OBSTACLE DETECTION USING MICROSOFT KINECT

by

Niclas Zeller

Bachelor of Engineering, Karlsruhe University of Applied Sciences, 2011

A thesis

presented to Ryerson University

and Karlsruhe University of Applied Sciences

in partial fulfillment of the

requirements for the degree of

Master of Engineering

in the Programs of

Electrical and Computer Engineering

(Ryerson University)

and

Electrical Engineering

(Karlsruhe University of Applied Sciences)

Karlsruhe, Germany, 2013

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University and Karlsruhe University of Applied Sciences to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University and Karlsruhe University of Applied Sciences to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Obstacle detection using Microsoft Kinect

Master of Engineering 2013

Niclas Zeller

Electrical and Computer Engineering

at Ryerson University

and

Electrical Engineering

at Karlsruhe University of Applied Sciences

Abstract

This thesis presents the development of image processing algorithms based on a Microsoft Kinect camera system. The algorithms developed during this thesis are applied on the depth image received from Kinect and are supposed to model a three dimensional object based representation of the recorded scene. The motivation behind this thesis is to develop a system which assists visually impaired people by navigating through unknown environments. The developed system is able to detect obstacles in the recorded scene and to warn about these obstacles. Since the goal of this thesis was not to develop a complete real time system but to invent reliable algorithms solving this task, the algorithms were developed in MATLAB. Additionally a control software was developed by which depth as well as color images can be received from Kinect.

The developed algorithms are a combination of already known plane fitting algorithms and novel approaches. The algorithms perform a plane segmentation of the 3D point cloud and model objects out of the received segments. Each obstacle is defined by a cuboid box and thus can be illustrated easily to the blind person. For plane segmentation different approaches were compared to each other to find the most suitable approach. The first algorithm analyzed in this thesis is a normal vector based plane fitting algorithm. This algorithm supplies very accurate results but also has a high computation effort. The second approach, which was finally implemented, is a gradient based 2D image segmentation combined with a RANSAC plane segmentation (6) in a 3D points cloud. This approach has the advantage to find very small edges within the scene but also builds planes based on global constrains.

Beside the development of the algorithm results of the image processing, which are really promising, are presented. Thus the algorithm is worth to be improved by further development. The developed algorithm is able to detect very small but significant obstacles but on the other hand does not represent the scene too detailed such that the result can be illustrated accurately to a blind person.

Contents

List of Tables

List of Figures

Acronyms and Abbreviations

2D	two dimensional
2.5D	2.5 dimensional
3D	three dimensional
CMOS	complementary metal-oxide-semiconductor
CPU	central processing unit
FIR	finite impulse response
FOV	field of view
fps	frames per second
GUI	graphical user interface
HSV	hue, saturation, and value
IP	image processing
IR	infrared
LCD	liquid crystal display
LS	least square
LSB	least significant bit
LUT	lookup table
n-**D**	n-dimensional
MSB	most significant bit
MV	minimum variance
RANSAC	random sample consensus
RGB	red, green, blue
SDK	software development kit
TOF	time of flight
WPF	Windows Presentation Foundation

Chapter 1

Introduction

This thesis describes the development of software algorithms in 3D[1] image processing. In this work a Microsoft Kinect is used to record indoor environments. The implemented algorithms are supposed to use the Kinect images to build an object based 3D reconstruction of the recorded environment. Since Kinect has a depth image camera beside a usual color camera, this camera was used to retrieve spacial perception. The depth camera of Kinect does not supply real 3D illustrations of the recorded surrounding but depth information for the scene, recorded by the camera. Since those kind of images are not a real 3D image of the recorded scene, but a depth image from one point of view, these images are often called 2.5D[2] images. The developed algorithms are used to reconstruct the 3D environment as precise as possible out of the gained depth information. Therefore the implemented algorithm performs a plane segmentation and defines objects within the recorded scene. After the whole depth image is separated into single objects, these objects are displayed textual. The purpose of describing a recorded scenario by 3D shapes is to give blind people a visualization of their environment. Later, the system shall be able to warn blind people about unexpected and dangerous obstacles.

1.1 Motivation

Basically the motivation of the work described in this thesis is to "make blind people see". The goal is to develop a system, which is able to give blinds a visualization of their environment. Therefore the main goal is to output warnings about unexpected and dangerous obstacles, which can appear in the way of a blind. Of course there are already existing tools (like a white cane or a seeing-eye dog) which assist blind people to navigate in a foreign environment. Nevertheless, those aids are still not able to capture all kinds of obstacles a blind will be faced with. A white cane for example is only able to detect objects lying on the ground. A low hanging branch of a tree or a hanging shelf will not be recognized early enough by using a white cane. Another problem are steps going downstairs, since a blind will not hit it by her or his cane. A dog also has problems to see obstacles which are not on the floor, but far

---

[1] three dimensional

[2] 2.5 dimensional

above its own height. Beside that, dogs usually only can be used outside and so in indoor environments a blind person most time is on her or his own. Another handicap is that for using a white cane as well as be guided by a dog, the person has to use at least one of her or his hands and is not able to use both hands for other activities. To face these problems the idea was to develop a system which gives a blind person a cognition of her or his environment, for example by audio visualization. Other researchers had this idea already before. In the last two decades several systems were invented and are still developed for giving blind people a travel assistance (4). Nevertheless, all systems rather supply the user with a very primitive representation of its environment. System (7) for example just transforms a recorded depth image into an audio representation without any further object retrieval. Other systems do not give any perspective on the users environment but just warn for upcoming obstacles detected by sensor systems (e.g. Ultra Sound). Thus another motivation of this work is to give blind people a better perspective on their environment than the already existing systems and thus give a reliable feedback on upcoming obstacles.

1.2 Thesis objectives

The goal of this thesis was to develop a system, which is able to caution blind people about obstacles in their surroundings. Therefore obstacles should be defined by their dimensions and position in a 3D coordinate system centered at the blind person's position. The person's surrounding shall be recorded by a Microsoft Kinect sensor system, to receive such a 3D reconstruction. A transformation description has to be defined, which describes the relationship between the depth image recorded by Kinect and the real environment. Afterwards this recorded 3D representation has to be divided in single objects, which can be outputted as obstacles. It is not the aim to give a very detailed representation of the environment but obstacles, which are considered to be dangerous for a blind person have to be detected reliable. After the 3D scene is reconstructed and defined by separate objects, an output scheme has to be developed, which displays the most urgent obstacle to the blind person.

1.3 Fields of activity

During the work on this thesis, I faced problems in several different fields. The following listing gives a short overview of the different fields of activity. Since each single step in the development process will be discussed in this thesis in detail, no further explanation is given here.

1. Getting familiar with Microsoft Kinect sensor system
2. Calibrating depth and color image of Microsoft Kinect
3. Developing image processing algorithms and implementation in MATLAB
    - Depth image based segmentation
    - Object construction

4. Visualization of potential obstacles
5. Verification of the developed algorithms and the obstacle detection system

Chapter 2

Fundamentals

This chapter conveys some fundamentals needed to understand the topics discussed in this thesis. First, some basic geometric structures (points, straight lines, and planes) are defined, which are used to describe objects within a 3D space. Afterwards, some operations in a 3D space (e.g. distance calculations) are described. These operations make use of the prior defined geometric structures. Sections 2.3 covers geometric transformations in 2D[1] and 3D vector spaces (coordinate systems). Geometric transformations are needed to describe the relationships between different coordinate systems. For example, the relationship between image coordinates (in pixels) and the world coordinates (in cm). Beside that in section 2.4 some basics on linear estimation are discussed. In this thesis linear estimation is needed, e.g. to simplify the recoded point clouds by estimated regression planes.

2.1 Basic geometric structures in a 3D space

Points, lines, and planes are the basic geometric structures up to the second dimension. This basic geometric structures are very important for describing relations in a 3D space and thus the definition of these structures will be given for a 3D space in the following section.

2.1.1 Point

A point is a zero dimensional geometric element. That means it can be defined by a vector space of the order zero. Since a vector space of the order zero is not a space at all it only contains one element, which is the point itself. In a higher dimensional vector space a point is defined by a vector. This vector points from the origin of the vector space to the position of the point within the vector space. In a 3D

---

[1] two dimensional

space a point P is described by the vector $\vec{p}$ of the order three as given in equation 2.1.

$$P: \quad \vec{p} = \begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix} \tag{2.1}$$

In this thesis a point P will be either denoted by a vector $\vec{p}$ or as $p\,(x, y)$ in 2D, respectively $p\,(x, y, z)$ in a 3D vector space. The variables x, y, and z give the coordinates of the point P in the vector space.

2.1.2 Straight line

A straight line is a one dimensional geometric element. Thus a straight line defines a one dimensional vectors space. In a 3D vector space a straight line L is defined by a 3D support vector $\vec{s}$, which points from the vector space origin to a certain point on the line L and a direction vector $\vec{d}$ which points from $\vec{s}$ along the line L. The straight line L is defined as given in equation 2.2 in a 3D vector space. The value λ is a scalar, which can be any real number and thus for any real number λ the equation 2.2 results in a point $\vec{x}$ on the straight line L.

$$L: \quad \vec{x} = \vec{s} + \lambda \cdot \vec{d} =$$

$$\begin{pmatrix} x_g \\ y_g \\ z_g \end{pmatrix} = \begin{pmatrix} x_s \\ y_s \\ z_s \end{pmatrix} + \lambda \cdot \begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} \tag{2.2}$$

Since a straight line is a one dimensional figure, it can be totally defined by two points on this line. For example, P_1 and P_2, which are described by $\vec{p}_1$ and $\vec{p}_2$, are two points defining a straight line. Then the support vector $\vec{s}$ can be defined as $\vec{s} = \vec{p}_1$ (or $\vec{s} = \vec{p}_2$) and the direction vector $\vec{d}$ as $\vec{d} = \vec{p}_1 - \vec{p}_2$ or ($\vec{d}$ as $\vec{d} = \vec{p}_2 - \vec{p}_1$). The direction vector $\vec{d}$ often is standardized to a norm of one ($\|\vec{d}\| = 1$) and then will be denoted by $\vec{d}_0$.

$$\vec{d}_0 = \frac{\vec{d}}{\|\vec{d}\|} \tag{2.3}$$

2.1.3 Plane

A plane is a two dimensional geometric element and thus it defines a 2D vector space. In a 3D vector space a plane is defined as given in equation 2.4. That means that all points $\vec{x}$ for which equation 2.4 holds true are part of the plane S.

$$S: \quad a \cdot x + b \cdot y + c \cdot z + d = 0 \tag{2.4}$$

A plane S can also be defined by a support vector $\vec{s}$, which is a certain point on the plane and by a normal vector $\vec{n}$, which is defined as given in equation 2.5.

$$\vec{n} = \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} = \vec{\nabla}\left(a \cdot x + b \cdot y + c \cdot z + d\right) = \begin{pmatrix} \partial(a\cdot x+b\cdot y+c\cdot z+d)/\partial x \\ \partial(a\cdot x+b\cdot y+c\cdot z+d)/\partial y \\ \partial(a\cdot x+b\cdot y+c\cdot z+d)/\partial z \end{pmatrix} = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \tag{2.5}$$

$$\vec{n}_0 = \frac{\vec{n}}{\|\vec{n}\|} \tag{2.6}$$

The normal vector $\vec{n}$ often is standardized to a norm of one ($\|\vec{n}\| = 1$) and then will be denoted by $\vec{n}_0$. The normal vector $\vec{n}$ is a vector which stands perpendicular to the plane S itself and thus for all points $\vec{p}_i$ lying on the plane S, equation 2.7 holds true.

$$\vec{n}^T \cdot (\vec{p}_i - \vec{s}) = 0 \tag{2.7}$$

Since a plane is a 2D figure it can be totally defined by three points (P_1, P_2, and P_3) situated on this plane, defined by the vectors $\vec{p}_1$, $\vec{p}_2$, and $\vec{p}_3$. Therefore one point can be chosen as support vector $\vec{s}$ pointing onto the plane. The plane's normal vector $\vec{n}$ can be calculated as the vector cross product of two difference vectors ($\vec{d}_1$ and $\vec{d}_2$). This difference vectors can be determined out of the given three vectors ($\vec{p}_1$, $\vec{p}_2$, and $\vec{p}_3$) as given in equation 2.8.

$$\vec{n} = \vec{d}_1 \times \vec{d}_2 = (\vec{p}_i - \vec{p}_j) \times (\vec{p}_i - \vec{p}_k) \qquad \text{with} \quad i, j, k \in \{1, 2, 3\} \quad \text{and} \quad i \neq j \neq k \tag{2.8}$$

Out of equation 2.8 again the general form for a plane S, as given in equation 2.4, can be retrieved. Here the coefficients of the normal vector (x_n, y_n, and z_n) are equivalent to the coefficients a, b, and c in equation 2.4 and d is received by inserting one of the three points P_i (with $i \in \{1, 2, 3\}$) into the plane equation as given in equation 2.9.

$$d = -a \cdot x_p^{(i)} - b \cdot y_p^{(i)} - c \cdot z_p^{(i)} \qquad \text{with} \quad i \in \{1, 2, 3\} \tag{2.9}$$

2.2 Some operations in a 3D space

In this section some useful operations in a 3D space will be presented. These operations include the calculation of distances between the structures described in the section above. Beside that, the projection of points onto straight lines or planes and the calculation of crossing lines between two planes is described.

2.2.1 Minimum distance from a point to a straight line in a 3D space

The minimum distance d between a point P and a straight line L is the length of a vector $\vec{v}$, which stands perpendicular on the straight line and points to the point P. The point P will be described by

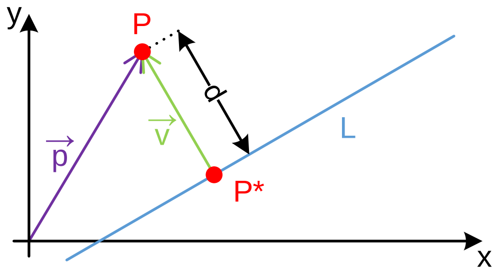


Figure 2.1: Minimum distance from a point P to any point P^* on a straight line L

a vector $\vec{p}$ and the straight line L by a support vector $\vec{s}$ and a direction vector $\vec{d}$. Figure 2.1 shows the minimum distance d between a point P and a straight line L for a 2D case. Nevertheless, the 2D case is equivalent to the 3D case. The easiest way to calculate the distance between P and L is to calculate a vector $\vec{u}$ which points from any point on the straight line L to the point P. Since the support vector $\vec{s}$ is already known, this point can be the point on the straight line L. Thus the vector $\vec{u}$ pointing from $\vec{s}$ to $\vec{p}$ is calculated as given in equation 2.10.

$$\vec{u} = \vec{p} - \vec{s} \tag{2.10}$$

By calculating the scalar product between the vector $\vec{u}$ and the standardized direction vector $\vec{d_0}$ the component of $\vec{u}$ along the straight line L, u_L is received. Since the vector $\vec{v}$ is supposed to stand perpendicular to the straight line L and thus also on $\vec{d}$, the distance d can be calculated by the Pythagorean theorem as given in equation 2.12.

$$u_L = \vec{u}^T \cdot \vec{d_0} \tag{2.11}$$

$$d = \sqrt{\|\vec{u}\|^2 - u_L^2} \tag{2.12}$$

2.2.2 Minimum distance from a point to a plane in a 3D space

Calculating the minimum distance from a point P to a plane S is very similar to calculating the distance between a point and a straight line. This distance d is again the length of a vector $\vec{v}$, which stands perpendicular on S and points to the point P. To calculate the distance d, a vector $\vec{u}$ is calculated, which points from any point on the plane S to the point P. Since in section 2.1.3 a support vector $\vec{s}$ was already defined, which points to any point on the plane S, this vector is used. Thus the vector $\vec{u}$ is calculated as given in equation 2.13.

$$\vec{u} = \vec{p} - \vec{s} \tag{2.13}$$

Since the length of the vector $\vec{v}$, which stands perpendicular to the plane S, is searched, this length is received by calculating the scalar product between the vector $\vec{u}$ and the standardized normal vector of the plane $\vec{n}_0$ as given in equation 2.14.

$$d = \vec{u}^T \cdot \vec{n}_0 \tag{2.14}$$

2.2.3 Projection of a point onto a straight line or a plane in a 3D space

It is often needed to project any point P onto the closest point P^* on a straight line L or a plane S. The projection of a point P onto a point P^* on a straight line L is shown in figure 2.1 for a 2D vector space. The vector $\vec{d^*}$ describing the point P^* can basically be derived from the definitions given in section 2.2.1 respectively section 2.2.2, since it is just the point on the straight line L respectively on the plane S, which has the closest distance to P and thus is calculated as given in equation 2.15.

$$\vec{p^*} = \vec{p} - \vec{v} \tag{2.15}$$

If the projection of the point P onto a plane is considered, the vector $\vec{v}$ is just the standardized normal vector $\vec{n}_0$ scaled by the distance d as given in equation 2.16.

$$\vec{v} = d \cdot \vec{n}_0 \tag{2.16}$$

For the projection of a point P onto a straight line L, a vector perpendicular to the direction vector $\vec{d}$ pointing into the direction of P is not already given. Thus the easiest way to calculate the vector of the projected point $\vec{p^*}$ is as given in equation 2.17. In this equation the vector $\vec{s}$ is the one that points onto the straight line L, which was prior selected and u_L is the component of the vector $\vec{u}$ along the straight line L.

$$\vec{p^*} = \vec{s} + u_g \cdot \vec{d_0} \tag{2.17}$$

2.2.4 Crossing line between two planes

The crossing between two planes S_1 and S_2 is a straight line L_c, which lays on both of the planes. This means that each point on the line L_c is also a point of plane S_1 as well as of plane S_2. For two planes there always exists not more than one crossing line. Only if both planes have got the same orientation and will not cross each other. The crossing line L_c is a straight line with a direction vector $\vec{d_c}$ orthogonal to the normal vectors of both planes ($\vec{n}_1$ and $\vec{n}_2$). Figure 2.2 shows the crossing line L_c (green) between plane S_1 (red) and plane S_2 (blue). The normal vectors of both planes are plotted onto the crossing line L_c, which stands perpendicular to them. Since the direction vector of L_c, $\vec{d_c}$ is orthogonal to $\vec{n}_1$ and $\vec{n}_2$ it can be calculated as their cross product as given in equation 2.18.

$$\vec{d_c} = \vec{n}_1 \times \vec{n}_2 \tag{2.18}$$

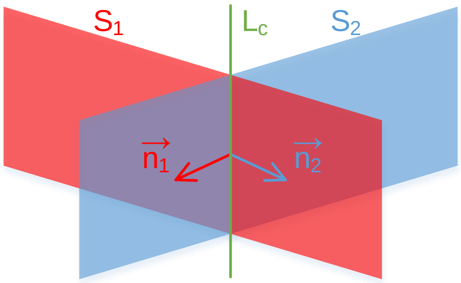


Figure 2.2: Crossing line L_c (green) between the planes S_1 (red) and S_2 (blue)

Beside the direction vector of L_c, $\vec{d_c}$, additionally the support vector of L_c, $\vec{s_c}$ has to be calculated, which has to be any point, which lays in both planes S_1 and S_2. Since $\vec{s_c}$ can be any point on the straight line L_c, one coordinate can be chosen free. Thus the z-component is set to zeros ($z_{sc} = 0$) and the system of equations derived from equations 2.19 and 2.20 as given equation 2.21 is received.

$$0 = a_1 \cdot x_{sc} + b_1 \cdot y_{sc} + c_1 \cdot z_{sc} + d_1$$

$$0 = a_1 \cdot x_{sc} + b_1 \cdot y_{sc} + d_1 \tag{2.19}$$

$$0 = a_2 \cdot x_{sc} + b_2 \cdot y_{sc} + c_2 \cdot z_{sc} + d_2$$

$$0 = a_2 \cdot x_{sc} + b_2 \cdot y_{sc} + d_2 \tag{2.20}$$

The x- and y-component of $\vec{s}_{sc}$ (x_{sc} and y_{sc}) then can be calculated by solving equation 2.21, which is derived from equations 2.19 and 2.20.

$$-\begin{pmatrix} d_1 \\ d_2 \end{pmatrix} = \begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \end{pmatrix} \cdot \begin{pmatrix} x_{sc} \\ y_{sc} \end{pmatrix} \tag{2.21}$$

Thus the support vector $\vec{s}_{sc}$ as given in equation 2.22 is received.

$$\vec{s}_{sc} = \begin{pmatrix} x_{sc} \\ y_{sc} \\ z_{sc} \end{pmatrix} = \frac{1}{a_1 \cdot b_2 - b_1 \cdot a_2} \cdot \begin{pmatrix} -d_1 \cdot b_2 + d_2 \cdot b_1 \\ d_1 \cdot a_2 - d_2 \cdot a_1 \\ 0 \end{pmatrix} \tag{2.22}$$

2.3 Geometric transformation

A geometric transformation is the projection from one (up to) 3D vector space into another. A special type of geometric transformation is the affine transformation, which will be discussed first. After introducing the affine transformation the projective transformation will be discussed. The projective transformation takes place for example in a distortion free projection process of a camera, by projecting a 3D space onto a 2D plane. This projection process is central projection and will be of major interest in this thesis. Projective transformations can be described in an elegantly way by homogeneous coordinates, which will be introduced in section 2.3.2.

2.3.1 Affine transformation

An affine transformation is a linear, geometric transformation in which straight lines and parallelisms are prevented but angles (e.g. between vectors) have the ability change. All affine transformations are invertible transformations. Translation, mirroring, rotation and scaling are affine transformations. In the following sections all four transformations will be described and will be defined in a 2D space first, since it is easier to visualize. Nevertheless, all those transformations can be adopted in a 3D space.

