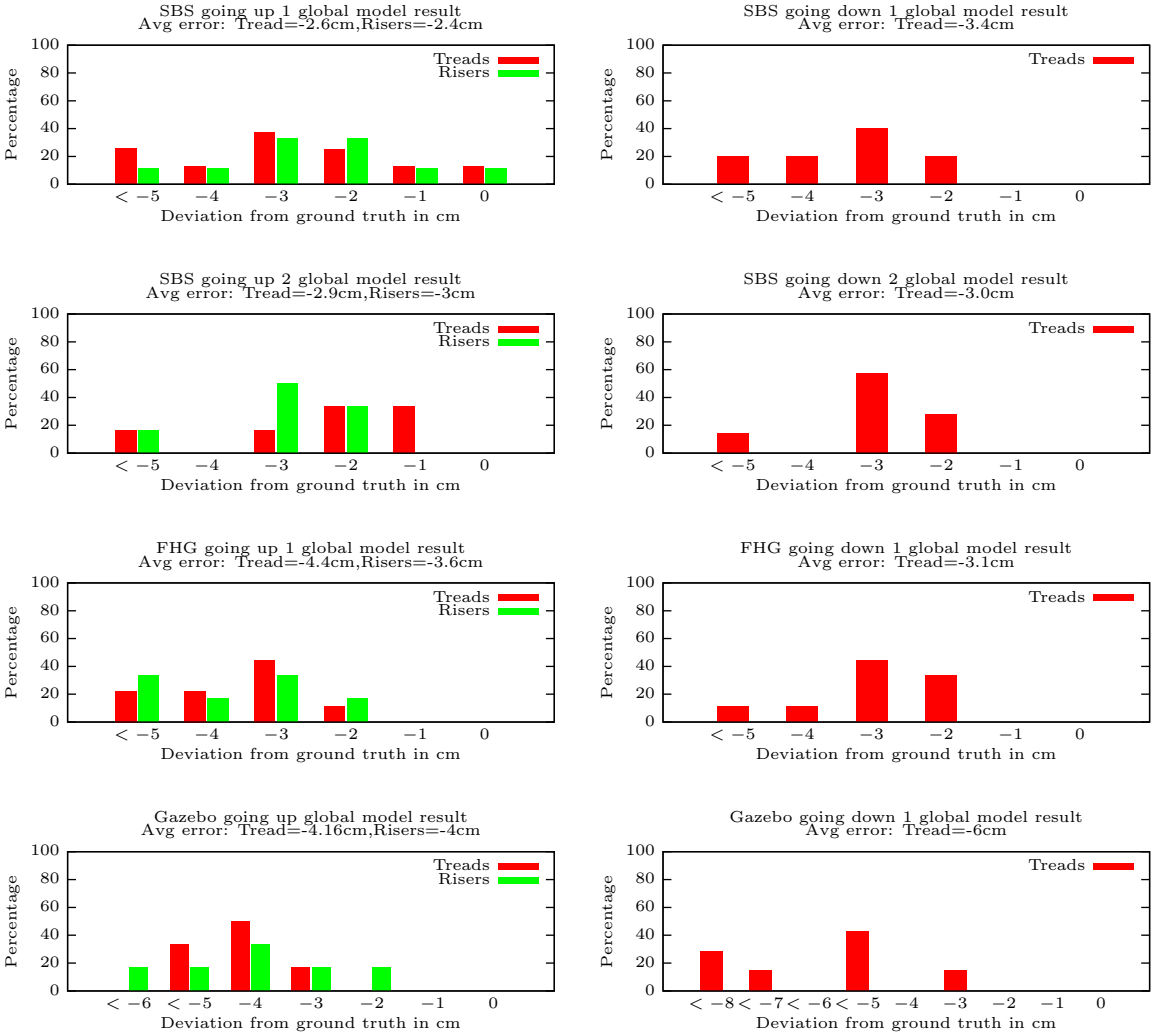


Figure 6.4: Incremental models results without reconstructing the occluded areas.

Sample name	Number of steps detected/reality
SBS going up 1	91%
SBS going up 2	92%
FHG going up 1	94%
Gazebo going up 1	94%
SBS going down 1	98%
SBS going down 2	98%
FHG going down 1	97%
Gazebo going down 1	91%

Table 6.1: The table shows the percent of detected steps compared to steps in reality. Abbreviations like SBS and FHG indicate simply the sample name while going down and going down indicates the direction of traversing stairs. Gazebo samples are the results of running our stair detection algorithm using a robot in the gazebo simulator.

Finally we show the result of the incremental model as explained in fig. 4.5 where the occluded parts reconstruction mechanism is used to reconstruct unseen and partially seen steps using their neighbours. Our goal using this approach is to enhance the detection and modelling result using existing knowledge about the stairs models. The result is shown in fig. 6.5. On one hand the result has been enhanced slightly, where the average of errors is around -2 *cm*, but on the other hand some treads are detected larger than they are in reality by up to 3 *cm*.

Tables 6.2 and 6.3 shows the results of the incremental global models with and without the reconstruction step. We can see that using the reconstruction, the result has been slightly enhanced. From this result we can say that reconstructing occluded parts of the stairs is a reasonable approach that enhances the result slightly but as mentioned earlier it causes some steps to be modelled larger than what they are in reality and hence using this approach has its own advantage and disadvantage.

Further we show in table 6.4 the number of steps detected compared to the number of steps in reality. It is seen that in some samples a step could not be detected, usually this is because only a very small portion of this step can be seen in the input image and we prefer to detect steps where enough parts of them can be seen. Hence the missing step has been discarded on purpose. This is a safer approach than detecting small chunks and consider them step, while in reality they are not.

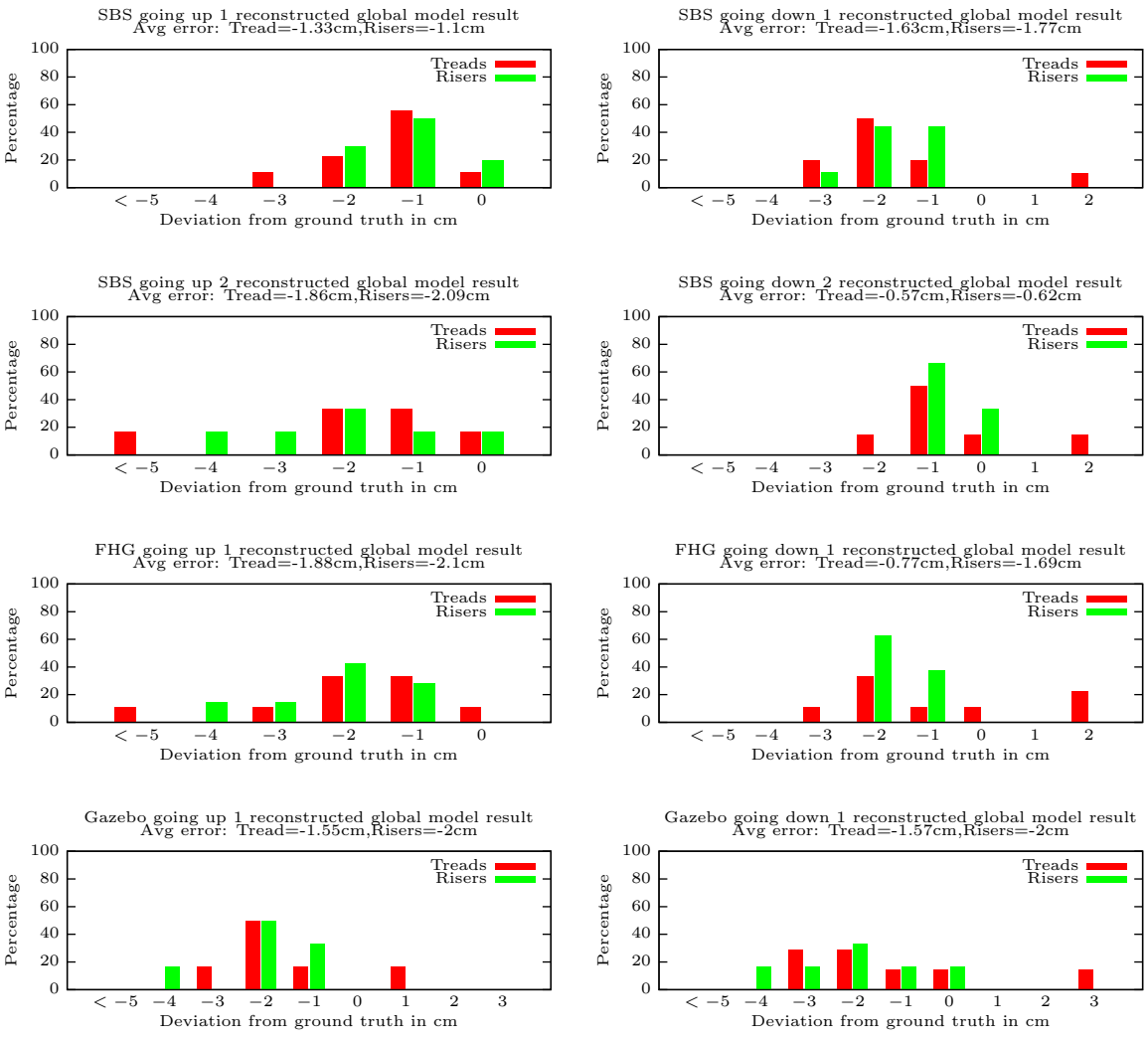


Figure 6.5: Incremental model with model reconstruction. We aim with this step through regenerating/reconstructing the unseen and partially seen steps to enhance the final result of the incremental model.

Sample name	Treads	Risers
SBS going up 1	$-2.6\ cm$	$-2.4\ cm$
SBS going up 2	$-2.9\ cm$	$-3\ cm$
FHG going up 1	$-4.4\ cm$	$-3.6\ cm$
Gazebo going up 1	$-4.16\ cm$	$-4\ cm$
SBS going down 1	$-3.4\ cm$	
SBS going down 2	$-3\ cm$	
FHG going down 1	$-3\ cm$	
Gazebo going down 1	$-6\ cm$	
Average	$-3.68cm$	$-3.25\ cm$
Total average	$-3.46\ cm$	

Table 6.2: The error average of incremental models without reconstructing occluded stairs parts.

Sample name	Treads	Risers
SBS going up 1	$-1.3\ cm$	$-1.1\ cm$
SBS going up 2	$-1.86\ cm$	$-2.09\ cm$
FHG going up 1	$-1.88\ cm$	$-2.1\ cm$
Gazebo going up 1	$-1.5\ cm$	$-2\ cm$
SBS going down 1	$-1.6\ cm$	$-1.77\ cm$
SBS going down 2	$-0.57\ cm$	$-0.66\ cm$
FHG going down 1	$-0.7\ cm$	$-1.62cm$
Gazebo going down 1	$-1.57\ cm$	$-2\ cm$
Average	$-1.44cm$	$-1.66\ cm$
Total average	$-1.55\ cm$	

Table 6.3: The error average of incremental models with reconstructing occluded stairs parts.

Sample name	Number of steps detected/reality
SBS going up 1	10/10
SBS going up 2	6/7
FHG going up 1	9/9
Gazebo going up 1	6/7
SBS going down 1	10/10