All transformations will be described as projections of a vector $\vec{v}$ into a vector $\vec{u}$ by a linear function $g(\vec{v})$ as given in equation 2.23. For all linear vector space transformations (as affine transformations are) the function $g(\vec{v})$ is always either a matrix multiplication, a vector addition, or both as given in equation 2.38 for the 2D case and respectively equation 2.39 for the 3D case.

$$\vec{u} = g(\vec{v}) \tag{2.23}$$

2.3.1.1 Translation

A translation is the movement of the coordinates by a certain amount and direction, which means that a translation is just the addition of a certain vector $\vec{t}$ to the vector $\vec{v}$ in the current vector space, as given in equation 2.24.

$$\vec{u} = \begin{pmatrix} x_u \\ y_u \end{pmatrix} = \vec{t} + \vec{v} = \begin{pmatrix} t_x \\ t_y \end{pmatrix} + \begin{pmatrix} x_v \\ y_v \end{pmatrix} \tag{2.24}$$

When a translation of a coordinate system by a vector $\vec{t}$ is performed, the origin of the new coordinate system is moved to the position $-\vec{t}$ of the prior one. The unit vectors of the new coordinate system will be similar to the old ones. Figure 2.3 shows a translation in a 2D space. In this figure the blue vector represents $\vec{v}$, the green (dashed) one $\vec{t}$, and the red one the resulting vector $\vec{u}$. The left part of the figure shows the new coordinate system (dotted lines) drawn into the old coordinate system (solid lines). In a 3D space all vectors are of the dimension 3 × 1 instead of 2 × 1 as given in equation 2.25.

$$\vec{u} = \begin{pmatrix} x_u \\ y_u \\ z_u \end{pmatrix} = \vec{t} + \vec{v} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} + \begin{pmatrix} x_v \\ y_v \\ z_v \end{pmatrix} \tag{2.25}$$

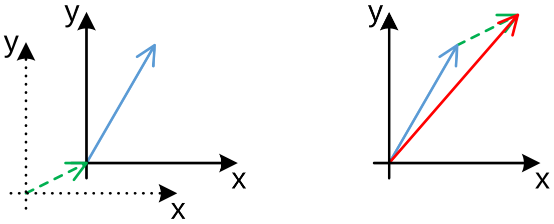


Figure 2.3: Translation of a 2D vector space by a vector $\vec{t}$ (green)

2.3.1.2 Mirroring

Mirroring is basically a special kind of scaling, where at least one scaling factor has a negative value. Here mirroring is defined as a function $g_M(\vec{v})$, which mirrors a vector space in the origin, along a coordinate axis, or at a plane spanned by two coordinate axis (only in 3D space possible) without changing the length of the unit vectors (all vector norms are preserved). Of course a mirroring can be performed at any other point, axis or plane in the vector space, but this can also be led back to a combination of translation, rotation, and mirroring. Equations 2.26 and 2.27 show the mathematical description of a mirroring at the y- respectively x-axis without any further scaling in a 2D space.

$$\vec{u} = \begin{pmatrix} x_u \\ y_u \end{pmatrix} = \boldsymbol{M}_y \cdot \vec{v} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \end{pmatrix} \tag{2.26}$$

$$\vec{u} = \begin{pmatrix} x_u \\ y_u \end{pmatrix} = \boldsymbol{M}_x \cdot \vec{v} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \end{pmatrix} \tag{2.27}$$

When a vector space is mirrored at the x-axis (in a 2D space) the unit vector representing the y-component $\vec{e}_y$ is just turned by 180° and points into the opposite direction ($\vec{e}_{yu} = -\vec{e}_{yv}$). Same happens to the unit vector representing the x-component $\vec{e}_x$ when the vector space is mirrored at the y-axis ($\vec{e}_{xu} = -\vec{e}_{xv}$). Equation 2.28 shows the mirroring in the origin of the coordinate system. This is basically the same as rotating the coordinate system by 180°.

$$\vec{u} = \begin{pmatrix} x_u \\ y_u \end{pmatrix} = \boldsymbol{M}_o \cdot \vec{v} = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \end{pmatrix} \tag{2.28}$$

Figure 2.4 shows a 2D vector space, which is mirrored along the y-axis. The left figure shows the new coordinate system (dotted lines) drawn into the prior one (solid lines). As one can see the y-axis stays unchanged and the x-axis is flipped. The mirroring in a 3D space is basically the same as in a 2D space. Only the matrix size increases from 2×2 to 3×3. Beside that a 3D space can be mirrored in a

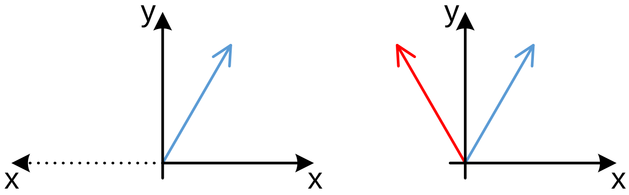


Figure 2.4: Mirroring of a vector along the y-axis in a 2D space

point, along an axis and also at a plane as already mentioned. Equation 2.29 gives an example for a 3D mirroring at the origin of the coordinate system. Therefore all three unit vectors ($\vec{e}_x$, $\vec{e}_y$, and $\vec{e}_z$) have to be flipped.

$$\vec{u} = \begin{pmatrix} x_u \\ y_u \\ z_u \end{pmatrix} = \boldsymbol{M}_o \cdot \vec{v} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \\ z_v \end{pmatrix} \tag{2.29}$$

Equation 2.30 gives an example for mirroring the coordinate system at the x-axis. In this case the unit vectors in direction of the y- and z-axis ($\vec{e}_y$, and $\vec{e}_z$) have to be flipped.

$$\vec{u} = \begin{pmatrix} x_u \\ y_u \\ z_u \end{pmatrix} = \boldsymbol{M}_x \cdot \vec{v} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \\ z_v \end{pmatrix} \tag{2.30}$$

Equation 2.31 shows the mathematical description of mirroring the coordinate system at the x-y-plane. In this case only the unit vector in z-direction $\vec{e}_z$ is flipped.

$$\vec{u} = \begin{pmatrix} x_u \\ y_u \\ z_u \end{pmatrix} = \boldsymbol{M}_{xy} \cdot \vec{v} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \\ z_v \end{pmatrix} \tag{2.31}$$

2.3.1.3 Rotation

A rotation turns a vector space around a certain point (2D) or axis (3D) by a certain angle ϕ. First, only the 2D case is considered. For simplicity, only the rotation around the origin of the coordinate system is defined. Rotations around other points can be realized in combination with translation. Equation 2.32 gives the definition of a 2D rotation of the vector space by the angle ϕ.

$$\vec{u} = \begin{pmatrix} x_u \\ y_u \end{pmatrix} = \boldsymbol{R} \cdot \vec{v} = \begin{pmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \end{pmatrix} \tag{2.32}$$

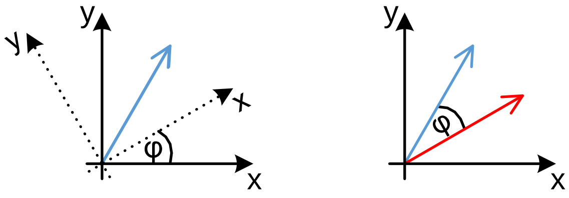


Figure 2.5: Rotation of a 2D vector space by the angle ϕ around the origin

Since the coordinate system is rotated by the angle ϕ, this results in a rotation of all vectors by the angle $-\phi$, which is shown in figure 2.5 where ϕ is a positive angle and the difference angle $\Delta\phi$ from $\vec{v}$ to $\vec{u}$ is negative ($\Delta\phi = -\phi$). In a 3D space a rotation can be performed around any coordinate axis. Thus three different rotation matrices ($\boldsymbol{R}_x$, $\boldsymbol{R}_y$, and $\boldsymbol{R}_z$) are defined. Equation 2.33 gives the definition for a rotation around the x-axis by the angle α, equation 2.34 for a rotation around the y-axis by the angle β, and equation 2.35 for a rotation around the z-axis by the angle γ.

$$\vec{u} = \begin{pmatrix} x_u \\ y_u \\ z_u \end{pmatrix} = \boldsymbol{R}_x \cdot \vec{v} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \\ z_v \end{pmatrix} \tag{2.33}$$

$$\vec{u} = \begin{pmatrix} x_u \\ y_u \\ z_u \end{pmatrix} = \boldsymbol{R}_y \cdot \vec{v} = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \\ z_v \end{pmatrix} \tag{2.34}$$

$$\vec{u} = \begin{pmatrix} x_u \\ y_u \\ z_u \end{pmatrix} = \boldsymbol{R}_z \cdot \vec{v} = \begin{pmatrix} \cos\gamma & \sin\gamma & 0 \\ -\sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \\ z_v \end{pmatrix} \tag{2.35}$$

One can see that for each of the three transformations in a 3D space always one unit vector is unchanged. This is the vector around which the rotation is performed. Out of a combination of all three rotation matrices the vector space can be turned into any direction.

2.3.1.4 Scaling

In scaling the unit vectors of a vector space are stretched or shrunk by a certain factor. Therefore the cases can be considered that all unit vectors are multiplied by the same factor or by different factors. Since the mirroring is already defined for flipping unit vectors in scaling only positive scaling factors

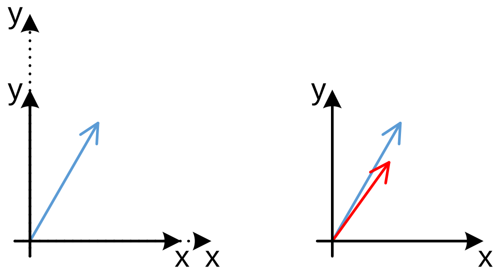


Figure 2.6: Scaling in x- and y-direction in a 2D space by factors $s_x < 1$ and $s_y < 1$

are considered. In the case of equal scaling factors the directions of all vectors in the vector space stay unchanged and only their norm is modified. If the scaling factors are different for each unit vector, the scaling changes besides the norm of a vector in the vector space also its direction. Equation 2.36 describes the scaling in general (different scaling factors) for a 2D space.

$$\vec{u} = \begin{pmatrix} u_x \\ u_y \end{pmatrix} = \boldsymbol{S} \cdot \vec{v} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \end{pmatrix} \tag{2.36}$$

If s_x and s_y are equal, the matrix $\boldsymbol{S}$ can be replaced by a scalar s, for which $s = s_x = s_y$ holds true. Figure 2.6 shows a scaling for factors $s_x < 1$, and $s_y < 1$, but $s_x \neq s_y$. Figure 2.7 shows a scaling for factors $s_x > 1$, and $s_y > 1$ and $s_x \neq s_y$. In the 3D case an additional scaling factor for the z-axis s_z exists and thus the scaling leads to equation 2.37.

$$\vec{u} = \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix} = \boldsymbol{S} \cdot \vec{v} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \\ z_v \end{pmatrix} \tag{2.37}$$

An affine transformation also can be a combination of the operations listed above. Therefore the general expression as given in equation 2.38 is received for the 2D case.

$$\vec{u} = \begin{pmatrix} u_x \\ u_y \end{pmatrix} = \boldsymbol{A} \cdot \vec{v} + \vec{b} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \tag{2.38}$$

In the 3D case a general affine transformation is defined as given in equation 2.39

$$\vec{u} = \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix} = \boldsymbol{A} \cdot \vec{v} + \vec{b} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{2.39}$$

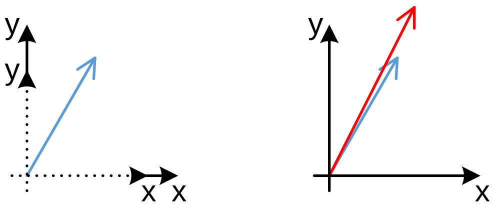


Figure 2.7: Scaling in x- and y-direction in a 2D space by factors $s_x > 1$ and $s_y > 1$

Since geometric transformations are described by matrix operations, it is obvious that they are not commutative. If multiple transformations are performed to a vector space, the result will be different when the order of the transformations is changed. For example, performing a rotation by a Matrix $\boldsymbol{R}$ and then a scaling by a Matrix $\boldsymbol{S}$ will result in a different vector space than performing firstly the scaling and then the rotation. The single operations have to be ordered in the equation from right to left in the sequence they shall be performed. E.g. the coordinates shall be translated, rotated, and then scaled. Therefore the operations have to be done in the same order as shown in equation 2.40.

$$\vec{u} = \boldsymbol{A} \cdot \vec{v} + \vec{b} = \boldsymbol{S} \cdot \boldsymbol{R} \cdot \left[\vec{v} + \vec{t}\right] = \boldsymbol{S} \cdot \boldsymbol{R} \cdot \vec{v} + \boldsymbol{S} \cdot \boldsymbol{R} \cdot \vec{t} \tag{2.40}$$

2.3.2 Homogeneous coordinates

Homogeneous coordinates are used to solve transformation problems of an n-dimensional space in a space of the dimension $n + 1$, which makes it possible to describe some nonlinear transformations in an n-D[2] space as linear transformations in a $(n + 1)$-D space. Homogeneous coordinates are used, for example, to perform a projective transformation (projection), which will be described in section 2.3.3 in detail. Equation 2.41 gives the relationship between homogeneous (x', y', and k) and "common" Cartesian coordinates (x, and y) of a 2D space.

$$\begin{pmatrix} x \\ y \end{pmatrix} \rightarrow \begin{pmatrix} x' \\ y' \\ k \end{pmatrix} \qquad \text{with} \qquad \begin{matrix} x' = k \cdot x \\ y' = k \cdot y \end{matrix} \tag{2.41}$$

---

[2] n-dimensional

Equation 2.42 shows the relationship between homogeneous and Cartesian coordinates for the 3D space, which is described in four homogeneous coordinates.

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \begin{pmatrix} x' \\ y' \\ z' \\ k \end{pmatrix} \qquad \text{with} \qquad \begin{aligned} x' &= k \cdot x \\ y' &= k \cdot y \\ z' &= k \cdot z \end{aligned} \tag{2.42}$$

Affine transformations can also be described by homogeneous coordinates. In affine transformations the value k is not modified. For simplicity k can be set to $k = 1$ for affine transformations. Equation 2.43 shows the general form of the affine transformations with homogenious coordinates.

$$\begin{pmatrix} x'_u \\ y'_u \\ z'_u \\ k_u \end{pmatrix} = \begin{pmatrix} x_u \\ y_u \\ z_u \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \\ z_v \\ 1 \end{pmatrix} \tag{2.43}$$

The coefficients a_{ij} (for $i, j \in \{1, 2, 3\}$) in equation 2.43 are equivalent to the matrix coefficients a_{ij} (for $i, j \in \{1, 2, 3\}$) in equation 2.39, and the coefficients a_{i4} (for $i \in \{1, 2, 3\}$) in equation 2.43 are equivalent to the vector coefficients b_i (for $i = 1, 2, 3$) in equation 2.39. If the transformation matrix does not perform any translation, the coefficients in the last column, a_{14}, a_{24}, and a_{34} are zero. The following equations (2.44 to 2.49) show all affine transformations defined before in a 3D space in homogeneous coordinates.

Translation

$$\begin{pmatrix} x_u \\ y_u \\ z_u \\ 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & t_x \\ 0 & 0 & 0 & t_y \\ 0 & 0 & 0 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \\ z_v \\ 1 \end{pmatrix} \tag{2.44}$$

Mirroring

$$\begin{pmatrix} x_u \\ y_u \\ z_u \\ 1 \end{pmatrix} = \begin{pmatrix} \pm 1 & 0 & 0 & 0 \\ 0 & \pm 1 & 0 & 0 \\ 0 & 0 & \pm 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \\ z_v \\ 1 \end{pmatrix} \tag{2.45}$$

Rotation

Equation 2.46 shows the rotation matrix for homogeneous coordinates around the x-axis by an angle α.

$$\begin{pmatrix} x_u \\ y_u \\ z_u \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \\ z_v \\ 1 \end{pmatrix} \tag{2.46}$$

Equation 2.47 shows the rotation matrix for homogeneous coordinates around the y-axis by an angle β.

$$\begin{pmatrix} x_u \\ y_u \\ z_u \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\beta & 0 & -\sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \\ z_v \\ 1 \end{pmatrix} \tag{2.47}$$

Equation 2.49 shows the rotation matrix for homogeneous coordinates around the z-axis by an angle γ.

$$\begin{pmatrix} x_u \\ y_u \\ z_u \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \\ z_v \\ 1 \end{pmatrix} \tag{2.48}$$

Scaling

$$\begin{pmatrix} x_u \\ y_u \\ z_u \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \\ z_v \\ 1 \end{pmatrix} \tag{2.49}$$

The affine transformation in a 2D space has basically the same structure with only one dimension less. The main reason why homogeneous coordinates are used are not affine transformations but projective transformations. For this case k is not constant but depends on the coordinates x_v, y_v, and z_v. Projective transformations will be discussed in the following section.

2.3.3 Projective transformation

A projective transformation describes the projection of one projective space into another. Since here the case of projecting a 3D scenery onto a 2D image plane is of special interest, the considered projective transformations are those from an affine (or euclidean) space into a projective space. The euclidean space as well as the affine space is a special case of a projective space. As discussed before, in an affine space parallelism of straight lines is prevented when this space is transformed by an affine transformation. This holds not true anymore for a projective transformation. In a projective space all parallel lines will

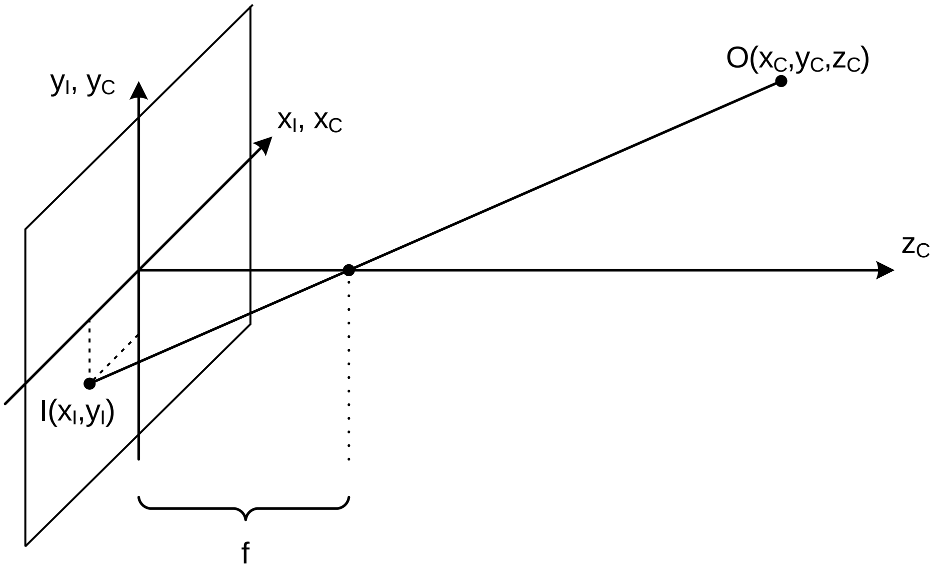


Figure 2.8: Central projection of point O to point I

intersect each other in certain points (vanishing points). These points are usually at a certain finite position, except for the case of affine or euclidean spaces. In this cases parallel lines will intersect in infinity. Projective transformations can be described elegantly by homogeneous coordinates. Here, the vector space is converted into a space of higher order and thus a projective transformation leads to a linear projection, which can be described by a matrix multiplication.

In order to model the imaging process of a camera some simplifications are made to make it describable by a projective transformation. Therefore the case of an image projection by a pinhole camera is considered instead of a projection by an optical lens. This model represents a real camera very well for points, which are situated in great distance from the camera in relation to the focal length f of the camera lens. Potential distortions by the camera lens are also neglected for this model. If the distortions are considerably high, the projective transformation is usually calculated firstly and afterwards the distortions are equalized in the 2D image. Figure 2.8 shows the image projection of a pinhole camera. For simplicity it is considered that the image plane lies on the x-y plane of the 3D camera coordinates (euclidean coordinates) described by x_C, y_C and z_C. Thus the image coordinates axis (x_I, y_I) point into the same direction as the camera coordinates axis (x_C, y_C). A point $O(x_C, y_C, z_C)$ is projected through the camera aperture onto a point $I(x_I, y_I)$ on the image plane. The distance between the image plane and the camera aperture (lens plane) is considered to be equivalent to the focal length f of an usual optical camera. This relation holds true for the assumed case that f is much smaller than the distance of the object point according to the camera. This projection is called central projection. Equation 2.50

gives the mathematical description of the central projection.

$$\begin{pmatrix} k \cdot x_I \\ k \cdot y_I \\ k \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -\frac{1}{f} & 1 \end{pmatrix} \cdot \begin{pmatrix} x_C \\ y_C \\ z_C \\ 1 \end{pmatrix} \tag{2.50}$$

The central projection given in 2.50 has its vanishing point for lines parallel to the z_C-axis in the origin of the Image ($x_I = 0$, $y_I = 0$). The further the objects are away from the image plane the smaller their projection will appear on the image. The equations 2.51 and 2.52 give the relationship between image coordinates x_I and y_I and camera coordinates x_C, y_C and z_C. In this equations it can be seen how x_I and y_I decrease when z_C increases.

$$x_I = \frac{f \cdot x_C}{f - z_C} \tag{2.51}$$

$$y_I = \frac{f \cdot y_C}{f - z_C} \tag{2.52}$$

A central projection can be combined with any other affine transformation to calculate the relation between image coordinates (x_I, y_I) of a camera and any world coordinates (x_W, y_W, z_W). Therefore the general equation is given in 2.53. For a usual projection onto an image plane the z-coordinate of the image z_I has no meaning but is inserted for completeness. This coordinate will be used for a projective transformation of a depth image, which will be described in chapter 4 *Camera calibration*.

$$\begin{pmatrix} k \cdot x_I \\ k \cdot y_I \\ k \cdot z_I \\ k \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} \tag{2.53}$$

2.4 Linear estimation

In parameter estimation an estimator is a function, which describes the relationship between some observation $\vec{y}$ and the estimated value or vector $\hat{\vec{b}}$ as given in equation 2.54.

$$\hat{\vec{b}} = g(\vec{y}) \tag{2.54}$$

An estimator is considered to be linear, if the function $g(\vec{y})$ is a linear function. Therefore $g(\vec{y})$ has the form as given in equation 2.55. $\boldsymbol{D}$ is called the estimation matrix since it projects $\vec{y}$ onto $\hat{\vec{b}}$.

$$\hat{\vec{b}} = \boldsymbol{D} \cdot \vec{y} + \vec{d} \tag{2.55}$$

The estimation function $g\left(\vec{y}\right)$ mostly is calculated based on some cost function $C\left(\vec{b},\hat{\vec{b}}\right)$, which is a function of the estimated value (or vector) $\hat{\vec{b}}$ and the real value (or vector) $\vec{b}$. The goal of most estimators is to minimize the expectation of the cost function $E\{C\left(\vec{b},\hat{\vec{b}}\right)\}$. The most common cost functions are the squared error (equation 2.56) or the absolute error (equation 2.57).

$$C_{SE}\left(\vec{b},\hat{\vec{b}}\right) = \left(\hat{\vec{b}}-\vec{b}\right)^2 \tag{2.56}$$

$$C_{AE}\left(\vec{b},\hat{\vec{b}}\right) = \left|\hat{\vec{b}}-\vec{b}\right| \tag{2.57}$$

An estimator is basically qualified on two properties. One is, whether the estimator is a biased or unbiased estimator. Equation 2.58 gives the definition for an unbiased estimation and equation 2.59 for a biased estimation.

$$E\{\hat{\vec{b}}|\vec{b}\} = E\{g\left(\vec{y}\right)|\vec{b}\} = \vec{b} \tag{2.58}$$

$$E\{\hat{\vec{b}}|\vec{b}\} = E\{g\left(\vec{y}\right)|\vec{b}\} = \vec{b}+\vec{\Delta} \tag{2.59}$$

This definitions mean that for an unbiased estimation the expectation of the estimated value $\hat{\vec{b}}$ is equal to the real value $\vec{b}$. For a biased estimator the expectation of the estimated value $\hat{\vec{b}}$ is not equal to $\vec{b}$, which means that not even on average the right value is estimated. If the bias $\vec{\Delta}$ is dependent on $\vec{b}$ itself and not a constant, it is called unknown. If the bias $\vec{\Delta}$ is constant it is called known. For a known bias the estimation can be led to an unbiased estimation by subtracting $\vec{\Delta}$ from the estimation result. Another quality of an estimator is the variance of the estimation $\sigma_{\hat{\vec{b}}^2}$ as given in equation 2.60.

$$\sigma_{\hat{\vec{b}}^2} = E\{\left(\hat{\vec{b}}-\vec{b}\right)\cdot\left(\hat{\vec{b}}-\vec{b}\right)^T|\vec{b}\} \tag{2.60}$$

For a good estimator the estimation variance $\sigma_{\hat{\vec{b}}^2}$ will be low. One way to design an estimator is that based on some statistics of the random value or vector $\vec{b}$, which has to be estimated and the random observation $\vec{y}$. One such estimator is the Bayes' Estimator. This estimator minimizes the posterior expectation of the cost function $E\{C\left(\vec{b},\hat{\vec{b}}\right)|\vec{y}\}$. One problem of such estimators is that one must have at least some knowledge about the probability distribution $f_{\vec{B},\vec{Y}}(\vec{b},\vec{y})$. In this thesis no Bayes' estimators were used and thus are not described in more detail. An extensive description about Bayes' estimators can be found in (2).