SBS going down 2	6/7
FHG going down 1	9/9
Gazebo going down 1	7/7
Steps detection Average	95.45%

Table 6.4: The number of detected and modelled steps compared to the number of steps in reality.

Chapter 7

Diploma Thesis Summary

7.1 Summary

In this thesis we presented a method to detect and model stairs. In the stair detection chapter 3 we first used a plane detection algorithm that is based on approximate local surface normal computation and region growing to detect planes. Then we estimated the camera orientation in terms of pose and height and rotate all planes, so that horizontal planes coincides with the gravity vector.

After that the neighbouring planes have been merged and small/irrelevant planes have been removed. In the next part of the chapter we showed two different edge detection methods that work in going upstairs and going downstairs perspective. We started by detecting edges from the normal image. The normal image has been calculated by finding points' normals in the point cloud and convert each normal to a color based on the normal's value. Then we detected edges using a pipeline of algorithms, canny edge detection, then contour finding then hough transform on contours. After that we applied a filter to remove vertical and small lines. Further using a histogram we merged near lines with similar orientations. In the second edge detection approach we used what we call depth jumps where neighbouring points in an organized point cloud that are separated by large depths are marked as depth jumps. These points are then used in a RANSAC line fitting algorithm to find the lines among them, and as with the first edge detection approach the lines are filtered and merged together.

In the modelling chapter. 4 we introduced what is called an initial model, that is built from the planes and edges detected in the previous chapter. First using the set of detected edges we consider each two lines that lie on the same horizontal or vertical plane and use them to construct a plane. The edges are also marked as either convex or concave edges accordingly. Then we merge the planes that we constructed with the planes detected from the previous chapter. Each plane holds important attributes like the bounding box, whether its edges are concave or convex,

L x D or L x H, etc., Further using the set of planes we try to find candidates for steps. If a riser and a tread, i.e., a vertical and horizontal plane are close enough to each other and they preserve the grammar defined in the mentioned chapter then we construct a step from them. After the set of steps have been constructed we use these steps to construct our stair models. To do that we "fit" the steps into a stair model, where neighbouring steps, that according to our grammar rules are a match are added to the stairs model. This model we call the initial model.

We also presented in this chapter the concept of model reconstruction. Using existing planes we try to reconstruct the occluded parts of the stairs, i.e., unseen or partially seen treads and risers for example by fitting a riser between each two treads and fitting a tread between each two risers.

The next chapter 5 shows how to build what's called an incremental model that is constructed from several inputs (several initial models). The first initial model is assigned as the incremental model. For each consecutive model we register the two models using a point registration technique for example Iterative Closest Point (*ICP*). This algorithms is given as an input the points of the treads and risers of the initial and incremental model. The algorithm tries then to find for each point in the initial model a matching point in the incremental model. We then calculate the transformation between the two models and the initial model is transformed and aligned to the incremental model. Further we try to merge the two models by finding for each step in the initial model a matching step in the incremental model or create a new step if non exists. We also detected the landing as a separate step when its depth exceeds a given threshold.

In the results (chapter 6) the result of this research has been presented. We saw that in the result of the initial model it is common, that the dimensions of the treads and risers varies significantly from the ground truth because in single frames it is usual that a tread or a riser is only partially seen. On the other hand we have seen that the incremental model has been well constructed and the majority of the steps have been detected and modelled. The results of the incremental model shows that its stairs varies on average $-3\ cm$ from the ground truth for both risers and treads. They also show that the majority of the steps have been detected.

7.2 Contributions of the thesis

We have seen in the state of the art that stair detection is done based on single features, for example on edge detection from RGB images like in [28] or on plane detection from point clouds using scan line grouping technique in [38] and RANSAC plane fitting in [29]. We tried on the other hand to consider various features and combine their results, for example plane detection from depth image; edge detection from normal image and from the depth jumps in the organized point cloud structure.

We further defined our grammar rules for the staircase modelling based on the grammar of [39]. We used the natural stair characteristics of stairs to build the initial model and showed how model attributes can be calculated. Further we showed how a consistent incremental stair model can be built from several initial model inputs, through the stairs natural characteristics like the treads, risers and the stair edges.

We handled stair detection and modelling in case of going downstairs, which to the best knowledge of the author is an unhandled case by the state of art. Further our approach is able to reconstruct the occluded parts of the steps.

Our approach also overcomes detecting false steps problem as mentioned by [28] and [38] so that our edge detection algorithm does not produce false edges and hence no false steps. Furthermore we were able to estimate the camera's orientation and distance related the the ground (camera's gravity) based on the detected planes.

7.3 Future work

There are several types of stairs in the world. We only handled in this thesis straight undamaged house stairs. There are for example damaged house staircases to handle, for example in an urban search and rescue scenario (*USAR*). The case of detecting spiral stairs has not been handled. Further there are other types like tread only stairs, metal net stairs with holes, maybe even ladders.

To guarantee successful registration of several stairs models we had to make our recordings while moving very slow so that the transformation between two frames is as small as possible. This is in no way tolerated in a real time application. For this reason, work could be done to enhance the registration algorithm to be able to find transformation between two 3D frames that have minimal overlap between them.

Furthermore the defined grammar of stairs starts and ends with a landing, and so it does not consider the case of multiple staircases separated by landings. Maybe the approach and the grammar could be altered to allow multiple staircases in one model.