Another type of linear estimation is the linear regression. Linear regression is used to estimate the relationship $\vec{b}$ between some output y and some input $\vec{x}$ in presence of an unknown error e. Here is only the case for an input vector but an output scalar considered. Therefore equation 2.61 gives the relationship between $\vec{x}$ and y based on $\vec{b}$.

$$y = \vec{x}^T \cdot \vec{b} + e = \begin{pmatrix} x_1 & x_2 & \cdots & x_N \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix} + e \tag{2.61}$$

The vector $\hat{\vec{b}}$ is not estimated based on statistics of the random variables but on a sequence of observations. Thus a linear relationship between $\hat{\vec{b}}$ and the sequence of observations $\vec{y}$ is defined based on a certain optimization criteria. Thus an estimation function as given in equation 2.62 is received where the element $y^{(i)}$ of $\vec{y}$ is the output of the i-th observation.

$$\hat{\vec{b}} = \begin{pmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \vdots \\ \hat{b}_N \end{pmatrix} = \boldsymbol{D} \cdot \vec{y} + \vec{d} = \begin{pmatrix} D_{11} & D_{12} & \cdots & D_{1M} \\ D_{21} & D_{22} & \cdots & D_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ D_{N1} & D_{N2} & \cdots & D_{NM} \end{pmatrix} \cdot \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(M)} \end{pmatrix} + \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_M \end{pmatrix} \tag{2.62}$$

The vector $\vec{y}$ has the length M, which is the number of observations. The matrix $\boldsymbol{D}$ and the vector $\vec{d}$ project these M observations onto the N coefficients of the vector $\hat{\vec{b}}$. Since different $y^{(i)}$ are observed for different input signals $\vec{x}^{(i)}$ the matrix $\boldsymbol{D}$ has to be dependent on the input signal $\vec{x}$. If it is considered that the random error e has no mean ($E\{e\} = 0$), the conditions of equation 2.63 and equation 2.64 have to be true for an unbiased estimation. $\boldsymbol{X}$ is the matrix of input vectors $\vec{x}^{(i)}$ for the single observations as given in equation 2.65 where $\vec{x}^{(i)}$ is the input vector of the i-th observation.

$$\boldsymbol{D} \cdot \boldsymbol{X} = \boldsymbol{I} \tag{2.63}$$

$$\vec{d} = \vec{0} \tag{2.64}$$

$$\boldsymbol{X} = \begin{pmatrix} \vec{x}^{(1)} & \vec{x}^{(2)} & \cdots \vec{x}^{(M)} \end{pmatrix}^T = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_N^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_N^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(M)} & x_2^{(M)} & \cdots & x_N^{(M)} \end{pmatrix} \tag{2.65}$$

2.4.1 LS[3] estimator

One optimization criteria for a linear regression is the least square criteria. In this criteria the squared error e^2 is minimized. Out of 2.61 the error e can be written as a function of the output y, the input $\vec{x}$,

---

[3] least square

and $\vec{b}$ as given in equation 2.66. Thus equation 2.67 gives the error $e^{(i)}$ of i-th observation.

$$e = y - \vec{b}^T \cdot \vec{x} \tag{2.66}$$

$$e^{(i)} = y^{(i)} - \vec{b}^T \cdot \vec{x}^{(i)} \tag{2.67}$$

Over all observations the sum of squared errors S_e is received as given in equation 2.68.

$$S_e = \sum_{i=1}^{M} {e^{(i)}}^2 = \begin{pmatrix} e^{(1)} & e^{(2)} & \cdots & e^{(M)} \end{pmatrix} \cdot \begin{pmatrix} e^{(1)} \\ e^{(2)} \\ \vdots \\ e^{(M)} \end{pmatrix} \tag{2.68}$$

S_e is basically the cost function C of the LS estimator, which has to be minimized. The result of the LS estimator $\hat{\vec{b}}_{LS}$ is the vector $\vec{b}$ for which S_e gets minimum.

$$\hat{\vec{b}}_{LS} = \arg\left[\min_{\vec{b}} (S_e)\right] \tag{2.69}$$

Out of equation 2.69 the equation 2.70 results for the estimation of $\vec{b}$ based on the LS criteria.

$$\hat{\vec{b}}_{LS} = \boldsymbol{D} \cdot \vec{y} = \left[\boldsymbol{X}^T \cdot \boldsymbol{X}\right]^{-1} \cdot \boldsymbol{X}^T \cdot \vec{y} \tag{2.70}$$

Under consideration of equations 2.63 and 2.64 it can be obtained out of equation 2.70 that the LS estimator is an unbiased estimator since $\vec{d} = 0$ and equation 2.71 holds true.

$$\boldsymbol{D} \cdot \boldsymbol{X} = \left[\boldsymbol{X}^T \cdot \boldsymbol{X}\right]^{-1} \cdot \boldsymbol{X}^T \cdot \boldsymbol{X} \overset{!}{=} \boldsymbol{I} \tag{2.71}$$

There exist various further linear regression schemes, which will not be discussed here. Nevertheless, the LS estimator is used for parameter estimation since it is the optimal estimator for uncorrelated Gaussian distributed disturbance. Another method is the MV[4] estimation. The MV criteria needs prior knowledge about the correlation between the single observations and thus can achieve better estimation results. For the case that all observations are uncorrelated and the variance of the error of each observation is equal the result of the MV estimator will be the same as that of the LS estimator. More on linear regression can be found in (3).

---

[4] minimum variance

Chapter 3

Microsoft Kinect sensor system

Microsoft Kinect is a sensor system consisting of a RGB[1] camera[2], a depth image camera and a mircrophone array. Kinect was originally developed for video gaming purpose (in combination with Microsoft XBox 360). The system was developed by Microsoft in cooperation with PrimeSense. Already shortly after the commercial release of Kinect in November 2010, Windows and Linux drivers for the system were developed by some computer hackers and published in open source projects. Two of these open source projects are, for example, OpenKinect and OpenNI. In June 2011 Microsoft released its own Kinect SDK[3] including Windows 7 drivers. Additionally Microsoft released the Kinect for Windows Sensor, especially for desktop applications. Basically, there is no difference between the original Kinect and the Kinect for Windows. The only difference is that Kinect for Windows has additionally to the default depth range a near depth range, especially for desktop applications. In desktop application the user sits usually in front of a desk and thus is closer to the Kinect sensor. Both Kinect models have a tilt motor, which can be used to change the Kinect's vertical angle. The tilt angle of the Kinect is measured by an accelerometer. Since SDK version 1.6 the sensor data of the accelerometer can be read out. Figure 3.1 shows the front view of Kinect.

3.1 Microphone array

The microphone array assembled in Kinect consists of four single microphones, which are arranged along the front side of the sensor. This microphone array can be used to detect the position of a sound source. Since the microphone array was not used during this thesis, it will not be described in more detail.

---

[1] red, green, blue

[2] the RGB color model divides color information in the components red, green, and blue

[3] software development kit

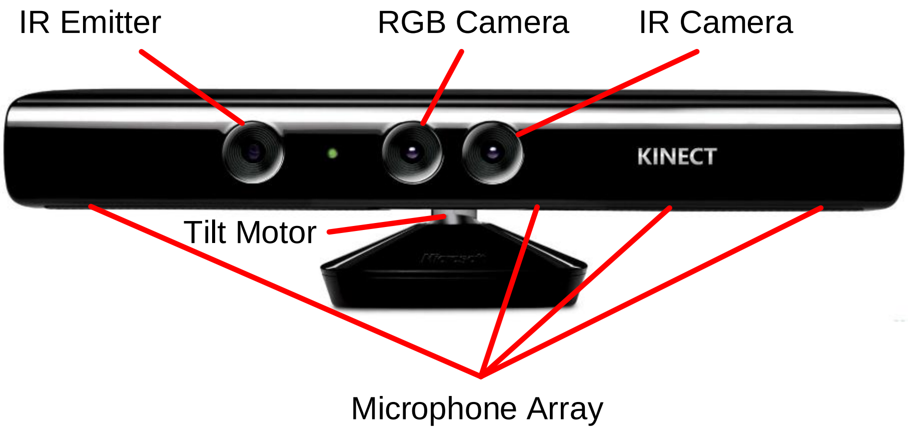


Figure 3.1: Front view of Microsoft Kinect for Windows sensor system

3.2 RGB camera

The RGB camera is arranged central on Kinect's front side as shown in figure 3.1. The assembled camera senor is a CMOS[4] image sensor MT9M112 produced by Micron (9). The whole camera system is completely integrated on one chip. It has a maximum resolution of 1.3 mega pixels (1280 × 1024) at a frame rate of 15 fps. At 640 × 512 the sensor reaches a frame rate of 30 fps. Each pixel has a size of 2.8 µm × 2.8 µm. The Kinect sensor provides only a maximum resolution of 1280 × 960 at a frame rate of 12 fps in RGB format with 24 bit color depth. Additionally, the RGB camera integrated in Kinect can be operated at a resolution of 640 × 480 with different color formats and frame rates (RGB with 24 bit color depth at 30 fps, YUV[5] at 15 fps, and YUV RAW at 15 fps). According to the specification the RGB camera has a focal length of 2.9 mm and a field of view of 43° vertical and 57° horizontal.

Parameter	**Value**
Pixel size	2.8 µm × 2.8 µm
Max. Resolution	1280 pixel × 960 pixel
Max. sample rate	30 fps
FOV[6]	ca. 43° × 57°
focal length f	2.9 mm

Table 3.1: RGB camera specification

---

[4] complementary metal-oxide-semiconductor

[5] YUV is a color model consisting of one intensity and two color channels

3.3 Depth image camera

More interesting than the RGB camera is the depth image camera of Kinect. This camera supplies an image, where each pixel does not carry color information but depth information. The interesting thing of the depth image camera system used for Kinect is that it only uses one IR[7] camera instead of a stereo camera system. The advantage of a stereo camera system is that it supplies both, color and depth information (the Kinect depth camera only supplies depth information). A disadvantage is the high computation effort, which is needed to find similarities in the two stereo images to get the depth information out of it. This high computation effort probably is the reason why Kinect does not use a stereo camera system. Kinect uses an IR camera, as already mentioned, in combination with an IR projector to retrieve depth information out of the scene. The IR camera is arranged right beside the RGB camera and the IR projector is placed in a distance of about 7 cm to the IR camera. The IR projector projects a pseudo random (but known) pattern onto the scene. This pattern is recorded by the IR camera. Since the camera records the pattern from a different angle of view, there will be a distortion of the recorded pattern based on the 3D position of the reflecting objects. E.g. if the pattern is reflected by an object, which is far away from the projector, it will be more stretched then for reflections on closer objects. Beside that, if the pattern is reflected on planes, which are not parallel to the projector's plane, the pattern will be distorted. Since the projection pattern is known by Kinect, the depth information can be calculated directly out of the distortion by triangulation. Therefore no similarities in both images have to be found first as it is in a stereo camera system. Only the already known pattern has to be correlated with the recorded IR image. In figure 3.2 the principle of the IR projection is visualized. There exist already a lot of publications about analysis of the Kinect depth sensor (e.g. (8; 10; 11)), hence in this thesis no further research on accuracy analysis was done. All given information in the following sections rely on specifications given by Microsoft and already published works.

3.3.1 IR pattern

In (10) the IR pattern is analyzed on its properties very well. The pseudo random pattern is generated by some kind of grating, which scatters the IR light source into a defined pattern. (10) shows that the whole pattern consists of one small pattern in the center, which then repeatedly is arranged eight times around it. Since the pattern has a pseudo random (white) structure, it has very appropriate correlation properties and thus each speckle in the pattern can be determined very well by correlation.

3.3.2 Specifications

The IR sensor used in the system is the MT9M001 from the company Aptina (1). It has a resolution of 1280 × 1024 at a frame rate of 30 fps. The pixel data is supplied with a resolution of 10 bit. In the Kinect system, again, only 1280 pixel × 960 pixel are used and a 2 × 2 pixel binning is performed. The pixel binning results in an effective pixel size of 10.4 µm × 10.4 µm. Kinect offers three different resolutions for the depth image (80 × 60, 320 × 240, and 640 × 480), which run all at a frame rate of 30 fps. At all

---

[7] infrared

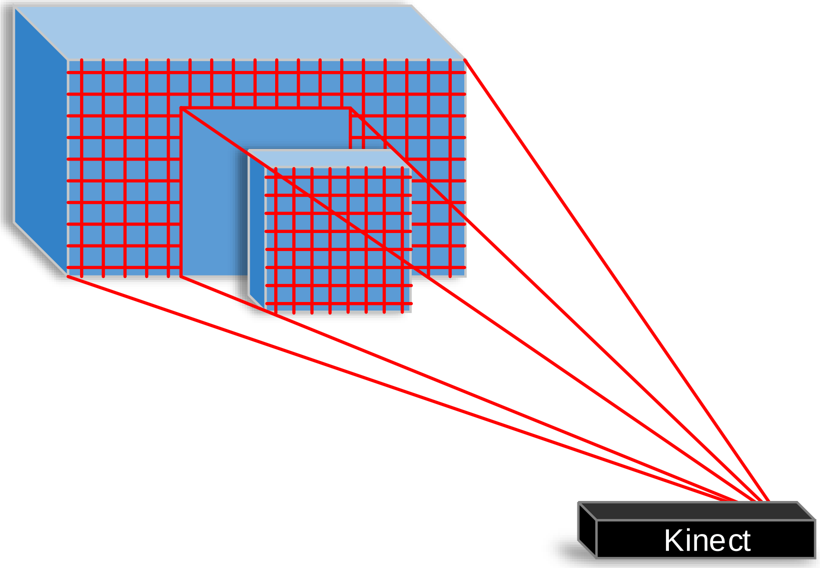


Figure 3.2: Visualization of the depth image camera principle

resolutions the depth information is quantized by 12 bit (unsigned). But tests showed that the actual resolution at far distances is much lower. The comes especially from interpolation of the depth value of undefined pixels out of the values of its neighbored pixels. In the default mode the depth information ranges from about 0.8 m to 4 m and in the near mode from 0.5 m to 3 m. The focal length of the camera is given with 5.9 mm and thus the field of view is almost the same as that of the RGB camera.

Parameter	**Value**
Pixel size	10.4 µm × 10.4 µm
Max. Resolution	640 pixel × 480 pixel
Max. sample rate	30 fps
FOV	ca. 43° × 57°
focal length f	5.9 mm
Depth quantization	12 bit
Depth range (near mode)	ca. 0.5 m to 3 m
Depth range (far mode)	ca. 0.8 m to 4 m

Table 3.2: Depth image camera specification

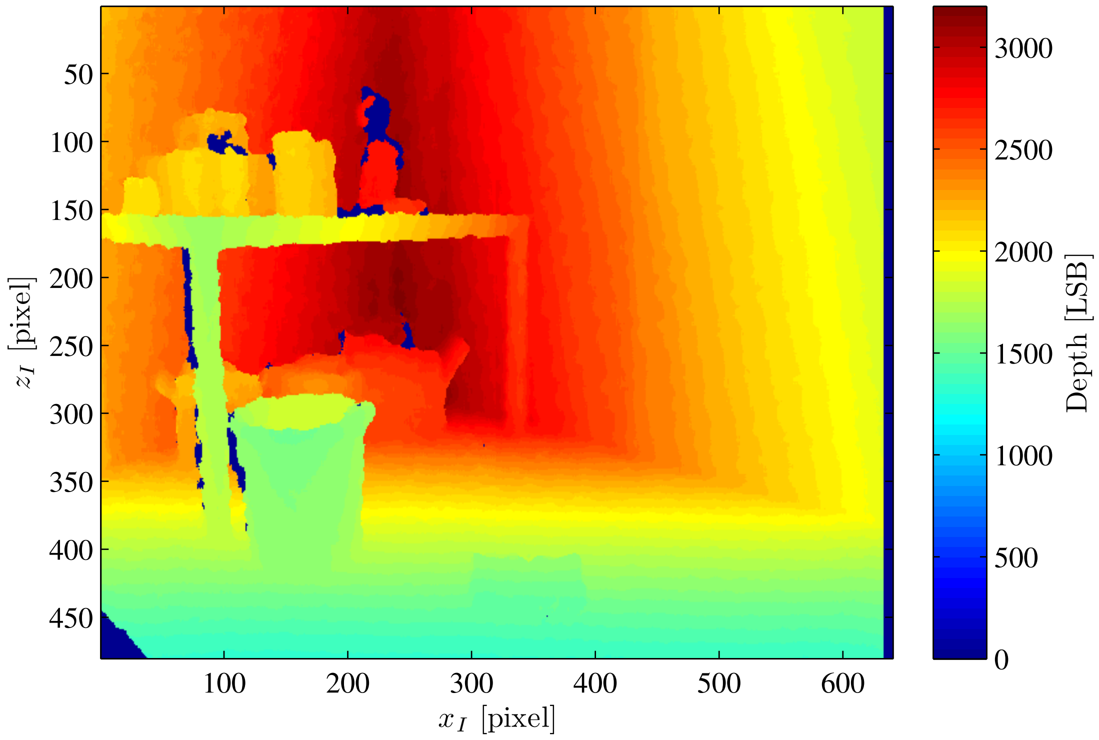


Figure 3.3: Depth image of Microsoft Kinect (640 × 480)

3.3.3 Depth image

As mentioned before, the depth image camera supplies depth information for each pixel with a resolution of 12 bit. Figure 3.3 shows a sample of such a depth image. In the image pixels with small depth values are blue and pixels with high depth values are red. The dark blue areas have got the depth value -1, which means they do not carry depth information. The reason that these areas do not carry any depth information is the lack of IR reflection in the camera direction. This can either result when the surface is shadowed from the IR pattern by any other object or when not enough IR light is reflected, which is based on the orientation and texture of the surface. Glossy or matt black surfaces, for example, basically do not reflect any light into the camera direction and therefore are inappropriate for depth determination with Kinect. Points, which are in a too close distance to the Kinect sensor are also represented by the value -1. Points, which are to far away from the sensor have got the maximum depth value $(2^{12}-1)$. Some experiments showed that the accuracy of the depth information decreases with increasing depth. While at about 0.5 m depth the accuracy is in a range of about ±5 mm, it decreases to about ±2.5 cm at 3 m depth. Since the density of the IR pattern also decreases with increasing depth, the depth information is interpolated at high depth values. That causes further inaccuracy for high depth values and a jumping depth information of neighbored pixels.

3.4 Kinect SDK

Beside the official Microsoft Kinect SDK there exist some open source projects for Kinect software development, as already mentioned. For this thesis the official SDK was used since it offers a really good online support and supports all needed functions (Read out of depth and RGB image). Beside those needed functionality a lot more tools are offered. The current version is Kinect SDK 1.6, which offers libraries for C# and C++ programming. SDK 1.6 offers the possibility to read out the RGB and the depth image. Beside that also the raw IR image can be read out. It offers a skeletal tracking mode, which returns 3D coordinates for each body joint. The SDK supports speech recognition as well as face tracking and offers accelerometer data of the Kinect sensor. These are basically the main features offered by the Kinect SDK. Additionally to the SDK a toolkit is offered by Microsoft. This toolkit contains sample code and demonstrations of different Kinect functions.

The development of Software using the Microsoft Kinect SDK is very simple. Basically all functions to control the Kinect and to read out the data from it are implemented in one class in the SDK library. This class is called `KinectSensor` and can be used in a C# as well as in C++ program. For all Kinect devices, which are connected to the computer, an object of the class `KinectSensor` is created automatically. Kinect itself as well as single functions, like the depth or RGB camera, can be activated or deactivated just by calling a member function of the `KinectSensor` class. For both, the depth camera as well as the RGB camera exist objects in `KinectSensor` by which the cameras can be controlled and images can be read out. These objects are of the class `ColorStream`, respectively `DepthStream` and are used for example to set the image resolution or format. The images streams recorded by the cameras can be received by events, which have to be set prior. Single images also can be received by calling a function. Beside the two objects for the color and depth image stream, which are of major interest for this thesis, there exist several other classes to control the different functionality. E.g. `SkeletonSrteam` supplies all functions needed to retrieve skeleton information of persons in front of Kinect. This object already supplies all body joints defined by 3D coordinates. Other functions are e.g. to read out the microphone array, which already gives the position of a detected sound source or to map the color and depth image onto each other.

Chapter 4

Camera calibration

To reconstruct the 3D environment out of the recorded images, a relationship between the environment and images has to be defined. This relationship is divided into two parts. One is the intrinsic camera orientation and the other the extrinsic camera orientation.

4.1 Intrinsic parameters

The intrinsic camera orientation is described by the intrinsic parameters. These parameters describe the relationship between image coordinates (x_I, and y_I) and camera coordinates (x_C, y_C, and z_C). For a pinhole camera, where the imaging process is described by a central projection, the only intrinsic parameters are the focal length f and the coordinates (x_{I0}, and y_{I0}) of the image focal point I_0. In figure 2.8 the image focal point I_0 is equal to the origin of the image coordinate system $I(0,0)$ thus $x_{I0} = 0$, and $y_{I0} = 0$.

In the case of a real camera, distortions caused by the lens occur, which effect the projection process from camera coordinates into image coordinates and thus also have to be considered as intrinsic parameters. For the distortion different models can be assumed. Here, only a circular distortion is considered. That means that the distortion is only dependent on the distance r to the distortion center D_0, which in this case is assumed to be the image focal point I_0. Thus the radial distance r is defined by equation 4.1. The radial optical distortion Δd is dependent on the radial distance r. Δd can either be estimated out of measurements or be described by an analytical function $\Delta d = f(r)$.

$$r = \sqrt{(x_I - x_{I0})^2 + (y_I - y_{I0})^2} \tag{4.1}$$

Since the image is a sampled (space discrete) image, it consist of a finite number of points. Thus for each image point $I(x_I, y_I)$ the correction factors Δx_I and Δy_I can be calculated as given in equation 4.2. Out of the distorted image point $I(x_I, y_I)$ and the correction factors Δx_I and Δy_I the coordinates

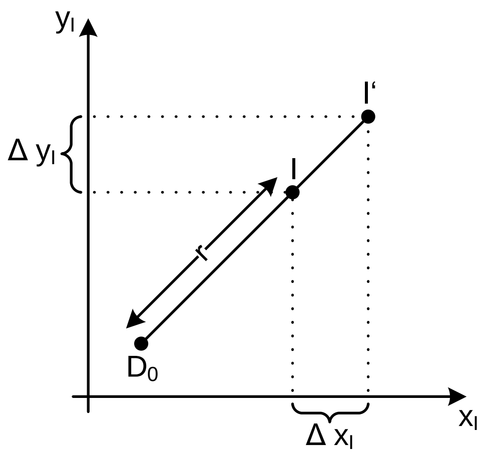


Figure 4.1: Equalizing a circular image distortion with distortion center D_0, distorted image point I, and equalized image point I'

of the corrected image point $I'(x_{I'}, y_{I'})$ can be calculated as shown in equation 4.3.