Chapter 8

Appendix

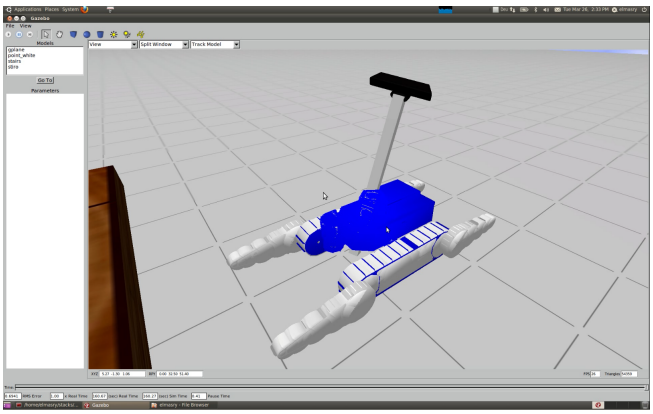
8.1 Gazebo simulator

Gazebo simulator[8] is an open source simulator written in C++ and based on the 3D graphics engine OGRE. Using Gazebo we wanted to model a robot, use that robot in climbing stairs and conduct our experiments using this simulator. Gazebo simulator is integrated in ROS. We decided to model the Bluebotics [2] robot that is used as part of the Nifti project [4]. As can be seein in fig 1.1 this is a tracked robot with two front and two back flippers. Also it contains a SICK laser scanner and a ladybug camera. We used a markup language to model our robot called URDF. This language define primitive 3D shapes like Cylinders, Rectangles, etc. It also defines bounding boxes for these shapes. Further we defined the joints between the shapes as well as the physics of robot and the simulated world. Since URDF allows the definition of only primitive shapes, we used several wheels to simulate the functionality of the robot's tracks. So each track and each flipper is simulated by four wheels. We also covered the wheels with 3D models of tracks and flippers to look realistic. For each side of the tracks five (5) wheels have been used and for each flipper four (4) wheels have been used. Furthermore we wrote staircase model to be used by our robot, with the dimensions $LengthxDepthxHeight = 130x25x18cm$.

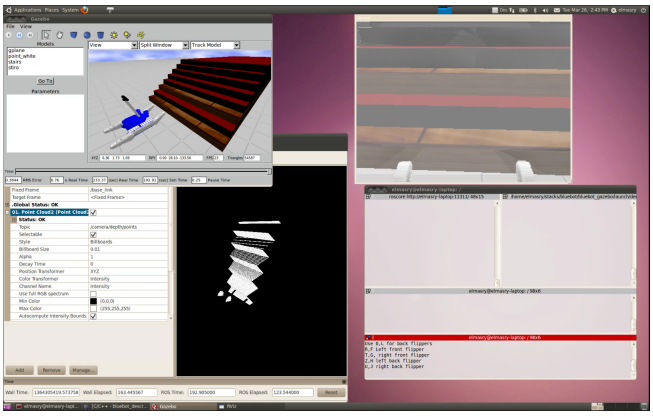
Gazebo enables us to define simulated motors which are installed on the wheel joints. Through these motors we were able to operate the robot and flippers. A C++ plugin has been written to serve as the backend for the motor controllers. We wrote also a keyboard controller that communicates with the motor controller plugin to be able to operate the robot. Further we used two plugins to simulate a Kinect and an RGB camera. These two plugins are available to use in Gazebo. Figure 8.1 shows the gazebo environment while using our simulated robot.

Using this environment we were able to use the robot to climb stairs in the simulator. While doing so we made our recordings, a stream of 3D images while going up and down the stairs. These streams of images were then used by our algorithm

to try to detect stairs. The advantage of that approach is that we have the ground truth data of the environment, i.e., number of stairs, LengthxDepthxHeight, height and tilting angle of the Kinect camera.



(a) Simulated robot



(b) Robot with kinect and rgb camera plugins

Figure 8.1: The Simulated Bluebot robot in Gazebo in (a) and we can see an arm in the middle of the robot and on its end a Kinect camera is mounted. In (b) we show how the 3D images were obtained. The Kinect simulated camera is able to produce a stream of 3D images while climbing the stairs as seen in the bottom left corner of the image and RGB images as in the bottom right corner. These images are then later used by our algorithm to detect and model stairs.

8.2 Microsoft Kinect as an RGB-D Camera

Microsoft Kinect is a game controller introduced for the game console Microsoft XBox360 [3]. The device has been since its release used by the robotics community

as a cheap and fast RGB-D Camera. RGB-D stands for a camera that is both a 2D RGB camera and a 3D camera that is able to capture depth information. Several drivers have been written for Kinect since then. One of the most famous that is also used in this research is OpenNi [5]. The 2D camera is a standard RGB camera while the 3D camera consists of two lenses, one lens is an infra-red rays emitter and the other is an infra-red camera to capture the falling points of these rays. Kinect is able to provide an organized point cloud, for example in a 3D image with 320 x 240 resolution there are 320 lines in the image, and each line contains 240 points, i.e., each point is assigned a row and column index. Knowing the neighbourhood of each point will prove very useful later in some of the algorithms used here, since the calculation of the neighbourhood if not provided is usually expensive. Next to that the Kinect is provided by a motor at its base that enables it 1-degree of freedom in a tilting up and down manner. Kinect is also provided by 4 microphones to capture sound, but no driver is provided by OpenNi for this feature and also it is not needed.

For this research the following resolutions have been used: 640 x 480, 320 x 240 and 160 x 120 with the last resolution providing the best performance and the lowest precision and vice versa. The reasons for choosing Kinect as our mean of obtaining images and conducting experiments is that Kinect can provide 2D/3D images at very high frequency (320 x 240 at 30Hz and 640 x 480 at 25Hz) without causing a 100% CPU usage. The second reason is that Kinect is very light and though it is easier to move it around and obtain images (in contrast to other heavier 3D sensors, like SICK laser scanner).

8.3 Point Cloud Library (PCL)

The Point Cloud Library (*PCL*)[10] is an open source library intended for 3D images. It covers several algorithms and operations done on 3D point clouds, like clouds registration, building kd-trees, octrees, surface reconstruction, visualization, etc.

We used in this diploma thesis amongst others the following algorithms: Normal Estimation, Iterative Closest Point, RANSAC, Greedy Triangulation, Mesh approximation. These are only some of the used algorithms and not all.

8.4 Open Computer Vision Library (OpenCV)

The OpenCV library[9] can perform several operations on 2D images. We used this library while detecting edges in chapter 3.4. Several algorithms have been used for edge detection like Canny edge detection, find contours, Hough transform.

8.5 Robotics Operating System (ROS)

ROS[6] is a stack of software targeted at the robotics world. It has various packages for robots hardware and camera drivers, visualization modules, control and navigation modules. Gazebo simulator used in this thesis is also part of ROS

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Slides on hough transform. "Website", 2010. http://free.download4.org/Introduction---Chapter-1---The-HoughTransform-download-w342.pdf.

[2] Bluebotics. "Website", 2012. http://www.bluebotics.com/.

[3] Microsoft Kinect. "Website", 2012. http://www.xbox.com/en-US/KINECT.

[4] Nifti. "Website", 2012. http://www.nifti.eu/.

[5] OpenNi. "Website", 2012. http://www.openni.org.

[6] Robots Operating System(ROS). "Website", 2012. http://www.ros.org.

[7] Sick laser scanner. "Website", 2012. http://www.sick.com/group/EN/home/Pages/homepage1.aspx.

[8] Gazebo simulator. "Website", 2032. http://gazebosim.org/.

[9] OpenCV. "Website", 2032. http://opencv.willowgarage.com/wiki.

[10] Point Cloud Library (PCL). "Website", 2032. http://pointclouds.org.

[11] P. Ben-Tzvi, Shingo Ito, and A.A. Goldenberg. Autonomous Stair Climbing with Reconfigurable Tracked Mobile Robot. In *Robotic and Sensors Environments, 2007. ROSE 2007. International Workshop on*, pages 1–6, oct. 2007. doi: 10.1109/ROSE.2007.4373976.

[12] P.J. Besl and Neil D. McKay. A method for registration of 3-d shapes. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 14(2):239–256, 1992. ISSN 0162-8828. doi: 10.1109/34.121791.

[13] Gary Bradski. The opencv library. In *Dr. Dobb's*. 2000.

[14] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8(6):679 –698, nov. 1986. ISSN 0162-8828. doi: 10.1109/TPAMI.1986.4767851.

[15] R. Cupec, E. K. Nyarko, and D. Filko. Fast 2.5D Mesh Segmentation to Approximately Convex Surfaces. In *in Proceedings of the European Conference on Mobile Robots (ECMR), Orebro, Sweden*, 2011.

[16] Richard O. Duda and Peter E. Hart. Use of the Hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15(1):11–15, January 1972. ISSN 0001-0782. doi: 10.1145/361237.361242. URL `http://doi.acm.org/10.1145/361237.361242`.

[17] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981. ISSN 0001-0782. doi: 10.1145/358669.358692. URL `http://doi.acm.org/10.1145/358669.358692`.

[18] T. Fong, I. Nourbakhsh, and K. Dautenhahn. A survey of socially interactive robots. *Robotics and autonomous systems*, 42(3-4):143–166, 2003.

[19] Kristiyan Georgiev, Ross T. Creed, and Rolf Lakaemper. Fast plane extraction in 3D range data based on line segments. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 3808–3815, sept. 2011. doi: 10.1109/IROS.2011.6094916.

[20] German Institute for Norms, 2012. URL `http://www.din.de`.