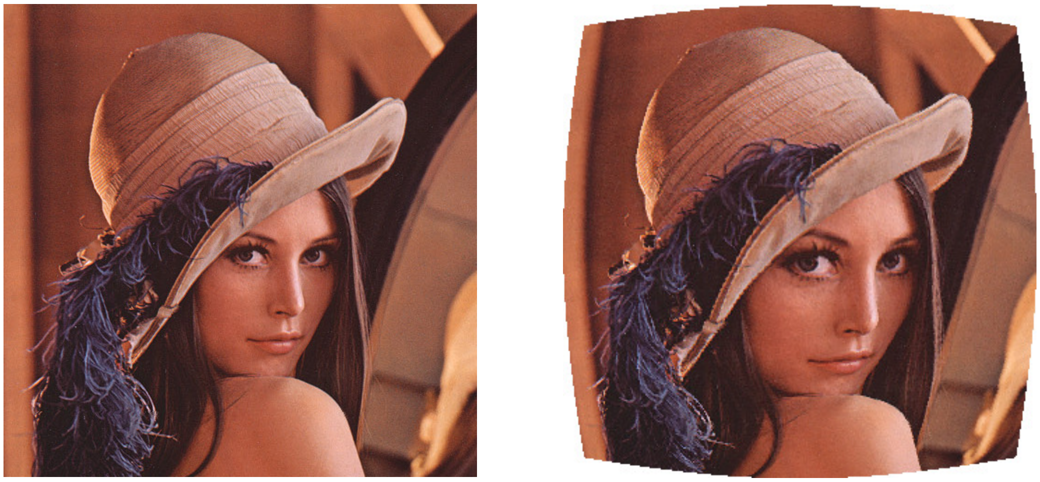
$$\Delta x_I = \frac{x_I - x_{I0}}{r} \cdot \Delta d(r) \qquad \Delta y_I = \frac{y_I - y_{I0}}{r} \cdot \Delta d(r) \tag{4.2}$$

$$x_{I'} = x_I + \Delta x_I \qquad y_{I'} = y_I + \Delta y_I \tag{4.3}$$

Figure 4.1 shows how the undistorted point I' results from the distorted image point I and the correction factors Δx_I and Δy_I. Figure 4.2 shows an example of such a circular image distortion where figure 4.2(a) is the undistorted and figure 4.2(b) the circular distorted image.

The relationship between the undistorted image point $I'(x_{I'}, y_{I'})$ and the corresponding object point $O(x_C, y_C)$ defined in camera coordinated, now can be described again by a central projection. Beside circular distortions also general radial distortions can occur. This means that the distortion is not constant on a circular curve around the distortions center (as circular distortion) but on an elliptical curve. Another important distortion is the tangential distortion, which is not only dependent on the radial distance r but also on the image coordinates (x_I and y_I) itself. Nevertheless, the distortion only can be equalized if the distortion function $\Delta d(x_I, y_I)$ is known. The method described by Tsai (12) as well as the one described by Zhang (13) are very popular to estimate lens distortions and the intrinsic parameters. Nevertheless, both methods consider the tangential distortions as insignificant.

Some experiments with Kinect have shown that the distortions of both, depth and RGB camera are very little. Thus lens distortions are negligible for the purpose described in this thesis. So for the Kinect's camera calibrations the distortions were not considered and the projection from camera into image coordinates is considered to be an ideal central projection.



(a) undistorted image

(b) image with circular distortion

Figure 4.2: Circular image distortion with distortion function $\Delta d(r) = 0.003 \cdot r^2$

4.2 Extrinsic parameters

The extrinsic parameters describe the relationship between camera coordinates (x_C, y_C, and z_C) and world coordinates (x_W, y_W, and z_W). In general world coordinates are defined based on any absolute point and orientation in a 3D space. Since both, the camera coordinates vector space and the world coordinates vector space are affine 3D spaces, the extrinsic parameters are basically an affine transformation matrix $\boldsymbol{A}$. In general, this transformation is described in homogenous coordinates and thus $\boldsymbol{A}$ results in a 4×4 matrix as given in 4.4.

$$\begin{pmatrix} x_C \\ y_C \\ z_C \\ 1 \end{pmatrix} = \boldsymbol{A} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} \tag{4.4}$$

To determine the extrinsic parameters of a projection process the position and orientation of the camera coordinate system has to be known and thus can be described in world coordinates. Since the camera coordinates system is usually not known, often the relationship between world coordinates (x_W, y_W, and z_W) and image coordinates (x_I, and y_I) is determined by calibration patterns. Out of the relationship between image and word coordinates the extrinsic parameters can be calculated when the intrinsic parameters of the camera are known. Except the third row of matrix $\boldsymbol{A}$ since the image usually does not carry any distance information.

For the work presented in this thesis we were not explicitly interested in the camera coordinate system and extrinsic or intrinsic parameters. The calibration should only be used to build a defined relationship between image coordinates and world coordinates. The world coordinates defined here are no real world coordinates since they will be attached to the Kinect sensor and thus will move when the Kinect is moving. The world coordinates we defined here basically will be in relation to the users positions and thus will describe distances from this position.

In the following section the calibration of the RGB camera will be described. The depth camera calibration is basically the same as the RGB image calibration but with consideration of depth information. The depth image calibration then will be described in section 4.4.

4.3 RGB camera calibration

The goal of the RGB camera calibration is to describe the relationship between a defined world coordinate system in centimeters and the image coordinate system in pixel. As already discussed, the lens distortion on the imaging process is negligible and thus the relationship between world and image coordinates can be described by a combination of affine transformations and a central projection. Thus the projection can be described by a transformation matrix $\boldsymbol{A}_{RGB}$ as given in equation 4.5.

$$\begin{pmatrix} k \cdot x_I \\ k \cdot y_I \\ k \cdot z_I \\ k \end{pmatrix} = \boldsymbol{A}_{RGB} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} \tag{4.5}$$

Since there is no z_I component in the image coordinates, the third row of the matrix $\boldsymbol{A}_{RGB}$ will be undefined. Thus the transformation between image coordinates (x_I, and y_I) and world coordinates (x_W, y_W, and z_W) also could be described just by a 3×4 matrix. For the depth image calibration there is a third image coordinate, the depth and thus the third row of the transformation matrix can be calculated. To keep the analogy between depth image and RGB image calibration the transformation is defined as a 4×4 matrix and the third row is just kept undefined.

The calibration method described here uses calibration points. Therefore a pattern with points in defined positions in world coordinates is recorded by the camera and its position in the image is determined. Thus, out of a set of measurements the transformation matrix $\boldsymbol{A}_{RGB}$ can be estimated. For the estimation of $\boldsymbol{A}_{RGB}$ the LS method as described in section 2.4.1 is used. The LS estimator minimizes the measurement error e for independent Gaussian distributed errors. If there is no systematical error in the measurement, the error usually can be assumed to be Gaussian. In the measurement described here actually a systematic error occurs since the image is sampled. Nevertheless, this systematic error is assumed to be negligible for the accuracy needed in this work. The distortion, which was neglected for this calibration model also results in a systematical error.

To apply the LS algorithm on the measurement points a linear relationship between input and output values has to be determined. Equation 4.5 gives a linear relationship between $\{k \cdot x_I, k \cdot x_I, k\}$ and $\{x_W, y_W, z_W\}$, but since k can not be measured, the LS estimator can not be applied just on this relationship.

Since k is not a measurable value and we also do not care about the value k itself, equation 4.5 is divided by a_{44} as given in equation 4.6.

$$\begin{pmatrix} {}^{k}\!/_{a_{44}} \cdot x_I \\ {}^{k}\!/_{a_{44}} \cdot y_I \\ {}^{k}\!/_{a_{44}} \cdot z_I \\ {}^{k}\!/_{a_{44}} \end{pmatrix} = \begin{pmatrix} {}^{a_{11}}\!/_{a_{44}} & {}^{a_{12}}\!/_{a_{44}} & {}^{a_{13}}\!/_{a_{44}} & {}^{a_{14}}\!/_{a_{44}} \\ {}^{a_{21}}\!/_{a_{44}} & {}^{a_{22}}\!/_{a_{44}} & {}^{a_{23}}\!/_{a_{44}} & {}^{a_{24}}\!/_{a_{44}} \\ {}^{a_{31}}\!/_{a_{44}} & {}^{a_{32}}\!/_{a_{44}} & {}^{a_{33}}\!/_{a_{44}} & {}^{a_{34}}\!/_{a_{44}} \\ {}^{a_{41}}\!/_{a_{44}} & {}^{a_{42}}\!/_{a_{44}} & {}^{a_{43}}\!/_{a_{44}} & 1 \end{pmatrix} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} \tag{4.6}$$

$$\begin{pmatrix} k^* \cdot x_I \\ k^* \cdot y_I \\ k^* \cdot z_I \\ k^* \end{pmatrix} = \boldsymbol{A}^*_{RGB} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} = \begin{pmatrix} a^*_{11} & a^*_{12} & a^*_{13} & a^*_{14} \\ a^*_{21} & a^*_{22} & a^*_{23} & a^*_{24} \\ a^*_{31} & a^*_{32} & a^*_{33} & a^*_{34} \\ a^*_{41} & a^*_{42} & a^*_{43} & 1 \end{pmatrix} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} \tag{4.7}$$

As already discussed, the third row of $\boldsymbol{A}_{RGB}$ and thus also of $\boldsymbol{A}^*_{RGB}$ will be undefined. Therefore the following three equations (4.8 to 4.10) are received out of equation 4.7 to describe the relation between image and world coordinates.

$$k^* \cdot x_I = a^*_{11} \cdot x_W + a^*_{12} \cdot y_W + a^*_{13} \cdot z_W + a^*_{14} \tag{4.8}$$

$$k^* \cdot y_I = a^*_{21} \cdot x_W + a^*_{22} \cdot y_W + a^*_{23} \cdot z_W + a^*_{24} \tag{4.9}$$

$$k^* = a^*_{41} \cdot x_W + a^*_{42} \cdot y_W + a^*_{43} \cdot z_W + 1 \tag{4.10}$$

Equation 4.10 can be inserted for k^* in equations 4.8 and 4.9 and equations 4.8 and 4.9 then can be solved for x_I respectively y_I as given in equations 4.11 and 4.12. These results in two functions where the outputs x_I respectively y_I are linear dependent on some input values, which are known functions of x_W, y_W, z_W, x_I, and y_I. This linear dependency is defined by 11 unknown coefficients and thus at least 11 equations are needed to calculate these coefficients. Since for each calibration point two equations are received, at least six points have to be measured.

$$x_I = a^*_{11} \cdot x_W + a^*_{12} \cdot y_W + a^*_{13} \cdot z_W + a^*_{14} - a^*_{41} \cdot x_W \cdot x_I - a^*_{42} \cdot y_W \cdot x_I - a^*_{43} \cdot z_W \cdot x_I \tag{4.11}$$

$$y_I = a^*_{21} \cdot x_W + a^*_{22} \cdot y_W + a^*_{23} \cdot z_W + a^*_{24} - a^*_{41} \cdot x_W \cdot y_I - a^*_{42} \cdot y_W \cdot y_I - a^*_{43} \cdot z_W \cdot y_I \tag{4.12}$$

Because of the presence of measurement errors, lot more than six measurement points are taken and the coefficients are estimated by the LS estimator. Out of the above given equations (4.11 and 4.12) the vector of measurement outputs $\vec{y}$ is received as given in equation 4.13. Here $x_I^{(n)}$ respectively $y_I^{(n)}$ is the image component of the n-th measurement in x respectively y direction.

$$\vec{y} = \begin{pmatrix} x_I^{(1)} \\ y_I^{(1)} \\ x_I^{(2)} \\ y_I^{(2)} \\ \vdots \\ x_I^{(N)} \\ y_I^{(N)} \end{pmatrix} \tag{4.13}$$

The matrix coefficients a^*_{ji} are ordered in a vector $\vec{b}$ row by row which is given in equation 4.14.

$$\vec{b} = \begin{pmatrix} a^*_{11} \\ a^*_{12} \\ a^*_{13} \\ a^*_{14} \\ a^*_{21} \\ a^*_{22} \\ a^*_{23} \\ a^*_{24} \\ a^*_{41} \\ a^*_{42} \\ a^*_{43} \end{pmatrix} \tag{4.14}$$

Based on the order of the matrix coefficients a^*_{ji} in the vector $\vec{b}$ the matrix of inputs $\boldsymbol{X}$ has the form given in equation 4.15, where $x_W^{(n)}$, $y_W^{(n)}$, and $z_W^{(n)}$ are the world coordinates for the n-th measurement point and $x_I^{(n)}$ and $y_I^{(n)}$ respectively the image coordinates.

$$\boldsymbol{X} = \begin{pmatrix} x_W^{(1)} & y_W^{(1)} & z_W^{(1)} & 1 & 0 & 0 & 0 & 0 & -x_W^{(1)}x_I^{(1)} & -y_W^{(1)}x_I^{(1)} & -z_W^{(1)}x_I^{(1)} \\ 0 & 0 & 0 & 0 & x_W^{(1)} & y_W^{(1)} & z_W^{(1)} & 1 & -x_W^{(1)}y_I^{(1)} & -y_W^{(1)}y_I^{(1)} & -z_W^{(1)}y_I^{(1)} \\ x_W^{(2)} & y_W^{(2)} & z_W^{(2)} & 1 & 0 & 0 & 0 & 0 & -x_W^{(2)}x_I^{(2)} & -y_W^{(2)}x_I^{(2)} & -z_W^{(2)}x_I^{(2)} \\ 0 & 0 & 0 & 0 & x_W^{(2)} & y_W^{(2)} & z_W^{(2)} & 1 & -x_W^{(2)}y_I^{(2)} & -y_W^{(2)}y_I^{(2)} & -z_W^{(2)}y_I^{(2)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_W^{(N)} & y_W^{(N)} & z_W^{(N)} & 1 & 0 & 0 & 0 & 0 & -x_W^{(N)}x_I^{(N)} & -y_W^{(N)}x_I^{(N)} & -z_W^{(N)}x_I^{(N)} \\ 0 & 0 & 0 & 0 & x_W^{(N)} & y_W^{(N)} & z_W^{(N)} & 1 & -x_W^{(N)}y_I^{(N)} & -y_W^{(N)}y_I^{(N)} & -z_W^{(N)}y_I^{(N)} \end{pmatrix} \tag{4.15}$$

By the vectors $\vec{y}$ and $\vec{b}$ and the matrix $\boldsymbol{X}$ defined above the realtion between measurement input and output is given in equation 4.16. Here all measurement errors are described by the error vector $\vec{e}$.

$$\vec{y} = \boldsymbol{X} \cdot \vec{b} + \vec{e} \tag{4.16}$$

As already described in section 2.4.1 *LS estimator* the estimated coefficients vector $\hat{\vec{b}}$ is calculated as given in equation 4.17 out of $\vec{y}$ and $\boldsymbol{X}$.

$$\hat{\vec{b}} = \begin{pmatrix} \hat{a}^*_{11} \\ \hat{a}^*_{12} \\ \hat{a}^*_{13} \\ \hat{a}^*_{14} \\ \hat{a}^*_{21} \\ \hat{a}^*_{22} \\ \hat{a}^*_{23} \\ \hat{a}^*_{24} \\ \hat{a}^*_{41} \\ \hat{a}^*_{42} \\ \hat{a}^*_{43} \end{pmatrix} = \left[\boldsymbol{X}^T \cdot \boldsymbol{X}\right]^{-1} \cdot \boldsymbol{X}^T \cdot \vec{y} \tag{4.17}$$

Out of the LS estimation the transformation matrix $\hat{A}^*_{RGB}$, which defines the relation between world and image coordinates is received as given in equation 4.18. As already mention are the coefficients $\hat{a}^*_{31}$, $\hat{a}^*_{32}$, $\hat{a}^*_{33}$, and $\hat{a}^*_{34}$ undefined.

$$\hat{\boldsymbol{A}}^*_{RGB} = \begin{pmatrix} \hat{a}^*_{11} & \hat{a}^*_{12} & \hat{a}^*_{13} & \hat{a}^*_{14} \\ \hat{a}^*_{21} & \hat{a}^*_{22} & \hat{a}^*_{23} & \hat{a}^*_{24} \\ \hat{a}^*_{31} & \hat{a}^*_{32} & \hat{a}^*_{33} & \hat{a}^*_{34} \\ \hat{a}^*_{41} & \hat{a}^*_{42} & \hat{a}^*_{43} & 1 \end{pmatrix} \tag{4.18}$$

Since the third row of the transformation matrix $\hat{\boldsymbol{A}}^*_{RGB}$ is undefined, the matrix is not invertible. Thus the transformation matrix can be used to calculate the image coordinates x_I and y_I for a certain point in the world coordinates but not vise versa.

Figure 4.3 shows a calibration image recorded by Kinect's RGB camera. In this image three calibration plates are placed on certain points in the world coordinate system. The defined world coordinate system has its origin right on the table central underneath the Kinect. The unit vector of the world coordinate system defining the y component, $\vec{e}_{Wy}$ stand perpendicular to the table's surface. The unit vector defining the z component $\vec{e}_{Wz}$ stands perpendicular to the image plane of the Kinect camera. The unit vector $\vec{e}_{Wx}$ of course has to be orthogonal to $\vec{e}_{Wy}$ and $\vec{e}_{Wz}$ and shows in the recorded image from left to right. Since the black squares which can be seen on the calibration plates have defined sizes and positions in the 3D space, the corners of these squares can be used as calibration points for the RGB camera calibration. The corresponding image coordinates in pixel ($x_I^{(n)}$ and $y_I^{(n)}$) are the measurement outputs. Thus, out of the image seen in figure 4.3 72 calibration points can be obtained. Nevertheless, the calibration points shown in the image are not enough since they lie all on the same plane and so the matrix $\boldsymbol{X}^T \cdot \boldsymbol{X}$ will be close to singular. There will be linear dependencies between the rows of the matrix and thus it will not be invertible. Because of this more calibration images have to be recorded such that the calibration points are distributed all over the 3D space. The more points will be used,

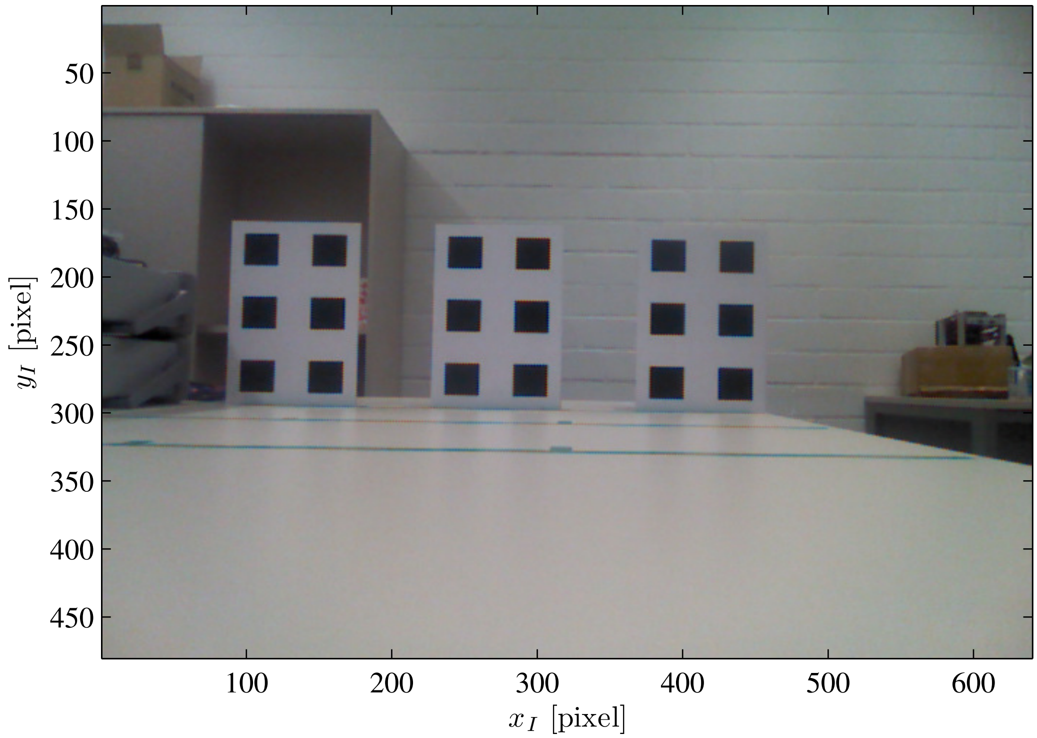


Figure 4.3: RGB calibration image recorded by Microsoft Kinect

the less influence measurement errors will have on the estimated parameter vector $\hat{\vec{b}}$. In figure 4.4 the dimensions of the calibration plates, which were used for the calibration are shown.

4.4 Depth camera calibration

The calibration of the depth camera basically is the same as the calibration of the RGB camera with one difference. For the depth camera three output parameters and not only two as for the RGB calibration can be measured. These three outputs are the two pixel position coordinates x_I and y_I as well as the depth information obtained by the depth camera. This depth information will be denoted by d in the following.

To include the depth information d into the calibration scheme it has to be figured out first what the value d describes at all. In our case it is known that d is any positive value, which is quantized by 12 bit and represents the distance of a point $O\left(x_W, y_W, z_W\right)$ to the depth camera, which is represented by the image pixel $I\left(x_I, y_I\right)$. The Kinect data sheet also states that the value d is linearly quantized. For the depth camera calibration first two different schemes were considered to figure out which of the calibration schemes describes the recorded depth image in a better way. Those two approaches are described in the following two sections.

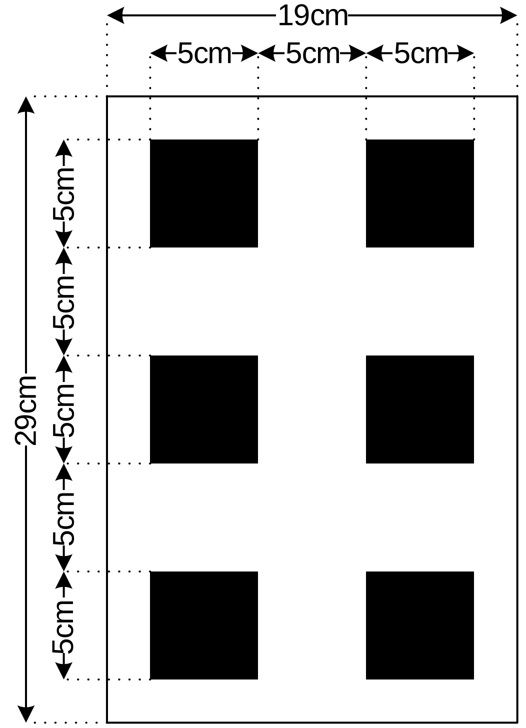


Figure 4.4: Calibration plate for camera calibration

4.4.1 Depth calibration approaches 1

In the first approach it was considered that the depth information d is a dimension for the length of a vector $\vec{d^*}$ from the optical center of the camera to the recorded 3D point $O\left(x_W, y_W, z_W\right)$. The norm of this vector d^* ($d^* = \|\vec{d^*}\|$) is assumed to be a linear function of d as given in equation 4.19. Figure 4.5 shows the interpretation of d for the approach described here. In this figure the coordinates x_I^* and y_I^* are defined as given in equations 4.20 and 4.21

$$d^* = \|\vec{d^*}\| = m \cdot (d - d_0) \tag{4.19}$$

$$x_I^* = x_I - x_{I0} \tag{4.20}$$

$$y_I^* = y_I - y_{I0} \tag{4.21}$$

In figure 4.5 the relationship between image and camera coordinates is shown. As already mentioned, the transformation from camera coordinates to world coordinates is an additional affine transformation,

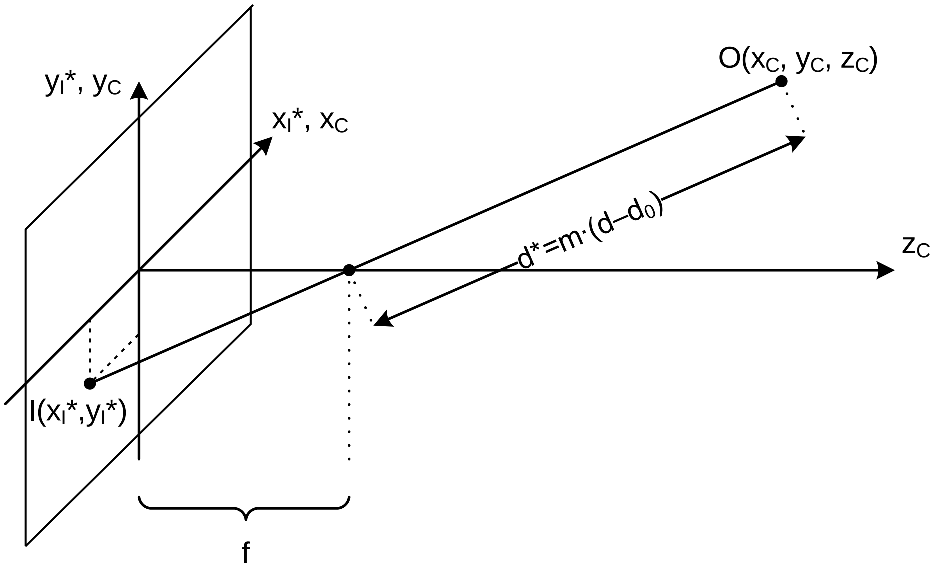


Figure 4.5: Depth camera calibration approach 1

which can be included afterwards.

To include the depth information d into the LS estimating scheme, a function between d and the missing z component of the image z_I has to be defined. Therefore the vector $\vec{d^*}$ is described in camera coordinates as given in equation 4.22.

$$\begin{aligned}\vec{d^*} &= \frac{m \cdot (d - d_0)}{\sqrt{\left(x_I^*\right)^2 + \left(y_I^*\right)^2 + f^2}} \cdot \begin{pmatrix} x_I^* \\ y_I^* \\ f \end{pmatrix} \\ &= \frac{m \cdot (d - d_0)}{\sqrt{\left(x_I - x_{I0}\right)^2 + \left(y_I - y_{I0}\right)^2 + f^2}} \cdot \begin{pmatrix} x_I - x_{I0} \\ y_I - y_{I0} \\ f \end{pmatrix} \end{aligned} \tag{4.22}$$

The vector describing the point O in camera coordinates $\vec{o}_C$ is the addition of the depth vector of the point O, $\vec{d}^{*(O)}$ and the unit vector $\vec{e}_{Cz}$ times the focal length f, as given in equation 4.23. Thus, the z component in camera coordinates z_C can be described as a function of the depth information d and the image coordinates x_I and y_I as given in equation 4.24.

$$\vec{o}_C = \begin{pmatrix} x_C^{(O)} \\ y_C^{(O)} \\ z_C^{(O)} \end{pmatrix} = f \cdot \vec{e}_{Cz} + \vec{d}^{*(O)} \tag{4.23}$$

$$\begin{aligned} z_C &= \frac{m \cdot (d - d_0)}{\sqrt{\left(x_I^*\right)^2 + \left(y_I^*\right)^2 + f^2}} \cdot f + f \\ &= \frac{m \cdot (d - d_0)}{\sqrt{\left(x_I - x_{I0}\right)^2 + \left(y_I - y_{I0}\right)^2 + f^2}} \cdot f + f \end{aligned} \tag{4.24}$$

Since z_C is now defined as a function of the variables x_I, y_I, and d, also z_I is defined because since $z_I = k \cdot z_C$ holds true. Therefore the transformation scheme is defined as given in equation 4.25. This equation describes the linear relation between the world coordinates x_W, y_W, and z_W and the coordinates x_I, y_I, and z_C. Since also a z component is defined, which can be obtained out of measurements, the whole transformation Matrix $\boldsymbol{A}_{Depth}$ can be estimated.

$$\begin{pmatrix} k \cdot x_I \\ k \cdot y_I \\ k \cdot z_I \\ k \end{pmatrix} = \begin{pmatrix} k \cdot x_I \\ k \cdot y_I \\ z_C \\ k \end{pmatrix} = \boldsymbol{A}_{Depth} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} = \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{pmatrix} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} \tag{4.25}$$

In the next step the whole system of equations is devided by b_{44}, as for the RGB calibration, except for the third row. Thus the equation given in 4.26 is received.

$$\begin{pmatrix} k^* \cdot x_I \\ k^* \cdot y_I \\ z_C \\ k^* \end{pmatrix} = \begin{pmatrix} k/b_{44} \cdot x_I \\ k/b_{44} \cdot y_I \\ z_C \\ k/b_{44} \end{pmatrix} = \begin{pmatrix} b_{11}/b_{44} & b_{12}/b_{44} & b_{13}/b_{44} & b_{14}/b_{44} \\ b_{21}/b_{44} & b_{22}/b_{44} & b_{23}/b_{44} & b_{24}/b_{44} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41}/b_{44} & b_{42}/b_{44} & b_{43}/b_{44} & 1 \end{pmatrix} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} \tag{4.26}$$

Out of equation 4.26 again as for the RGB camera calibration (equations 4.11 and 4.12) two equations describing the image coordinates x_I and y_I can be determined as given in equations 4.27 and 4.28. Out of those two equations the parameters of the first two rows and the last row of $\boldsymbol{A}_{Depth}^*$ can be estimated.

$$x_I = b_{11}^* \cdot x_W + b_{12}^* \cdot y_W + b_{13}^* \cdot z_W + b_{14}^* - b_{41}^* \cdot x_W \cdot x_I - b_{42}^* \cdot y_W \cdot x_I - b_{43}^* \cdot z_W \cdot x_I \tag{4.27}$$

$$y_I = b_{21}^* \cdot x_W + b_{22}^* \cdot y_W + b_{23}^* \cdot z_W + b_{24}^* - b_{41}^* \cdot x_W \cdot y_I - b_{42}^* \cdot y_W \cdot y_I - b_{43}^* \cdot z_W \cdot y_I \tag{4.28}$$

Additionally a third equation is needed to describe the third row of the system of equations. This equation is obtained by inserting the definition of z_C given in equation 4.24 into equation 4.26. Thus equation 4.29 is received.

$$\begin{aligned} z_C &= \frac{m \cdot (d - d_0)}{\sqrt{(x_I - x_{I0})^2 + (y_I - y_{I0})^2 + f^2}} \cdot f + f \\ &= b_{31} \cdot x_W + b_{32} \cdot y_W + b_{33} \cdot z_W + b_{34} \end{aligned} \tag{4.29}$$

Form equation 4.29 f is subtracted and the whole equation is divided by $m \cdot f$ and thus results in equation 4.30

$$\begin{aligned} z_C^* = \frac{z_C - f}{m\dot{f}} &= \frac{d - d_0}{\sqrt{(x_I - x_{I0})^2 + (y_I - y_{I0})^2 + f^2}} \\ &= \frac{b_{31}}{m \cdot f} \cdot x_W + \frac{b_{32}}{m \cdot f} \cdot y_W + \frac{b_{33}}{m \cdot f} \cdot z_W + \frac{b_{34} - f}{m \cdot f} \\ &= b_{31}^* \cdot x_W + b_{32}^* \cdot y_W + b_{33}^* \cdot z_W + b_{34}^* \end{aligned} \tag{4.30}$$

Since we are not interested in z_C itself, the third line of the system of equations given in equation 4.26 can be replaced by equation 4.30 and results in equation 4.31.

$$\begin{pmatrix} k^* \cdot x_I \\ k^* \cdot y_I \\ z_C^* \\ k^* \end{pmatrix} = \boldsymbol{A}_{Depth}^* \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} = \begin{pmatrix} b_{11}^* & b_{12}^* & b_{13}^* & b_{14}^* \\ b_{21}^* & b_{22}^* & b_{23}^* & b_{24}^* \\ b_{31}^* & b_{32}^* & b_{33}^* & b_{34}^* \\ b_{41}^* & b_{42}^* & b_{43}^* & 1 \end{pmatrix} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} \tag{4.31}$$

z_C^* is still depentent on the depth information d, the image coordinates x_I and y_I and the constant but unknown values d_0, f, x_{I0}, and y_{I0}. For solving this estimation problem it is needed to get an equation describing one output variable based on a linear combination of a certain number of known input variables out of equation 4.30. To solve this problem two different ways will be described.

4.4.1.1 Known focal length f and image focal point (x_{I0}, y_{I0})

The first and easier way is to assume the parameters x_{I0}, y_{I0}, and f as given and therefore beside the matrix coefficients b_{31}^* to b_{34}^* only the parameter d_0 has to be estimated. The parameters x_{I0}, y_{I0}, and f for example could be estimated prior by the calibration method of Tsai (12) or Zhang (13). In the case of the Kinect cameras f is already given and x_{I0} and y_{I0} usually can be assumed to be in the image center. Thus the equation 4.32 is received where d is a measurable output and u_1, u_2, u_3, and u_4 are variable but defined inputs.

$$\begin{aligned} d &= b_{31}^{*} \cdot x_W \cdot \sqrt{\left(x_I - x_{I0}\right)^2 + \left(y_I - y_{I0}\right)^2 + f^2} + b_{32}^{*} \cdot y_W \cdot \sqrt{\left(x_I - x_{I0}\right)^2 + \left(y_I - y_{I0}\right)^2 + f^2} \\ &\quad + b_{33}^{*} \cdot z_W \cdot \sqrt{\left(x_I - x_{I0}\right)^2 + \left(y_I - y_{I0}\right)^2 + f^2} + b_{34}^{*} \cdot \sqrt{\left(x_I - x_{I0}\right)^2 + \left(y_I - y_{I0}\right)^2 + f^2} + d_0 \\ &= b_{31}^{*} \cdot u_1 + b_{32}^{*} \cdot u_2 + b_{33}^{*} \cdot u_3 + b_{34}^{*} \cdot u_4 + d_0 \end{aligned} \tag{4.32}$$

The inputs u_1 to u_4 are defined as given in equation 4.33 to 4.36.

$$u_1 = x_W \cdot \sqrt{\left(x_I - x_{I0}\right)^2 + \left(y_I - y_{I0}\right)^2 + f^2} \tag{4.33}$$

$$u_2 = y_W \cdot \sqrt{\left(x_I - x_{I0}\right)^2 + \left(y_I - y_{I0}\right)^2 + f^2} \tag{4.34}$$

$$u_3 = z_W \cdot \sqrt{\left(x_I - x_{I0}\right)^2 + \left(y_I - y_{I0}\right)^2 + f^2} \tag{4.35}$$

$$u_4 = \sqrt{\left(x_I - x_{I0}\right)^2 + \left(y_I - y_{I0}\right)^2 + f^2} \tag{4.36}$$

Out of the equations 4.28, 4.29 and 4.32 a system of linear equations is received with 16 unknown coefficients, which have to be estimated by a LS estimation.

4.4.1.2 Unknown focal length f and image focal point (x_{I0}, y_{I0})

If the parameters f, x_{I0}, and y_{I0} are not known prior, they have to be estimated, too. But therefore an linear relation between input and output has to be obtained. One way would be to solve the function by $d - d_0$ and build the square of the whole function. This would eliminate the square root but would also result in a huge number of coefficients to be estimated. This coefficients result from products of combinations of the unknown parameters and thus is rather impractical. Another way would be to approximate the nonlinear square root function by a linear function. This can be done by a Taylor approximation. But since from this approximation an error results, this error will be added to the measurement error. This approximation error is not a stochastic but a systematic error. Thus the overall error can not be assumed as uncorrelated and Gaussian anymore.

Nevertheless, since the parameters f, x_{I0}, and y_{I0} where already known before the method described in section 4.4.1.1 was applied. But also for the case that the intrinsic parameters are not known before a prior estimation of those parameters is probably more efficient than the method described in this section.

4.4.2 Depth calibration approaches 2

For the second approach it was assumed that the depth information d is a dimension for the distance to the image plane and thus for the z component of the camera coordinate system z_C. Therefore the z

component in camera coordinates z_C can be described as a function of the depth information d as given in equation 4.37.

$$z_C = m \cdot d - d_0 \tag{4.37}$$

By inserting equation 4.37 into the transformation equation the equation given in equation 4.38 is received.

$$\begin{aligned}\begin{pmatrix} k \cdot x_I \\ k \cdot y_I \\ k \cdot z_I \\ k \end{pmatrix} = \begin{pmatrix} k \cdot x_I \\ k \cdot y_I \\ z_C \\ k \end{pmatrix} = \begin{pmatrix} k \cdot x_I \\ k \cdot y_I \\ m \cdot d - d_0 \\ k \end{pmatrix} &= \boldsymbol{A}_{Depth} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{pmatrix} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix}\end{aligned} \tag{4.38}$$

As already done before the whole system of equations given in equation 4.38 will be divided by b_{44} except the third row. To the third row d_0 will be added and it will be devided by m. This will result in the equation given in 4.39.

$$\begin{aligned}\begin{pmatrix} k^* \cdot x_I \\ k^* \cdot y_I \\ d \\ k^* \end{pmatrix} = \begin{pmatrix} {}^{k}\!/_{b_{44}} \cdot x_I \\ {}^{k}\!/_{b_{44}} \cdot y_I \\ {}^{(z_C+d_0)}\!/_{m} \\ {}^{k}\!/_{b_{44}} \end{pmatrix} &= \begin{pmatrix} {}^{b_{11}}\!/_{b_{44}} & {}^{b_{12}}\!/_{b_{44}} & {}^{b_{13}}\!/_{b_{44}} & {}^{b_{14}}\!/_{b_{44}} \\ {}^{b_{21}}\!/_{b_{44}} & {}^{b_{22}}\!/_{b_{44}} & {}^{b_{23}}\!/_{b_{44}} & {}^{b_{24}}\!/_{b_{44}} \\ {}^{b_{31}}\!/_{m} & {}^{b_{32}}\!/_{m} & {}^{b_{33}}\!/_{m} & {}^{b_{34}+d_0}\!/_{m} \\ {}^{b_{41}}\!/_{b_{44}} & {}^{b_{42}}\!/_{b_{44}} & {}^{b_{43}}\!/_{b_{44}} & 1 \end{pmatrix} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} \\ = \boldsymbol{A}^*_{Depth} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} &= \begin{pmatrix} b^*_{11} & b^*_{12} & b^*_{13} & b^*_{14} \\ b^*_{21} & b^*_{22} & b^*_{23} & b^*_{24} \\ b^*_{31} & b^*_{32} & b^*_{33} & b^*_{34} \\ b^*_{41} & b^*_{42} & b^*_{43} & 1 \end{pmatrix} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix}\end{aligned} \tag{4.39}$$

Out of equation 4.39 the same definitions for the image coordinates x_I and y_I are retrieved as for the RGB camera calibration. Beside that and additional equation for the depth information d is obtained, which is just the third row of the equation given in 4.39. All three functions are given in equations 4.40 to 4.42.

$$x_I = b^*_{11} \cdot x_W + b^*_{12} \cdot y_W + b^*_{13} \cdot z_W + b^*_{14} - b^*_{41} \cdot x_W \cdot x_I - b^*_{42} \cdot y_W \cdot x_I - b^*_{43} \cdot z_W \cdot x_I \tag{4.40}$$

$$y_I = b^*_{21} \cdot x_W + b^*_{22} \cdot y_W + b^*_{23} \cdot z_W + b^*_{24} - b^*_{41} \cdot x_W \cdot y_I - b^*_{42} \cdot y_W \cdot y_I - b^*_{43} \cdot z_W \cdot y_I \tag{4.41}$$

$$d = b^*_{31} \cdot x_W + b^*_{32} \cdot y_W + b^*_{33} \cdot z_W + b^*_{34} \tag{4.42}$$

Out of these three equations again an input matrix $\boldsymbol{X}$ (given in equation 4.43) and an output vector $\vec{y}$ (given in 4.44) as described in section 4.3 have to be defined.

$$\boldsymbol{X} = \begin{pmatrix} x_W^{(1)} & y_W^{(1)} & z_W^{(1)} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -x_W^{(1)} x_I^{(1)} & -y_W^{(1)} x_I^{(1)} & -z_W^{(1)} x_I^{(1)} \\ 0 & 0 & 0 & 0 & x_W^{(1)} & y_W^{(1)} & z_W^{(1)} & 1 & 0 & 0 & 0 & 0 & -x_W^{(1)} y_I^{(1)} & -y_W^{(1)} y_I^{(1)} & -z_W^{(1)} y_I^{(1)} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_W^{(1)} & y_W^{(1)} & z_W^{(1)} & 1 & 0 & 0 & 0 \\ x_W^{(2)} & y_W^{(2)} & z_W^{(2)} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -x_W^{(2)} x_I^{(2)} & -y_W^{(2)} x_I^{(2)} & -z_W^{(2)} x_I^{(2)} \\ 0 & 0 & 0 & 0 & x_W^{(2)} & y_W^{(2)} & z_W^{(2)} & 1 & 0 & 0 & 0 & 0 & -x_W^{(2)} x_I^{(2)} & -y_W^{(2)} x_I^{(2)} & -z_W^{(2)} x_I^{(2)} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_W^{(2)} & y_W^{(2)} & z_W^{(2)} & 1 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_W^{(N)} & y_W^{(N)} & z_W^{(N)} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -x_W^{(N)} x_I^{(N)} & -y_W^{(N)} x_I^{(N)} & -z_W^{(N)} x_I^{(N)} \\ 0 & 0 & 0 & 0 & x_W^{(N)} & y_W^{(N)} & z_W^{(N)} & 1 & 0 & 0 & 0 & 0 & -x_W^{(N)} y_I^{(N)} & -y_W^{(N)} y_I^{(N)} & -z_W^{(N)} y_I^{(N)} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_W^{(N)} & y_W^{(N)} & z_W^{(N)} & 1 & 0 & 0 & 0 \end{pmatrix} \tag{4.43}$$

For the input matrix $\boldsymbol{X}$ and the output vector $\vec{y}$ the corresponding vector of coefficients, which has to be estimated, $\vec{b}$ is given in equation 4.45.

$$\vec{y}^T = \begin{pmatrix} x_I^{(1)} & y_I^{(1)} & d^{(1)} & x_I^{(2)} & y_I^{(2)} & d^{(2)} & \cdots & x_I^{(N)} & y_I^{(N)} & d^{(N)} \end{pmatrix}^T \tag{4.44}$$

$$\vec{b}^T = \begin{pmatrix} b^*_{11} & b^*_{12} & b^*_{13} & b^*_{14} & b^*_{21} & b^*_{22} & b^*_{23} & b^*_{24} & b^*_{31} & b^*_{32} & b^*_{33} & b^*_{34} & b^*_{41} & b^*_{42} & b^*_{43} \end{pmatrix}^T \tag{4.45}$$

The linear system of equations, which has to be estimated here consists of three equations with overall 15 unknown coefficients and thus at least five measurement points have to be taken to solve this system. Out of those three equations all coefficients of the transformation matrix $\boldsymbol{A}^*_{Depth}$ can be calculated. Since this matrix will be invertible, also the world coordinates x_W, y_W, and z_W can be calculated out of the recorded image coordinates x_I and y_I and depth information d. This transformation is given in equation 4.46.

$$\begin{pmatrix} x_W/k^* \\ y_W/k^* \\ z_W/k^* \\ 1/k^* \end{pmatrix} = \left(A^*_{Depth}\right)^{-1} \cdot \begin{pmatrix} x_I \\ y_I \\ d/k* \\ 1 \end{pmatrix} = \begin{pmatrix} b'_{11} & b'_{12} & b'_{13} & b'_{14} \\ b'_{21} & b'_{22} & b'_{23} & b'_{24} \\ b'_{31} & b'_{32} & b'_{33} & b'_{34} \\ b'_{41} & b'_{42} & b'_{43} & b'_{44} \end{pmatrix} \cdot \begin{pmatrix} x_I \\ y_I \\ d/k* \\ 1 \end{pmatrix} \tag{4.46}$$

To perform this transformation k^* has to be calculates first out of the fourth row as given in equation 4.47 to get d/k^*.

$$k^* = \frac{1 - b'_{43} \cdot d}{b'_{41} \cdot x_I + b'_{42} \cdot y_I + b'_{44}} \tag{4.47}$$

The depth image calibration was performed with both described approaches. Hereby for the first approach the focal length f given in the Kinect's data sheet was used. The image focal point was assumed to be in the center of the depth camera image. First measurements showed that the first approach does not fit the reality at all. Planes parallel to the image plane were projected onto spherical surfaces, which showed that the assumed model fits not that of the depth image camera. Thus the accuracy of the depth image calibration was good for image points close to the image center but highly decayed for decentralized image points.

The second approach showed good accuracy all over the image. Only with increasing distance of recorded points to the Kinect sensor the accuracy decreased. But this rather comes from the Kinect itself, which shows a decreasing accuracy for high depth values, than from the calibration. Nevertheless, with this calibration approach a sufficient accuracy can be reached. After the calibration the position and size of an object can measured with an accuracy of about ±5 mm in a distance of about 1 m and with an accuracy of about ±25 mm in a distance of about 3 m.