[21] R.L. Graham. An efficient algorith for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132 – 133, 1972. ISSN 0020-0190. doi: 10.1016/0020-0190(72)90045-2. URL `http://www.sciencedirect.com/science/article/pii/0020019072900452`.

[22] Ahad Harati, Stefan Gchter, and Roland Siegwart. Fast range image segmentation for indoor 3d-slam. Technical report, Intelligent Autonomous Vehicles, Volume 6, Part 1, 2006.

[23] Dirk Holz and Sven Behnke. Fast Range Image Segmentation and Smoothing using Approximate Surface Reconstruction, Bilateral Filtering and Region Growing. In *IEEE International Conference on Robotics and Automation*, 2012.

[24] Dirk Holz, Stefan Holzer, Radu Bogdan Rusu, and Sven Behnke. Real-Time Plane Segmentation using RGB-D Cameras. In *RoboCup Symposium*, 2011 2011.

[25] S. Holzer, R.B. Rusu, M. Dixon, S. Gedikli, and N. Navab. Adaptive neighborhood selection for real-time surface normal estimation from organized point cloud data using integral images. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 2684–2689, 2012. doi: 10.1109/IROS.2012.6385999.

[26] A. Hoover, G. Jean-Baptiste, X. Jiang, P.J. Flynn, H. Bunke, D.B. Goldgof, K. Bowyer, D.W. Eggert, A. Fitzgibbon, and R.B. Fisher. An experimental comparison of range image segmentation algorithms. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 18(7):673–689, jul 1996. ISSN 0162-8828. doi: 10.1109/34.506791.

[27] X.Y. Jiang and H. Bunke. Fast Segmentation of Range Images into Planar Regions by Scan Line Grouping. In *Machine Vision and Applications*, pages 115–122, 1994.

[28] L. Matthies, Y. Xiong, R. Hogg, D.Zhu, A. Rakin, B. Kennedy, M. Hebert, R. Maclahlan, C. Won, T. Frost, G. Sukhatme, M. McHenry, and S. Goldberg. A Portable, autonomous, urban reconnaissance robot. *Elsevier Science B.V.*, pages 163–173, 2002.

[29] Benoit Morisset, Radu Bogdan Rusu, Aravind Sundaresan, Kris Hauser, Motilal Agrawal, Jean-Claude Latombe, and Michael Beetz. Leaving Flatland: Toward Real-Time 3D Navigation. In *IEEE International Conference on Robotics and Automation*, pages 3786–3739, 2009.

[30] Robin R. Murphy, Satoshi Tadokoro, Daniele Nardi, Adam Jacoff, Paolo Fiorini, Howie Choset, and Aydan M. Erkmen. *Springer handbook of robotics*, chapter Search and Rescue Robotics, pages 1151– 1173. Springer-Verlag New York Inc, 2008.

[31] V. Nguyen, A. Martinelli, N. Tomatis, and R. Siegwart. A comparison of line extraction algorithms using 2D laser rangefinder for indoor mobile robotics. In *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 1929–1934, aug. 2005. doi: 10.1109/IROS.2005.1545234.

[32] Franck Nielsen and Mariette Yvinec. An output-sensitive convex hull algorithm for planar objects. *International Journal of Computational Geometry and Applications*, 08(01):39–65, 1998. doi: 10.1142/S0218195998000047. URL `http://www.worldscientific.com/doi/abs/10.1142/S0218195998000047`.

[33] K. Nishiwaki, S. Kagami, Y. Kuniyoshi, M. Inaba, and H. Inoue. Toe joints that enhance bipedal and fullbody motion of humanoid robots. In *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, volume 3, pages 3105–3110, 2002. doi: 10.1109/ROBOT.2002.1013704.

[34] Andreas Nuechter, Kai Lingemann, and Joachim Hertzberg. Kurt3D A Mobile Robot for 3D Mapping of Environments. ELROB Technical Paper, January 2006.

[35] Andreas Nchter, Jan Elseberg, Peter Schneider, and Dietrich Paulus. Study of parameterizations for the rigid body transformations of the scan registration problem. *Computer Vision and Image Understanding*, 114(8):963 –

980, 2010. ISSN 1077-3142. doi: 10.1016/j.cviu.2010.03.007. URL http://www.sciencedirect.com/science/article/pii/S107731421000072X.