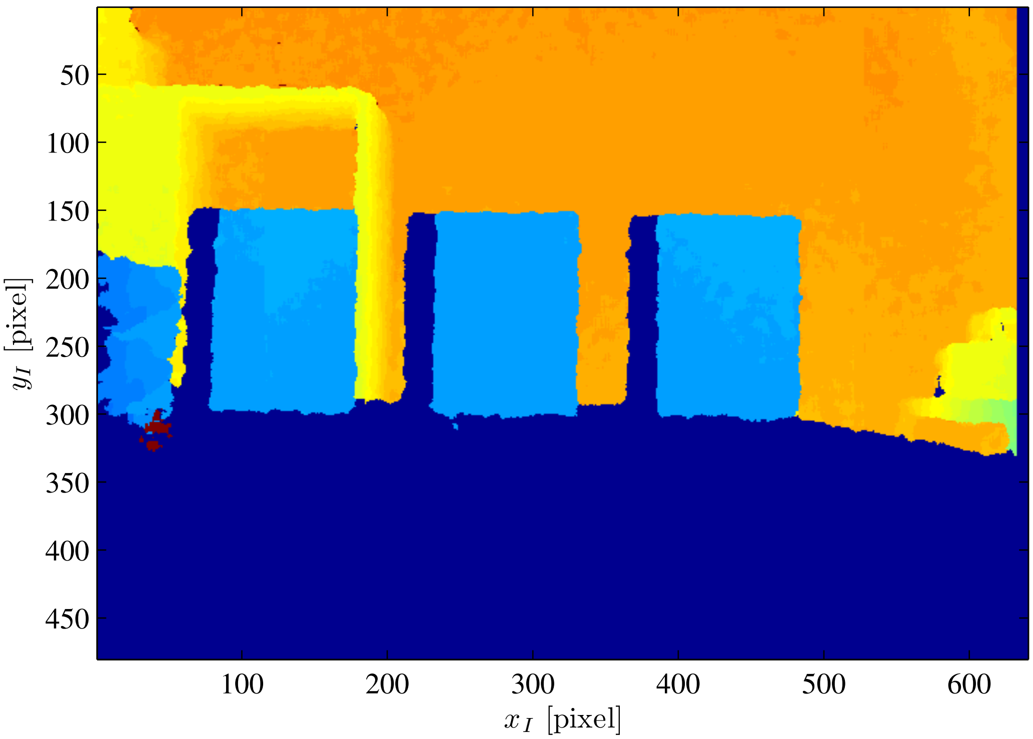


Figure 4.6: Depth calibration image recorded by Microsoft Kinect

Since the depth image camera does not record colors, the corners of black squares on the calibration plates can not be used as measurement points. Thus the corners of the calibration plates are used as calibration points since their position in world coordinates is also known. So, from each plate four calibration points can be retrieved. Since the edges of the plates recorded by the camera are very noise (as shown in figure 4.6), it is important to use a large number of calibration points for the parameter estimation. It is also important to take calibration points with many different depth values d to get an distinct set of data.

Chapter 5

Development of algorithms

This chapter describes the development of the image processing algorithms. The algorithms are supposed to model a recorded scene by 3D geometric objects. These objects then shall be outputted as obstacles for the blind user by some other sense than vision. Until now there is not really an output system developed and thus for first experiments the output is done textual.

The calibration described in chapter 4 *Camera calibration* is needed to project the recorded image back into world coordinates. This calibration has to be done prior using the Kinect cameras as navigation device. Since the intrinsic parameters of the Kinect do not change and the extrinsic parameters are defined relatively to the Kinect's position and thus are constant, the calibration has to be performed only once. Nevertheless, changing the image resolution of the Kinect cameras affects the image coordinates and thus also the transformation matrix. To avoid this for the development of the algorithms the image resolution was kept constant. Both, the RGB as well as the depth camera are working with an image resolution of 640 × 480. Since the calibration was performed prior the image processing itself, for the algorithms described here both transformation matrices $\boldsymbol{A}^*_{RGB}$ and $\boldsymbol{A}^*_{Depth}$ are considered to be already defined. $\boldsymbol{A}^*_{RGB}$ defines the transformation from world coordinates (x_W, y_W, and z_W) into RGB image coordinates ($x_I^{(RGB)}$ and $y_I^{(RGB)}$) as given in equation 5.1. Respectively, $\boldsymbol{A}^*_{Depth}$ defines the transformation from world coordinates (x_W, y_W, and z_W) into depth image coordinates ($x_I^{(Depth)}$ and $y_I^{(Depth)}$) as given in equation 5.2. In both equations the matrix coefficients are notated in the same way as given in chapter 4 *Camera calibration*. For the RGB image transformation the third row given in equation 4.5 is ignored since its coefficients are undefined.

$$\begin{pmatrix} k^* \cdot x_I^{(RGB)} \\ k^* \cdot x_I^{(RGB)} \\ k^* \end{pmatrix} = \boldsymbol{A}^*_{RGB} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} = \begin{pmatrix} a^*_{11} & a^*_{12} & a^*_{13} & a^*_{14} \\ a^*_{21} & a^*_{22} & a^*_{23} & a^*_{24} \\ a^*_{41} & a^*_{42} & a^*_{43} & a^*_{44} \end{pmatrix} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} \tag{5.1}$$

$$\begin{pmatrix} k^* \cdot x_I^{(Depth)} \\ k^* \cdot x_I^{(Depth)} \\ d \\ k^* \end{pmatrix} = \boldsymbol{A}^*_{Depth} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} = \begin{pmatrix} b^*_{11} & b^*_{12} & b^*_{13} & b^*_{14} \\ b^*_{21} & b^*_{22} & b^*_{23} & b^*_{24} \\ b^*_{31} & b^*_{32} & b^*_{33} & b^*_{34} \\ b^*_{41} & b^*_{42} & b^*_{43} & b^*_{44} \end{pmatrix} \cdot \begin{pmatrix} x_W \\ y_W \\ z_W \\ 1 \end{pmatrix} \tag{5.2}$$

Basically, the idea behind the image processing is to divide it into three parts. Since the Kinect supplies color as well as depth information, one step would be to perform a color image based segmentation and another to perform a depth image based segmentation. Then either the results of both segmentation shall be merged to gain a reliable segmentation result. Another approach would be that already the segmentation is performed on a fusion of color and depth data. The third step after the segmentation of color and depth image would be to model obstacles out of the received segments, which can be outputted.

The depth image received by Kinect often has undefined areas (without depth information). This lack of depth information can separate one big object into several small ones. In this case the color information can be helpful to figure out if small, neighbored segments are actually one big, coherent segment or not. At the current state no color based segmentation algorithm is implemented in the system. Nevertheless, some approaches were analyzed. One of these algorithms is called *Graph-based image segmentation* and was developed by Felzenszwalb and Huttenlocher (5). This algorithm is considered to be suitable for the purpose described here. Thus this algorithm will be described very briefly in the next section to give some knowledge about the idea behind it. All algorithms developed until now are based on image processing of single images. At the current state of development there is no processing of image sequences.

5.1 Color image segmentation

The color image segmentation approach (*Graph-based image segmentation*) presented here (5) is considered to be suitable for the above described purpose since it is efficient to implement. Besides, the algorithm can be applied on full RGB data and not only on gray scale values. Another advantage is that the algorithm not only considers local criteria for segmentation but also global ones. Thus it is able to combine areas with fine structured color pattern or areas with a color gradient to one segment. The algorithm results in a segmentation where each pixel within the image is assigned to a segment based on its RGB value and the RGB values of the other pixels within the segment. This gives the opportunity to use the segments to support the depth image based plane segmentation by transforming it into 3D world coordinates.

As the name of the algorithm already says is the segmentation performed based on a graph. In this graph all image pixels are assigned as nodes. At the beginning of the algorithm all possible edges of the graph are calculated. From each node P_i a weighted edge e_{ij} to its eight next neighbors P_j is defined. For each edge a weight $w(P_i, P_j) = w(e_{ij})$ is calculated. The weight $w(P_i, P_j)$ is calculated as given in equation 5.3 where $R(P)$, $G(P)$, and $B(P)$ are the RGB information of a pixel P and P_i and P_j are

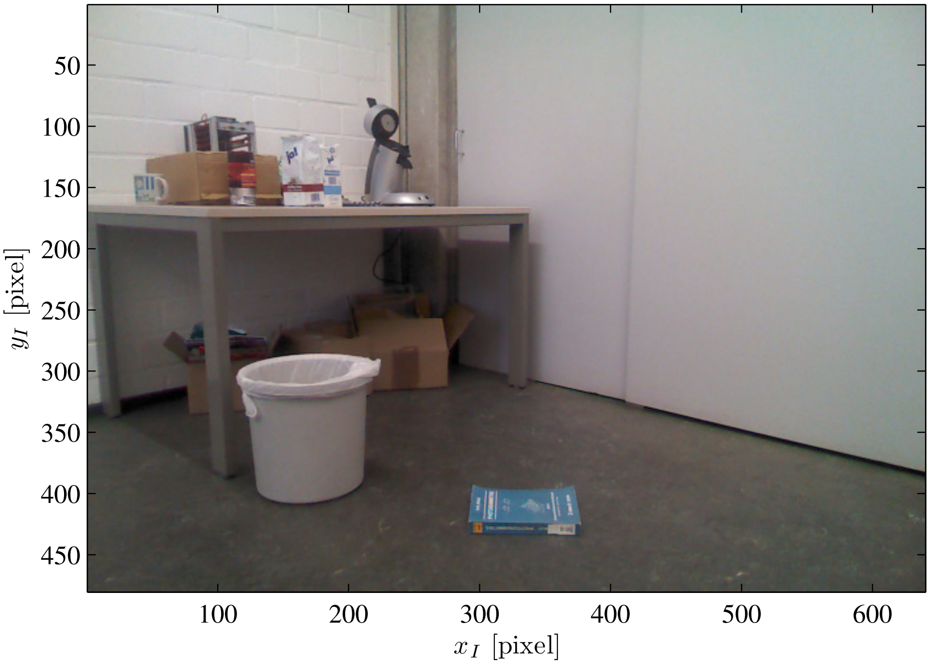


Figure 5.1: Color image of recorded scene on which the *Graph-based image segmentation* is applied

neighbored pixels.

$$w(P_i, P_j) = \left\| \begin{pmatrix} R\left(P_i\right) \\ G\left(P_i\right) \\ B\left(P_i\right) \end{pmatrix} - \begin{pmatrix} R\left(P_j\right) \\ G\left(P_j\right) \\ B\left(P_j\right) \end{pmatrix} \right\| \tag{5.3}$$

Based on the calculated weights $w(P_i, P_j)$ of the edge e_{ij} between two pixels (P_i and P_j) segments are built. At the beginning each single pixel P is considered to be one segment C. For each segment C the so called *initial difference* $Int(C)$ is calculated, which is defined as the maximum weight $w(e)$ within the segment C. Two neighbored segments C_i and C_j are considered to be different segments if the definition $D(C_i, C_j)$ as given in equation 5.4 is true.

$$D(C_i, C_j) = \begin{cases} \text{true} & \text{if } Dif(C_i, C_j) > MInt(C_i, C_j) \\ \text{false} & \text{else.} \end{cases} \tag{5.4}$$

In equation 5.4 $Dif(C_i, C_j)$ is defined as the minimum weight of all edges between the two segments C_i and C_j as given in equation 5.5.

$$Dif(C_i, C_j) = \min_{P_k \in C_i, P_l \in C_j} w(P_k, P_l) \tag{5.5}$$

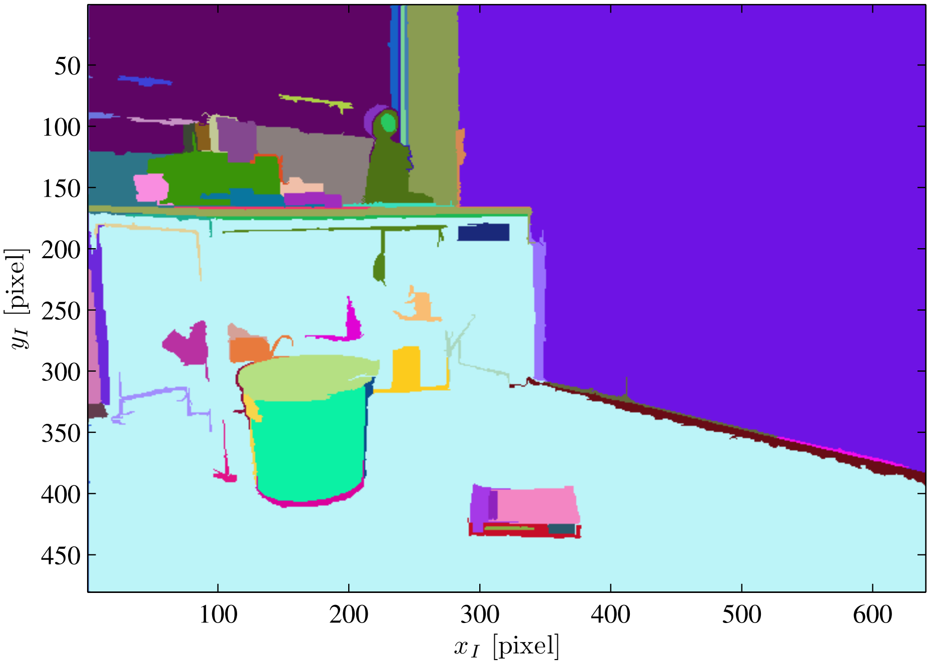


Figure 5.2: Result after *Graph-based image segmentation* with $k = 500$ and $\sigma = 0.5$

$MInt(C_i, C_j)$ is called the *minimum internal difference*, which is defined as given in equation 5.6. $\tau(C)$ is a threshold function which is dependent on some constant k and the segment size $|C|$.

$$MInt(C_i, C_j) = \min(Int(C_i) + \tau(C_i), Int(C_j) + \tau(C_j)) \tag{5.6}$$

$$\tau(C) = \frac{k}{|C|} \tag{5.7}$$

The segmentation algorithm is applied to the image shown in figure 5.1. Figure 5.2 shows the result of the corresponding segmentation algorithm. In the image shown in figure 5.2 each segment is represented by a different color. For the segmentation $k = 500$ was chosen. Besides, the image was filtered by a 2D Gaussian FIR[1] filter with standard deviation $\sigma = 0.5$ prior the segmentation. Additionally, a minimum segment size of 100 pixel was defined and thus all pixels of smaller segments are assigned to neighbored segments. In (5) more information about the algorithm can be found.

---

[1] finite impulse response

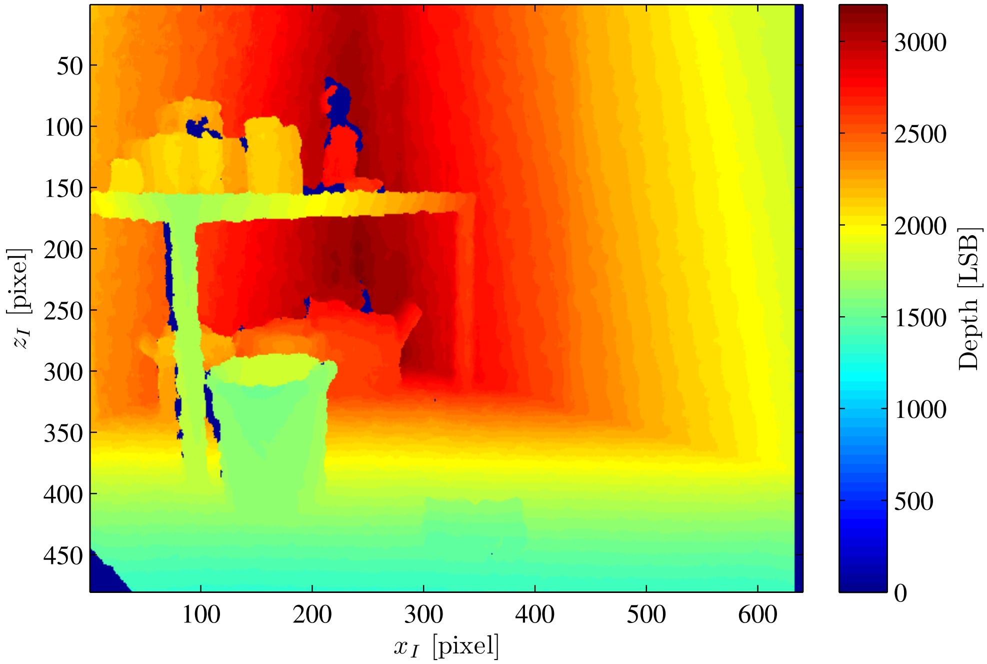


Figure 5.3: Depth image recorded by Microsoft Kinect

5.2 Depth image processing

The idea of the depth image processing is to project the depth image into a 3D point cloud and divide this point cloud into several planes. Afterwards the received planes shall be combined to objects, which describe the recorded scenery as good as possible but also only with very primitive shapes, which easily can be represented by text or later by audio signals. To receive a 3D point cloud out of the recorded image the corresponding position in world coordinates has to be calculated for each pixel in the depth image. To do so, the matrix given in equation 5.2 has to be solved to the world coordinates vector $(x_W \quad y_W \quad z_W \quad 1)^T$ as shown in equations 4.46 and 4.47. Figure 5.3 shows an example of a recorded depth image. This image is transformed into world coordinates and thus results in the 3D point cloud shown in figure 5.4. In this point cloud each pixel is defined by 3D coordinates, which are defined in centimeters. For calculating and visualizing this point cloud only the pixels of each seventh row and column are transformed. Thus the point cloud shown in figure 5.4 has a point density which is by the factor 49 lower than the one of the real depth image. Since the depth image is recorded by a resolution of 632 x 480pixel (because the eight most right columns do not carry depth information), the point could resulting from this image will consist out of up to 303 360 points irrespective of some additional points, which do not carry depth information.

As already mentioned consists the depth image segmentation out of two major steps. The first one is the plane segmentation, which also includes the calculation of the 3D point cloud. After this

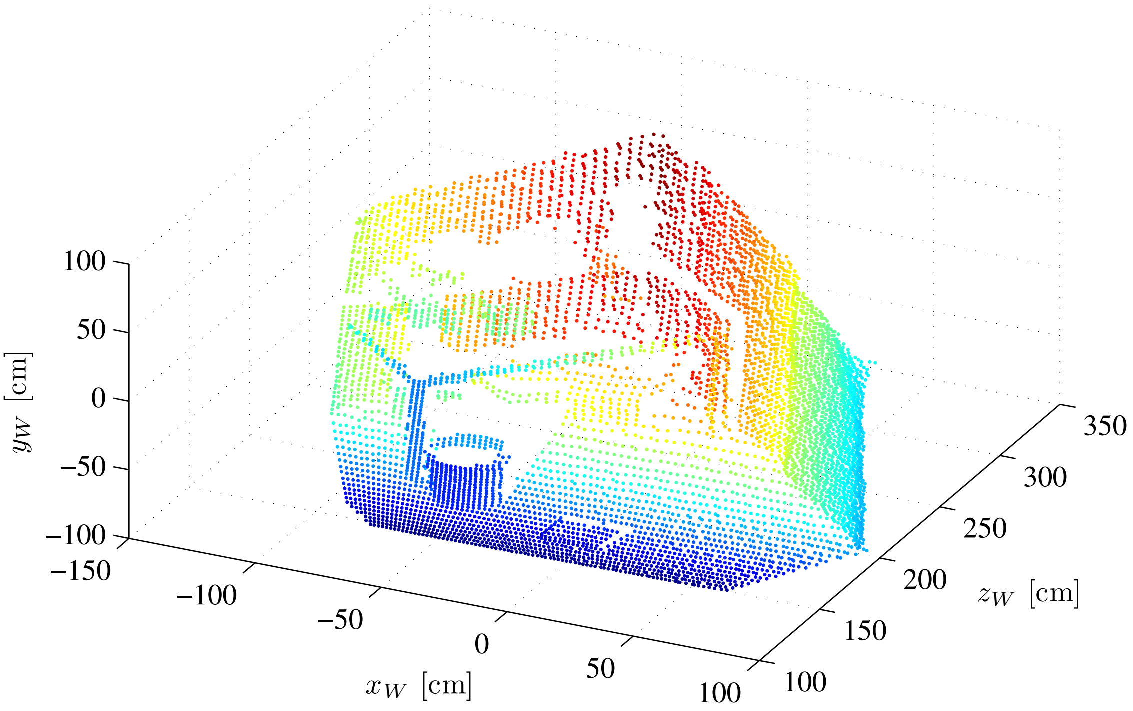


Figure 5.4: Point cloud of depth image transformed into world coordinates (down sampled by $R = 7$)

segmentation the planes have to be combined to objects, which is the second major step. This step includes finding planes, which correspond to the same object and representing this object as a 3D solid figure.

5.2.1 Plane segmentation

There already exist various different methods for plane segmentation in point clouds. During the development of the image processing algorithm different approaches were implemented and tested. In this section two different plane segmentation approaches are presented and their pros and cons are analyzed. The first approach is a normal vector based plane fitting algorithm, which is very often used for 3D point cloud segmentation. The second approach is the one which was actually chosen for depth image processing in this thesis. This approach is a combination of the very popular RANSAC[2] algorithm (6) and a gradient based segmentation approach.

5.2.1.1 Normal based plane segmentation

The first plane segmentation approach, which will be discuss in this thesis is called the normal based plane fitting approach since the plane segments are built based on the normal vectors of neighbored

---

[2] random sample consensus

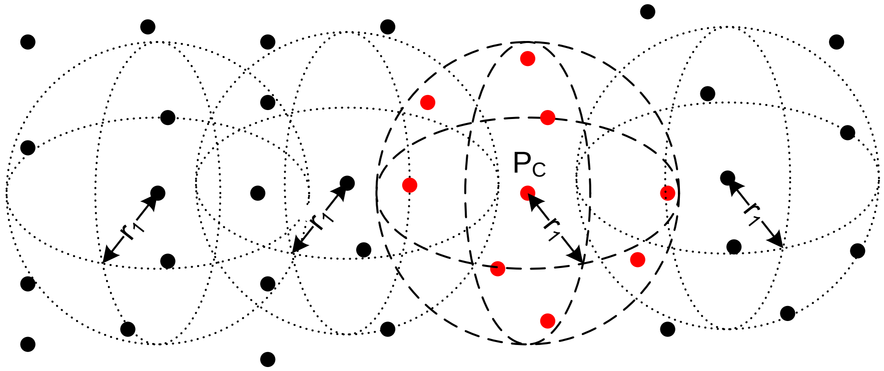
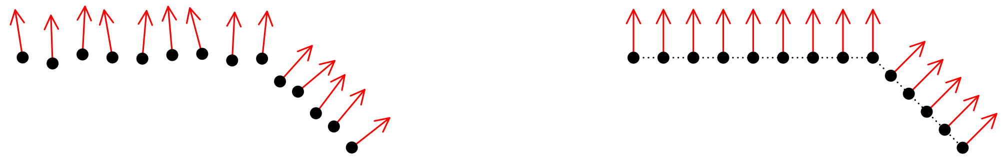


Figure 5.5: Combining points to estimate the tangential normal vector in point p_C (all red point are used to estimate the normal vector)

plane elements. For this approach each point in the point cloud will be defined by its position in 3D world coordinates and additionally by a normal vector defining the orientation of the tangential plane in this point. As already described in section 2.1.3 *Plane* a plane can be totally defined out of three points. Thus for each point in the point cloud its tangential normal vector can be estimated out of the point itself and its two nearest neighbor points. The resulting normal vector calculated out of three neighbored point will not be really accurate since the recorded depth image and thus the point cloud is noisy and includes quantization and interpolation errors. Beside that the number of data in the point cloud will be very high since for each image pixel a 3D position vector as well as a 3D normal vector is received. Therefore the normal vector is not calculated for each single point in the point cloud but for a group of points within a radius r_1 around a center point P_C. By combining a number of points the noise is reduced and so the amount of data. But also the spacial resolution is reduced by combining data points. Thus for r_1 a suitable value has to be chosen to keep a high enough spacial resolution. For first experiments a radius of $r_1 = 5\,\text{mm}$ was chosen. The data points are overlapping combined such that one point can be considered for more than one normal vector estimation. Figure 5.5 shows how the points of the point cloud are combined to calculated the normal vectors for the center points. All red marked points in the figure are used to estimate the normal vector for the point P_C since they lie within the dashed sphere of radius r_1. The other three dotted spheres show the radius around the neighbored center points. Any point lying outside the radius r_1 can be considered as the next center point for which a normal vector is estimated. The point of the point cloud are combined to groups of points until all points lie in at least one sphere around a center point. For estimating the normal vector $\vec{n}$ of the tangential plane the LS estimation method as described in section 2.4.1 *LS estimator* is used. For the LS estimation the component c of the plane function given in equation 2.4 has to be set to -1 and thus the function can be written as given in equation 5.8. In this equation z is the output signal



(a) Normals of tangential planes before merging neighbored points

(b) Normals of tangential planes after merging neighbored points

Figure 5.6: Combining neighbored points based on the normal of their tangential plan (cross section view)

and $(x, y, 1)$ the input vector for the linear estimation.

$$z = a \cdot x + b \cdot y + d \tag{5.8}$$

After the normal vector $\vec{n}$ is estimated for each of the center points P_c neighbored center points will be compared to each other based on their normal vector. It is checked if the angle $\Delta\phi^{(ij)}$ between the normal vectors $\vec{n}^{(i)}$ and $\vec{n}^{(j)}$ of two neighbored center points $P_c^{(i)}$ and $P_c^{(j)}$ is within a threshold T_ϕ. Center points within a radius r_2 (with $r_2 > r_1$) are considered to be neighbors and will be compared to each other. If the angle $\Delta\phi$ is underneath the threshold T_ϕ, the two groups of points are merged together to one group. All points within this group are considered to be of the same plane. This has to be done for the whole point cloud and thus results in a plane segmentation. Figure 5.6 shows a cross section of how the single points (or groups of point) (figure 5.6(a)) are merged to planes based on the angle between the normal vectors. Figure 5.6(b) shows a cross section of the points (or groups of point) projected onto the resulting plane segments with their corrected normal vectors. Each point in the figures represents a center point P_c. For all points, which are assigned to the same plane the normal vector of the corresponding tangential planes will be equivalent.

The angle $\Delta\phi^{(ij)}$ between two normal vectors $\vec{n}^{(i)}$ and $\vec{n}^{(j)}$ can be calculated based on the scalar product between those vectors as given in equation 5.9.

$$\Delta\phi^{(ij)} = \arccos\left(\frac{\left(\vec{n}^{(i)}\right)^T \cdot \vec{n}^{(j)}}{\|\vec{n}^{(i)}\| \cdot \|\vec{n}^{(j)}\|}\right) \tag{5.9}$$

With the described normal based plane segmentation very accurate and reliable results can be reached. Nevertheless, the algorithm is also very time consuming. The problem here is that for each point its neighbor points have to be searched in a 3D space. If the point cloud is not sorted in any way this means each point has to be compared with all other points which results in a complexity of $O(n^2)$ only for searching the neighbored points where n is the number of points in the point cloud. After the points are merged to small groups out of which the normal vector of the tangential plane can be estimated again the neighbor groups have to be found. Data structures like an *Octree* can be used to reduce the searching effort but this tree has to be calculated first out of the point cloud. In the case described here it is advantageous that the point cloud comes from a 2D depth image. Thus neighbored points in the 3D point cloud also can be considered to be neighbored pixels in the 2D image and so searching neighbored point within an radius r_1 in the 3D space is replaced by search neighbored pixels in the 2D depth image. This approach reduces the computation time of searching neighbored points. But since in the presented algorithm very often new plane normal vectors have to be calculated, it is still very time consuming. The best segmentation result is reached if immediately after merging two groups of points their new normal vector is estimated. If the normal vectors are all estimated at the very end after all groups of points where compared, there exists a risk that points are combined to one plane, which are not of the same plane at all. This is the case for example for curved surfaces where neighbored points have almost the same normal vector. Nevertheless, a curved surface is not one single plane at all. Therefore a compromise has to be found between how often the planes are re-estimated and how accurate the segmentation shall be.

5.2.1.2 Gradient based plane segmentation

The idea of the second approach was to do as much processing as possible in the 2D depth image since here most of the operations (e.g. searching routines) are much more efficient than in the 3D space. In this algorithm first a rough gradient based segmentation within the 2D image is performed. Afterwards the RANSAC algorithm (6) is applied to the prior built segments. The advantage of combining these two different segmentation methods is on one hand to apply the 3D RANSAC algorithm to smaller point clouds and thus reduce the computation effort, on the other hand to benefit from the different advantages of both methods. While the RANSAC algorithm can handle large numbers of outliers within large planes very well, is the gradient based image segmentation very sensitive to small but sharp edges, which result in a large gradient magnitudes. The segmentation method described here is divided into several steps as listed below:

1. Dividing depth image into clusters
2. Gradient based segmentation of single clusters
3. RANSAC based plane segmentation

In the following paragraph all these steps will be described in detail.

Dividing depth image into clusters

The first step which is done for the plane segmentation is to calculate the gradient of the depth image. Since the recorded depth image is a function of two variables ($d = f(x_I, y_I)$), the gradient of d, $\vec{\nabla} d$, results in a 2D vector, which is dependent on the variables x_I and y_I as given in equation 5.10.

$$\vec{\nabla} d(x_I, y_I) = \frac{\partial d(x_I, y_I)}{\partial x_I} \cdot \vec{e}_{xI} + \frac{\partial d(x_I, y_I)}{\partial y_I} \cdot \vec{e}_{yI} = \begin{pmatrix} \frac{\partial d(x_I, y_I)}{\partial x_I} \\ \frac{\partial d(x_I, y_I)}{\partial y_I} \end{pmatrix} \tag{5.10}$$

The gradient vector $\vec{\nabla} d\left(x_I, y_I\right)$ at the position $\left(x_I, y_I\right)$ points into the direction of the steepest slope at this position. Beside that the norm of this vector gives a dimension for the steepness of this slope.

Since the variables x_I and y_I are not continuous but discrete, the function $d = f(x_I, y_I)$ will not be differentiable and thus the differential has to be led to a difference. There exist different ways to calculate a difference representing the depth image gradient $\vec{\nabla} d$. The easiest one is just to calculate the difference between two neighbored values in x- and y-direction as given in equation 5.11.

$$\begin{aligned}\vec{g}_d^{(1)}\left(x_I, y_I\right) &= {}^1\!/_{\Delta x} \cdot \left[d\left(x_I, y_I\right) - d\left(x_I - \Delta x, y_I\right)\right] \cdot \vec{e}_{xI} + {}^1\!/_{\Delta y} \cdot \left[d\left(x_I, y_I\right) - d\left(x_I, y_I - \Delta y\right)\right] \cdot \vec{e}_{yI} \\ &= \begin{pmatrix} {}^1\!/_{\Delta x} \cdot \left[d\left(x_I, y_I\right) - d\left(x_I - \Delta x, y_I\right)\right] \\ {}^1\!/_{\Delta y} \cdot \left[d\left(x_I, y_I\right) - d\left(x_I, y_I - \Delta y\right)\right] \end{pmatrix} \approx \vec{\nabla} d\left(x_I, y_I\right) \qquad (5.11)\end{aligned}$$

For the images, which are considered here the x- and y-coordinate have got the dimension pixel. Thus the distances between two neighbored values Δx respectively Δy are equivalent to one pixel ($\Delta x = \Delta y = 1$). A problem of calculating the gradient this way is that the result of $\vec{g}_d^{(1)}(x_I, y_I)$ at the position (x_I, y_I) does not really present the gradient vector $\vec{\nabla} d\left(x_I, y_I\right)$ but the gradient vector $\vec{\nabla} d\left(x_I - 0.5\Delta x, y_I - 0.5\Delta y\right)$. There exist different approaches to calculate the gradient vector out of the discrete image data which consider this problem. One method is shown in equation 5.12 where the gradient at the position (x_I, y_I) is calculated out of its next four neighbor pixels.

$$\begin{aligned}\vec{g}_d^{(2)}\left(x_I, y_I\right) &= 0.5 \cdot \left[d\left(x_I + 1, y_I\right) - d\left(x_I - 1, y_I\right)\right] \cdot \vec{e}_{xI} + 0.5 \cdot \left[d\left(x_I, y_I + 1\right) - d\left(x_I, y_I - 1\right)\right] \cdot \vec{e}_{yI} \\ &= 0.5 \cdot \begin{pmatrix} d\left(x_I + 1, y_I\right) - d\left(x_I - 1, y_I\right) \\ d\left(x_I, y_I + 1\right) - d\left(x_I, y_I - 1\right) \end{pmatrix} \approx \vec{\nabla} d\left(x_I, y_I\right) \qquad (5.12)\end{aligned}$$

Since in equation 5.12 the difference is calculated over a distance of two pixels, this difference has to be scaled by 0.5 to get a result which is comparable to $\vec{\nabla} d$ ($\Delta x = \Delta y = 2$). From equations 5.11 and 5.12 it can be seen that the two gradients $\vec{g}_d^{(1)}\left(x_I, y_I\right)$ and $\vec{g}_d^{(2)}\left(x_I, y_I\right)$ are defined by a non recursive difference equation and thus also can be described by 2D FIR filters. Thus the calculation of a discrete gradient $\vec{g}_d\left(x_I, y_I\right)$ can be described by two filters, one calculating the x- and one the y-component of of the gradient. Equation 5.13 gives the filter impulses response matrix $H_x^{(1)}$ for the x-component of the gradient $\vec{g}_d^{(1)}\left(x_I, y_I\right)$ and equation 5.14 $H_y^{(1)}$ respectively for the y-component.

$$H_x^{(1)} = \begin{pmatrix} h_x^{(1)}\left(-1, 0\right) & h_x^{(1)}\left(0, 0\right) \end{pmatrix} = \begin{pmatrix} -1 & 1 \end{pmatrix} \qquad (5.13)$$

$$H_y^{(1)} = \begin{pmatrix} h_y^{(1)}\left(0, -1\right) \\ h_y^{(1)}\left(0, 0\right) \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \end{pmatrix} \qquad (5.14)$$

For the calculation method described in equation 5.12 the impulse responses $H_x^{(2)}$ and $H_y^{(2)}$ are received as given in equations 5.15 and 5.16.

$$H_x^{(2)} = \begin{pmatrix} h_x^{(2)}(-1,0) & h_x^{(2)}(0,0) & h_x^{(2)}(1,0) \end{pmatrix} = \begin{pmatrix} -0.5 & 0 & 0.5 \end{pmatrix} \tag{5.15}$$

$$H_y^{(2)} = \begin{pmatrix} h_y^{(2)}(0,-1) \\ h_y^{(2)}(0,0) \\ h_y^{(2)}(0,1) \end{pmatrix} = \begin{pmatrix} -0.5 \\ 0 \\ 0.5 \end{pmatrix} \tag{5.16}$$

With the given impulse responses the gradient over the depth image is received by calculating the 2D convolution of the depth image and the corresponding filter impulse response. Equation 5.17 gives the definition for a convolution of a 2D input signal $I_x(x,y)$ with a 2D impulse response $h(x,y)$.

$$I_y(x,y) = \sum_{n=-\inf}^{\inf} \sum_{m=-\inf}^{\inf} I_x(n,m) \cdot h(x-n, y-m) \tag{5.17}$$

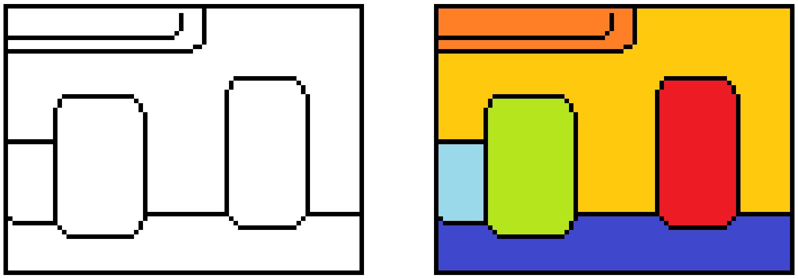
There exist further methods which combine the gradient calculation with any kind of low pass filtering. Those methods are for example the *Prewitt* operator, the *Sobel* operator, or the *Canny* filter. In this approaches the filter impulse response matrix is not only a $1 \times m$ or $n \times 1$ but a $n \times m$ dimensional matrix. This filters can handle noisy signal very well and can be useful for special application, e.g. like edge detection. In the algorithm presented here the gradient filter, which was described first (equations 5.13 and 5.14) is used. To handle the noise within the signal the calculated gradient is filtered with a median filter before the segmentation.

Before the actual gradient based segmentation is performed is the depth image divided into smaller clusters. This clusters are defined based on steps in the depth image. Since the derivative of a step is a Dirac impulse, the norm of the calculated gradient vector $\vec{g}_d$ for this steps will be very large. To detect these steps the norm of the gradient vector $\vec{g}(x_I, y_I)$, $\|\vec{g}(x,y)\|$, is calculated and based on some threshold T_{step} it is decided whether at the current pixel $p(x_I, y_I)$ is a step or not as given in equation 5.18.

$$s(x_I, y_I) = \begin{cases} 1 & \text{if} \quad \|\vec{g}_d(x_I, y_I)\| \geq T_{\text{step}} \\ 0 & \text{else.} \end{cases} \tag{5.18}$$

Thus $s(x_I, y_I)$ gives an image in which all pixels are 0 except those corresponding to steps, which will be 1. The threshold T_{step} is defined experimentally.

Out of $s(x_I, y_I)$ the clusters are built in such way that all pixels are assigned to the same cluster, which are not separated by steps. Figure 5.7 shows how out of the image of steps $s(x_I, y_I)$ (5.7(a)) the separated clusters (5.7(b)) results. In the image shown in figure 5.7(b) the single clusters are represented by different color. The clustering basically results in a cluster function $c(x_I, y_I)$ where to each pixel $p(x_I, y_I)$ the number i (for $i \in \{1, 2, \ldots, N_{CL}\}$) of the cluster it belongs to is assigned. N_{CL} represents the total number of clusters. Equation 5.19 gives a mathematical definition of the cluster function



(a) Steps image $s(x_I, y_I)$

(b) Cluster image $c(x_I, y_I)$ with clusters coded in different colors

Figure 5.7: Building cluster image $c(x_I, y_I)$ out of steps image $s(x_I, y_I)$

$c(x_I, y_I)$. In equation 5.19 C_i is the set of image points included in the i-th cluster. All image pixels representing steps or pixels with no depth information are set to zero.

$$c(x_I, y_I) = \begin{cases} i & \text{if} \quad p(x_I, y_I) \in C_i \quad \text{for} \quad i \in \{1, 2, \cdots, N_{CL}\} \\ 0 & \text{else.} \end{cases} \tag{5.19}$$

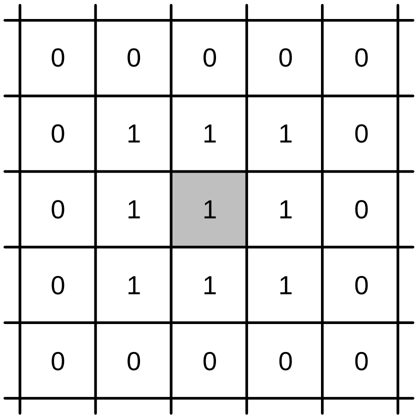
In order to assign all points to the corresponding cluster sets C_i a cluster growth algorithm is used. In this algorithm starting from a seed pixel, which is a pixel that is not classified as step all four direct neighbor pixels are checked whether they are either classified as steps, not assigned to a cluster (unprocessed), or already assigned to a cluster (processed). If a neighbor pixel is unprocessed it is assigned to the same cluster as the seed pixel. Of the neighbor pixels again all neighbors are checked whether they are steps, processed or unprocessed and based on that assigned to the current cluster. This is performed until no unprocessed neighbor can be found anymore. When there is no unprocessed neighbor left a new seed pixel, which is unprocessed and not classified as step has to be selected. From this pixel the cluster growth algorithm continues to build the next cluster. This procedure is continued until there is no unprocessed pixel left. Algorithm 1 describes the clustering process.

Gradient based segmentation

After the image is separated into single clusters to each of the clusters a gradient based segmentation algorithm is applied. Therefore the image gradient is filtered by a median filter first to get rid of the noise. A median filter is a non linear filter, which calculates based on some filter mask the median out of all pixels within this mask. This median value is assigned to that pixel at which the mask is adjusted. For the purpose described here a quadratic filter mask is used as given in figure 5.8. For the shown mask the filter output value for the gray marked pixel is the median of all pixel for which the filter mask has the value one. Since in the case presented here only quadratic masks will be considered and the (gray) pixel at which the mask is adjusted is always in the center of the mask, the filter will be just defined by the dimension N_{MED} (for which counts $N_{MED} \in \{1, 3, 5, \ldots\}$). The value N_{MED} gives the size of the filter mask which is $N_{MED} \times N_{MED}$ pixels. In this thesis the filtered gradient will be

Algorithm 1 Clustering depth image

```
i = 0
for $x_n = 1 \rightarrow N_x$ do                                        ▷ $N_x \rightarrow$ number of pixels in $x$-direction
    for $y_n = 1 \rightarrow N_y$ do                                    ▷ $N_y \rightarrow$ number of pixels in $y$-direction
        if $p\left(x_n, y_n\right)$ is unprocessed $\wedge$ $s\left(x_n, y_n\right) \neq 1$ then    ▷ $p\left(x_n, y_n\right) \rightarrow$ pixel at $x_I = x_n$, $y_I = y_n$
            $i = i + 1$
            $C_i \leftarrow \{\}$
            $C_i^{new} \leftarrow \{p\left(x_n, y_n\right)\}$
            $c(x_n, y_n) = i$
            for $\forall p\left(x_m, y_m\right) \in C_i^{new}$ do
                $C_i \leftarrow \{C_i, p\left(x_m, y_m\right)\}$
                $C_i^{new} \leftarrow C_i^{new} \backslash p\left(x_m, y_m\right)$
                if $p\left(x_m + 1, y_m\right)$ is unprocessed $\wedge$ $s\left(x_m + 1, y_m\right) \neq 1$ then
                    $C_i^{new} \leftarrow \{C_i^{new}, p\left(x_m + 1, y_m\right)\}$
                    $c(x_m + 1, y_m) = i$
                    mark $p\left(x_m + 1, y_m\right)$ as processed
                end if
                if $p\left(x_m, y_m + 1\right)$ is unprocessed $\wedge$ $s\left(x_m, y_m + 1\right) \neq 1$ then
                    $C_i^{new} \leftarrow \{C_i^{new}, p\left(x_m, y_m + 1\right)\}$
                    $c(x_m, y_m + 1) = i$
                    mark $p\left(x_m, y_m + 1\right)$ as processed
                end if
                if Pixel $p\left(x_m - 1, y_m\right)$ is unprocessed $\wedge$ $s\left(x_m - 1, y_m\right) \neq 1$ then
                    $C_i^{new} \leftarrow \{C_i^{new}, p\left(x_m - 1, y_m\right)\}$
                    $c(x_m - 1, y_m) = i$
                    mark $p\left(x_m - 1, y_m\right)$ as processed
                end if
                if Pixel $p\left(x_m, y_m - 1\right)$ is unprocessed $\wedge$ $s\left(x_m, y_m - 1\right) \neq 1$ then
                    $C_i^{new} \leftarrow \{C_i^{new}, p\left(x_m, y_m - 1\right)\}$
                    $c(x_m, y_m - 1) = i$
                    mark $p\left(x_m, y_m - 1\right)$ as processed
                end if
            end for
        end if
    end for
end for
return $c\left(x_I, y_I\right)$ and $C_1$ to $C_i$                      ▷ $C_1$ to $C_i$ are all cluster sets built
```



0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

Figure 5.8: Quadratic median filter mask of size 3 × 3

denoted as $\vec{g}_d^{*}(x_I, y_I)$. Since $\vec{g}_d(x_I, y_I)$ is a 2D vector field, the filtering has to be performed for its x- and y-component separately and thus $\vec{g}_d^{*}(x_I, y_I)$ also results in a 2D vector field.

After the filtering the segmentation is performed based on the filtered gradient $\vec{g}_d^{*}(x_I, y_I)$ and the depth information $d(x_I, y_I)$. The segmentation is done for each cluster separately since pixels from two different clusters can also not be from the same segment. For dividing the clusters into segments again each pixel is compared to its neighbors. For each pixel it is checked whether the absolute value of a difference vector $\|\Delta\vec{g}_d\|$ between the gradients of two neighbored pixels ($\vec{g}_d\left(x_I^{(i)}, y_I^{(i)}\right)$, and $\vec{g}_d\left(x_I^{(j)}, y_I^{(j)}\right)$) is underneath a certain threshold T_g. The difference vector $\Delta\vec{g}_d$ is calculated as given in equation 5.20.

$$\Delta\vec{g}_d = \vec{g}_d\left(x_I^{(i)}, y_I^{(i)}\right) - \vec{g}\left(x_I^{(j)}, y_I^{(j)}\right) \tag{5.20}$$

Beside the gradient itself based on the depth value $d\left(x_I^{(i)}, y_I^{(i)}\right)$ and the gradient $\vec{g}_d\left(x_I^{(i)}, y_I^{(i)}\right)$ of one pixel the expected depth value of the neighbor pixel $\hat{d}\left(x_I^{(j)}, y_I^{(j)}\right)$ is calculated and compared to the recorded depth value at the neighbor pixel $d\left(x_I^{(j)}, y_I^{(j)}\right)$. If the difference between both depth values Δd is underneath a certain threshold T_d and $\|\Delta\vec{g}_d\|$ is underneath the threshold T_g, the pixels $p\left(x_I^{(i)}, y_I^{(i)}\right)$ and $p\left(x_I^{(j)}, y_I^{(j)}\right)$ are considered to be elements of the same segment. The value $\hat{d}(x_2, y_2)$ is calculated as given in equation 5.21.

$$\hat{d}\left(x_I^{(j)}, y_I^{(j)}\right) = d\left(x_I^{(i)}, y_I^{(i)}\right) + \vec{g}_d\left(x_I^{(i)}, y_I^{(i)}\right)^T \cdot \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \tag{5.21}$$

Δx and Δy are defined as given in equation 5.22. Since only the four next neighbors of a pixel $p(x_I, y_I)$

are compared with the pixel itself, only Δx or Δy is equal to ±1 and the other component is equal to 0.

$$\begin{pmatrix}\Delta x\\ \Delta y\end{pmatrix} = \begin{pmatrix}x_I^{(j)} - x_I^{(i)}\\ y_I^{(j)} - y_I^{(i)}\end{pmatrix} \tag{5.22}$$

Out of the above given constrains it results that two neighbored pixels $p\left(x_I^{(i)}, y_I^{(i)}\right)$ and $p\left(x_I^{(j)}, y_I^{(j)}\right)$ are only within the same segment S_n when $|\hat{d}\left(x_I^{(j)}, y_I^{(j)}\right) - d\left(x_I^{(j)}, y_I^{(j)}\right)| < T_d$ and $\|\Delta\vec{g}_d\| < T_g$ holds true. This is again stated in equation 5.23 where N_{SEG} is the total number of segments.

$$\begin{aligned}&[|\hat{d}\left(x_I^{(j)}, y_I^{(j)}\right) - d\left(x_I^{(j)}, y_I^{(j)}\right)| < T_d] \wedge [\|\Delta\vec{g}_d\| < T_g] \quad :\\ &[p\left(x_I^{(i)}, y_I^{(i)}\right) \in S_n] \wedge [p\left(x_I^{(j)}, y_I^{(j)}\right) \in S_n] \quad \exists! n \quad n \in \{1, 2, \ldots, N_{SEG}\}\end{aligned} \tag{5.23}$$

This segmentation procedure results in algorithm 2 where S_n $(n \in 1, 2, \ldots, N_{SEG})$ is the set of pixels in the n-th segment and $s_{SEG}\left(x_I, y_I\right)$ is the function of segments as given in equation 5.24.

$$s_{SEG}\left(x_I, y_I\right) = \begin{cases} n & \text{if} \quad p\left(x_I, y_I\right) \in S_n \quad \text{for} \quad n \in \{1, 2, \cdots, N_{SEG}\}\\ 0 & \text{else.}\end{cases} \tag{5.24}$$

The advantage of the gradient based segmentation is that especially very small steps can be detected very well (e.g. a book lying on the floor), which result in a large absolute value of the the gradient vector. Curved surfaces, for example of a cylindrical object, can not be segmented very well since the algorithm compares only direct neighbored pixels. In this case there is not a big difference between two neighbored pixels and thus the whole surface is combined to one plane. To avoid this a second plane segmentation is applied to the segments S_n $(n \in 1, 2, \ldots N_{SEG})$, which are received form the gradient based segmentation approach. The method used therefore is a very popular plane segmentation approach which is called RANSAC algorithm and will be described in the following paragraph.