[36] Andreas Nchter, Oliver Wulf, Kai Lingemann, Joachim Hertzberg, Bernado Wagner, and Hartmut Surmann. 3d mapping with semantic knowledge. In *IN ROBOCUP INTERNATIONAL SYMPOSIUM*, pages 335–346, 2005.

[37] S. Osswald, A. Gorog, A. Hornung, and M. Bennewitz. Autonomous climbing of spiral staircases with humanoids. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 4844 –4849, sept. 2011. doi: 10.1109/IROS.2011.6094533.

[38] S. Osswald, J.-S. Gutmann, A. Hornung, and M. Bennewitz. From 3D point clouds to climbing stairs: A comparison of plane segmentation approaches for humanoids. In *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on*, pages 93–98, oct. 2011. doi: 10.1109/Humanoids.2011.6100836.

[39] Lutz Plümer. Gebüde Modellierung. 2010.

[40] J. Poppinga, N. Vaskevicius, A. Birk, and K. Pathak. Fast plane detection and polygonalization in noisy 3D range images. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 3378–3383, sept. 2008. doi: 10.1109/IROS.2008.4650729.

[41] J. Sankaranarayanan, Hanan Samet, and Amitabh Varshney. A fast k-neighborhood algorithm for large point-clouds. *Proceedings of the 3rd IEEE/Eurographics Symposium on Point-Based Graphics. ACM, Boston, MA, USA*, 2006/// 2006.

[42] N. Sato, F. Matsuno, T. Yamasaki, T. Kamegawa, N. Shiroma, and H. Igarashi. Cooperative task execution by a multiple robot team and its operators in search and rescue operations. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2004.

[43] Jörg Schmittwilken. *Attributierte Grammatiken zur Rekonstruktion und Interpretation von Fassaden.* Dissertation, Bonn University, 2012.

[44] Jörg Schmittwilken and Lutz Plümer. Model Selection for Composite Objects with Attribute Grammars. In *Proceedings of 12th AGILE International Conference on Geographic Information Science*, 2009.

[45] Jörg Schmittwilken and Lutz Plümer. MODEL-BASED RECONSTRUCTION AND CLASSIFICATION OF FACADE PARTS IN 3D POINT CLOUDS. In *Proceedings of PCV 2010 - ISPRS Technical Commission III Symposium on Photogrammetric Computer Vision and Image Analysis*, volume 38, 2010.

[46] Jörg Schmittwilken, Jens Saatkamp, Wolfgang Förstner, Thomas H. Kolbe, and Lutz Plümer. A Semantic Model of Stairs in Building Collars. In *Photogrammetrie, Fernerkundung, Geoinformation*, chapter Heft 6, pages 415–428. 2007.

[47] Jörg Schmittwilken, Dirk Dörschlag, and Lutz Plümer. Attribute grammar for 3D city models. In A. Krek, M. Rumor, S. Zlatanova, and E. M. Fendel, editors, *Urban and Regional Data Management*, chapter Attribute grammar for 3D city models, pages 49–58. CRC Press, 2009.

[48] J. Scholtz, B. Antonishek, and J. Young. A field study of two techniques for situation awareness for robot navigation in urban search and rescue. In *Robot and Human Interactive Communication, 2005. ROMAN 2005. IEEE International Workshop on*, pages 131–136. IEEE, 2005.

[49] Satoshi Suzuki and KeiichiA be. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1):32–46, 1985. ISSN 0734-189X. doi: 10.1016/0734-189X(85)90016-7. URL `http://www.sciencedirect.com/science/article/pii/0734189X85900167`.

[50] M. Buehler D. E. Koditschek U. Saranli. Rhex: A simple and highly mobile hexapod robot. In *Int. J. of Robotics Research*, pages 616–631, 2001.

[51] J. Vandorpe, H. Van Brussel, and H. Xu. Exact dynamic map building for a mobile robot using geometrical primitives produced by a 2D range finder. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 1, pages 901–908 vol.1, apr 1996. doi: 10.1109/ROBOT.1996.503887.

[52] Qing Xie and Xiaoyao Xie. Point cloud data reduction methods of octree-based coding and neighborhood search. In *Electronic and Mechanical Engineering and Information Technology (EMEIT), 2011 International Conference on*, volume 7, pages 3800–3803, aug. 2011. doi: 10.1109/EMEIT.2011.6023069.

Declaration

I hereby declare that this Diploma thesis has been written by me without any outside help other than the mentioned resources, that have been cited and marked properly as external resources.

Bonn, in

Moataz Elmasry