RANSAC based plane segmentation

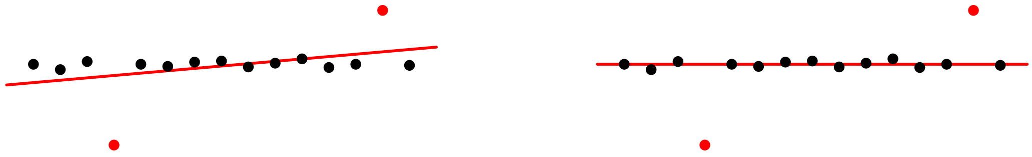
In this paragraph the second plane segmentation step will be described. This segmentation is applied to the set of points S_i of each segment $(i \in \{1, 2, \ldots, N_{SEG}\})$, which results from the gradient based segmentation method. For the second segmentation step the RANSAC algorithm is used. Thus in this paragraph first the RANSAC algorithm itself is presented and afterwards it is described how this algorithm is used for plane segmentation.

RANSAC The RANSAC algorithm was developed by Fischler and Bolles (6) formally as an estimation algorithm for functions out of a set of measurements as it also the LS estimator does. The algorithm basically can be used for any kind of functions but since in this thesis we are only interested in linear functions only those functions will be considered. RANSAC is an acronym for random sample consensus, which basically means that out of a number measurement point randomly a certain number

Algorithm 2 Gradient based depth image segmentation

```
$i = 0$
for $\forall j \in \{1, 2, \ldots, N_{CL}\}$ do
    for $\forall p\left(x_n, y_n\right) \in C_j$ do
        if $p\left(x_n, y_n\right)$ is unprocessed then
            $i = i + 1$
            $S_i \leftarrow \{\}$
            $S_i^{new} \leftarrow \{p\left(x_n, y_n\right)\}$
            $s_{SEG}(x_n, y_n) = i$
            for $\forall p\left(x_m, y_m\right) \in S_i^{new}$ do
                $S_i \leftarrow \{S_i, p\left(x_m, y_m\right)\}$
                $S_i^{new} \leftarrow S_i^{new} \backslash p\left(x_m, y_m\right)$
                for $\forall\left(\Delta x, \Delta y\right) \in \{(1,0), (0,1), (-1,0), (0,-1)\}$ do
                    if $p\left(x_m \Delta x, y_m + \Delta y\right)$ is unprocessed $\wedge$ $p\left(x_m + \Delta x, y_m + \Delta y\right) \in C_j$ then
                        if $\|\vec{g}\left(x_m, y_m\right) - \vec{g}\left(x_m + \Delta x, y_m + \Delta y\right)\| < T_g$ then
                            if $d\left(x_m, y_m\right) + \begin{pmatrix}\Delta x & \Delta y\end{pmatrix} \cdot \vec{g}\left(x_m, y_m\right) - d\left(x_m + \Delta x, y_m + \Delta y\right) < T_d$ then
                                $S_i^{new} \leftarrow \{S_i^{new}, p\left(x_m + \Delta x, y_m + \Delta y\right)\}$
                                $s_{SEG}(x_m + \Delta x, y_m + \Delta y) = i$
                                mark $p\left(x_m + \Delta x, y_m + \Delta y\right)$ as processed
                            end if
                        end if
                    end if
                end for
            end for
        end if
    end for
end for
return $s_{SEG}\left(x_I, y_I\right)$ and $S_1$ to $S_i$        ▷ $S_1$ to $S_i$ are all segment sets built
```

of samples is picked. Based on this random samples the searched function is defined and it is checked how well all other measurement points fit to this function. This procedure is performed until a function is found, which fits very well to all measurement points. In this thesis the RANSAC algorithm, for example, is used to estimate planes out of point clouds. Since a plane can be defined by three points, randomly three points are picked out of the point cloud and the plane, which fits to those three points is calculated. Afterwards the distance from the plane to all other points in the point cloud is calculated and based on some threshold T_{RAN} it is decided if a point is considered to be an inlier or outlier. This procedure is done several times to find a plane with as much inliers as possible. Of course not all combinations of points are checked since this would result in an not acceptable effort. The break condition for the RANSAC algorithm is calculated based on a probability p, which is defined prior the processing. p is the probability therefore that there was in all performed trials at least one trial where all three randomly picked samples where inliers. Therefore after each trial, which results in a new best fitting function (plane) a number of trails N_{trials} is calculated, which is in total needed to reach the probability p. For each of this trials the probability that all selected samples are inliers can be estimated as given in equation 5.25. In this equation the exponent gives the number of picked samples for one trial, which



(a) Estimation of a straight line using LS estimator (b) Estimation of a straight line using RANSAC

Figure 5.9: Estimation of a straight line in the presence of outliers (LS vs. RANSAC)

is three for the plane estimation, $n_{inliers}$ the number of inliers in the current trial and n_{points} the total number of measurement points.

$$P(\text{all 3 samples are inliers}) = \left(\frac{n_{inliers}}{n_{points}}\right)^3 \tag{5.25}$$

Thus the probability that at least one sample is an outlier is received as given in equation 5.26.

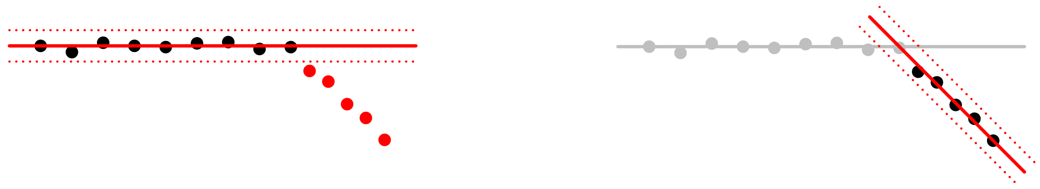
$$P(\text{at least 1 outlier}) = 1 - P(\text{all 3 samples are inliers}) \tag{5.26}$$

For N_{trials} consecutive trials the probability that in each of these trials at least one sample is an outlier can be calculated as given in equation 5.27. Out of equation 5.27 the probability that in all N_{trials} trials will be at least one trial with no outlier as sample is received as given in equation 5.28 and has to be lower than the prior defined probability p. Equation 5.28 can be solved by N_{trial} to get the minimum number of trials needed to ensure the defined probability p.

$$P(\text{at least 1 outlier in each of } N_{trials} \text{ trials}) = P(\text{at least 1 outlier})^{N_{trials}} \tag{5.27}$$

$$\begin{aligned} P(\text{at least 1 trial without outliers}) &= 1 - P(\text{at least 1 outlier in each of } N_{trials} \text{ trials}) \\ &= 1 - P(\text{at least 1 outlier})^{N_{trials}} \geq p \end{aligned} \tag{5.28}$$

One big advantage of this algorithm is that the estimation result is not really effected by large outliers since they are not considered for the estimation. This is visualized in figure 5.9 where a straight line is estimated out of a number of data points with two outliers. Figure 5.9(a) shows the estimation result for a LS estimator and figure 5.9(b) for the RANSAC algorithm. Since in figure 5.9 the function of a straight line wants to be estimated, the RANSAC algorithm randomly picks two points and calculates the straight line crossing these two points. Then the distance from all other points tho the line is calculated and it is checked whether this distance is higher than the threshold T_{RAN}. This procedure is done several times (defined by the break criteria) until the best fitting line (the line with the highest



(a) Building first segment out of data points (points of second segment are considered as outliers)

(b) Second segment is built out of the outliers left after building the first segment

Figure 5.10: Line segmentation using RANSAC algorithm

number of inliers) is found. By comparing both results one can see that the LS estimator is affected by the two outlier other than the RANSAC algorithm, which totally ignores them.

RANSAC for segmentaion In the previous paragraph the RANSAC algorithm is used for plane estimation. Since the algorithm divides a point cloud into inliers and outliers, it can not only be used for plane estimation but also for segmentation. When the algorithm found the best fitting plane (smallest number of outliers) the resulting plane and the points of the point cloud corresponding to this plane are extracted. To the outliers left again the RANSAC algorithm is applied. Figure 5.10 shows the RANSAC based segmentation for a 2D space where out of a number of data points two straight lines are segmented. For the first straight line all data points corresponding to the second line are considered to be outliers and thus are not included to the estimation of the first line (figure 5.10(a)). In the second step all data points assigned to the first line are ignored (gray data points) and the RANSAC algorithm is applied to the left data points as shown in figure 5.10(b). In both figures the red, dotted lines represent the threshold T_{RAN}, which classifies points in outliers and inliers. All points within the line are considered as inliers and thus the goal of the algorithm is to get as much points as possible in between those lines.

In this thesis the RANSAC algorithm is applied to the point cloud of the single segments S_i ($i \in \{1, 2, \ldots, N_{SEG}\}$). Therefore the depth image pixels have to be transformed into world coordinates first as described in section 4.4.2 *Depth calibration approaches 2* and thus resulting in sets of 3D points denoted by $S_i^{(3\text{D})}$ ($i \in \{1, 2, \ldots, N_{SEG}\}$) for the signle segments. The reason why the RANSAC algorithm is applied after the gradient based approach is because this algorithm, other than the gradient based approach, considers global criteria. Thus e.g. curved surfaces, which are still considered to be one plane after the gradient based method are split into a number of planes. All of the resulting planes will have a slightly different orientation. For the plane segmentation the RANSAC algorithm randomly picks three points out of the point cloud $S_i^{(3\text{D})}$ and calculates the corresponding plane function as given in section 2.1.3 *Plane*. To decide for a data point whether it is an inlier or an outlier the distance d of each point $P^{(i)} \in S_i^{(3\text{D})}$ to the current plane has to be calculated. Afterwards it is checked if the distance d is within the given threshold T_{RAN}. The distance d between the data point and the plane can be calculated as given in section 2.2.2 *Minimum distance from a point to a plane in a 3D space*. Algorithm 3 gives a pseudo code representation of the RANSAC based plane segmentation.

Algorithm 3 RANSAC based plane segmentation

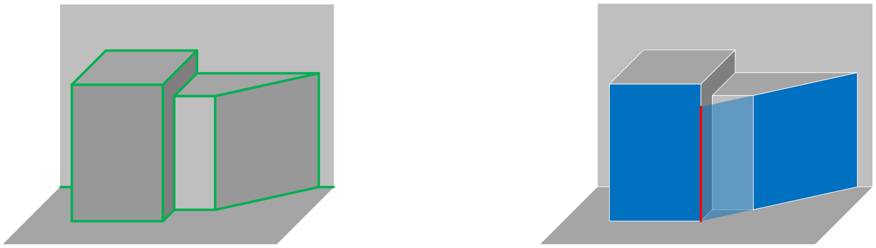
```
$j = 0$
for $\forall S_i^{(3D)}, i \in \{1, 2, \ldots, N_{SEG}\}$ do
    $O \leftarrow S_j^{(3D)}$
    while $|O| >$ min. segment size do
        $O, I \leftarrow$ perform RANSAC on $O$        ▷ $O \rightarrow$ set of outliers, $I \rightarrow$ set of inliers
        $j = j + 1$
        $R_j^{(3D)} \leftarrow I$
    end while
end for
return $R_1^{(3D)}$ to $R_j^{(3D)}$
```

In this algorithm the segments S_i ($i \in \{1, 2, \ldots, N_{SEG}\}$), respectively $S_i^{(3D)}$ are divided into the new segments R_j ($j \in \{1, 2, \ldots, N_{SEG}^{new}\}$), respectively $R_j^{(3D)}$. Thus also a new function of segments $r_{SEG}(x_I, y_I)$ can be defined where to each pixel of the depth image the corresponding segment number is assigned as given in equation 5.29.

$$r_{SEG}(x_I, y_I) = \begin{cases} i & \text{if} \quad p(x_I, y_I) \in R_i \quad \text{for} \quad i \in \{1, 2, \cdots, N_{SEG}^{new}\} \\ 0 & \text{else.} \end{cases} \tag{5.29}$$

5.2.2 Crossing edges between planes

In the following sections the notation S_i ($i \in \{1, 2, \ldots N_{SEG}^{new}$) will be used to define the set of points as well as the estimated plane of the i-th segment. Now that the plane segmentation is performed, the whole depth image is represented by planes except of some outliers which were erased. The next step is to find crossing lines between neighbored planes as described in section 2.2.4 *Crossing line between two planes*. The difficulty here is not to calculated the crossing line between two planes, but to figure out which crossings really exist. A crossing line can be calculated between any to planes, of which the normal vectors are not parallel. But since a plane by definition is not limited to some area this crossing lines can lie anywhere in the 3D space and thus also e.g. beyond the field of view of the depth camera. Figure 5.13 shows two sketches after the plane segmentation. Figure 5.11(a) shows the result of correctly defined crossing edges and figure 5.11(b) of a wrong defined crossing edge between the two blue colored planes. To avoid that wrong crossing edges are calculated a neighbor graph based on the segment function $r_{SEG}(x_I, y_I)$ is built. In this neighbor graph it is stored which segments are direct neighbors in the 2D image plane as defined in equation 5.30 where a neighborhood between a segment i and j is denoted as $i \leftrightarrow j$ (for $i \neq j$ and $i, j \in \{1, 2, \ldots, N_{SEG}^{new}\}$). This means two segments are connected to each other if there exist at least one pixel in each segment set such that those pixels have a distance



(a) Sketch showing correct crossing edges (green)

(b) Sketch showing one wrong crossing edge (red)

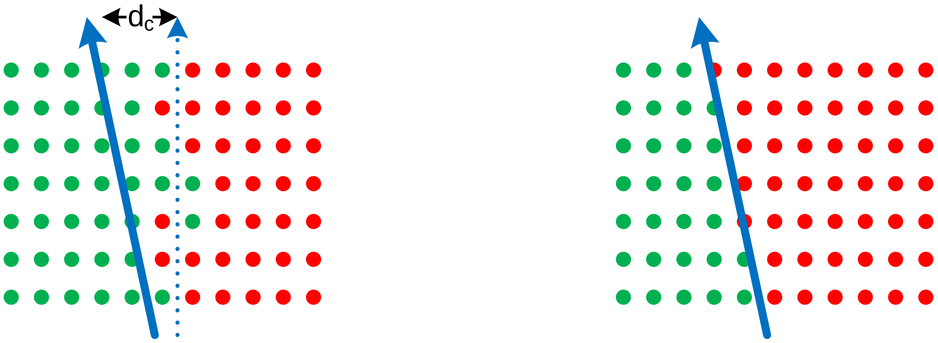
Figure 5.11: Sketches of the result after plane segmentation

smaller than or equal to one to each other.

$$\begin{aligned} &i \leftrightarrow j \qquad \text{if} \qquad |x_I^{(1)} - x_I^{(2)}| + |y_I^{(1)} - y_I^{(2)}| \le 1 \quad \exists p\left(x_I^{(1)}, y_I^{(1)}\right) \in R_i \wedge \exists p\left(x_I^{(2)}, y_I^{(2)}\right) \in R_j \\ &i \not\leftrightarrow j \qquad \text{else.} \end{aligned} \tag{5.30}$$

Beside the existence of a connection between segment i and j also the pixels are stored at which the segments are connected $[p\left(x_I^{(1)}, y_I^{(1)}\right), p\left(x_I^{(2)}, y_I^{(2)}\right)]$ (with $p\left(x_I^{(1)}, y_I^{(1)}\right) \in R_i$ and $p\left(x_I^{(2)}, y_I^{(2)}\right) \in R_j$) in a set P_{ij}. In the following step only segments, which have neighbored pixels in the image plane are taken into consideration of having a shared crossing edge. Out of the 3D points belonging to the set of connecting pixels P_{ij} a straight line $\hat{L}_c^{(ij)}$ is estimated by an LS estimation. This estimated straight line is compared to the crossing edge $L_c^{(ij)}$ between the planes of the segments i and j. Between the straight line $\hat{L}_c^{(ij)}$ and the crossing edge $L_c^{(ij)}$ a distance d_c is calculated. This distance is the maximum distance from the crossing edge $L_c^{(ij)}$ to the straight line $\hat{L}_c^{(ij)}$ within a range defined by the connecting pixels P_{ij}. The range is defined by those two points out of P_{ij}, which have the greatest distance between each other projected onto the straight line $\hat{L}_c^{(ij)}$. If the distance d_c (see figure 5.12(a)) is less than some threshold T_{cross} the crossing edge $L_c^{(ij)}$ is considered to be existent. Beside the straight line function $L_c^{(ij)}$ each crossing edge has a defined direction. Therefore the direction vector $\vec{d}_c^{(ij)}$ points always in a direction such that the segment i is on the left side to it and the segment j is on the right side to it when the crossing line is projected onto the 2D image plane.

After all crossing edges are calculated there will be overlapping pixels (or 3D points) for each segment. This means, points of the segment i which lie on the right side of the crossing edge $L_c^{(ij)}$ in pointing direction and respectively points of the segment j which lie on the left side of crossing edge. Those point will be assigned to the respectively other segment (i or j). The easiest way to do this is to project the crossing edges into the 2D image plane by the transformation matrix A_{Depth}^* and process the overlapping pixels is the 2D plane. Figure 5.12 visualizes these overlapping points in a 2D sketch. After all crossing edges are calculated this results in a 3D model as given in figure 5.11(a) but still without any objective representation. At this point the 3D model is still described only by planes. Nevertheless, the knowledge



(a) estimated (dotted line) and calculated (solid line) crossing edges between two segments i (green) and j (red)

(b) Readjusted segment points based on the calculated crossing edge $L_c^{(ij)}$

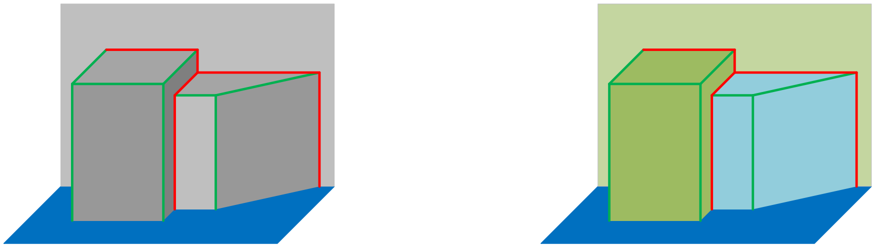
Figure 5.12: Crossing edge between two segments in the 2D image plane

about existing crossing edges between the planes is gained, which will be needed later to combine the planes to objects. This will be described in section 5.2.4 *Object modeling*. Before that in the next section will be described how the floor plane is extracted.

5.2.3 Floor plane extraction

The floor plane has to be extracted out of the 3D representation of the scene since it depicts not an obstacle. The approach presented to detect the floor plane is based on the camera position in the 3D world coordinate system. Since the camera will be mounted to a persons body, its orientation according the floor plane will be more or less constant. Furthermore, also the distance from the camera center C to the floor plane has only a little variance. Therefore, these two properties are used to retrieve the floor plane, which will be denoted as S_F out of the segmented planes in 3D world coordinates. Thus, as parameters for finding the floor plane a predefined floor plane normal vector $\vec{n}_F$, based on the orientation of the camera, as well as the distance d_F from the camera center C to the floor plane S_F are defined. The camera center C does not have to be known really precise an thus can be estimated by trial and error method out of a recorded image. Nevertheless, any other point within the world coordinate system could basically be used to define the floor plane distance. Based on the thresholds T_ϕ and T_{dist} it is decided whether a plane S_i (for $i \in \{1, 2, \dots, N_{SEG}^{new}\}$) is the floor plane S_F or not. The angle $\Delta\phi_i$, between the normal vector $\vec{n}^{(i)}$ of a plane S_i, which is checked to be the floor plane, and the given normal vector $\vec{n}_F$, must be less than T_ϕ. The angle $\Delta\phi$ is calculated as given in equation 5.31.

$$\Delta\phi = \arccos\left(\frac{n_F^T \cdot \vec{n}^{(i)}}{\|n_F^T\| \cdot \|\vec{n}^{(i)}\|}\right) \tag{5.31}$$



(a) Sketch showing segmented planes and their crossing edges (convex crossing edges in green and concave ones in red)

(b) Sketch showing segmented plane combined to objects based on their crossing edges

Figure 5.13: Sketches of object modeling based on crossing edges

Beside that, the difference between the minimum distance $d^{(i)}$ from a plane S_i to the camera center C and the predefined floor plane distacen d_F has to be less than T_{dist}. The distance $d^{(i)}$ can be calculated as given in section 2.2.2 *Minimum distance from a point to a plane in a 3D space*. For the case that more than one plane fits to the floor plane constrains the one which consists out of the most data points is chosen. It also can happen that no plane fits the constrains since the floor plane is outside the field of view. Then no floor plane will be assigned.

For simplicity it is assumed that the floor plane S_F is the lowest plane in the scene and thus it can not be any plane underneath. Therefore all planes which are underneath the floor plane S_F are considered to be outliers and are erased. This of course is not true for all scenarios but for the most. Thus most of the planes laying underneath the floor plane result from segmentation errors. For detecting steps which are going downwards planes underneath the floor plane must be defined. Nevertheless, this scenario was not considered yet but has to be considered in future work.

5.2.4 Object modeling

Now that all planes are defined and also the floor plane is extracted, the planes that are left have to be modeled to objects. Neighbored planes, which have a convex, shared crossing edge will be combined to one object. Figure 5.13(a) shows a sketch of a 3D scene after plane segmentation and crossing edge calculation. In this sketch all convex crossing edges are marked green and all concave crossing edges are marked red. The crossing edges to the floor plane (blue) are not drawn into sketch since this plane is already extracted. Figure 5.13(b) shows the objects built based on the convex crossing edges in different colors.

5.2.4.1 Classifying concave and convex edges

To figure out if a crossing edge is convex or concave all plane normal vectors have to point towards the camera center C. For each plane S_i two different normal vector directions can be defined resulting in the same plane. If the standardized normal vector of a plane S_i is $\vec{n}_0^{(i)}$, the same plane can also be defined with a standardized normal vector $\vec{n}_0^{*(i)}$ pointing in the opposite direction ($\vec{n}_0^{*(i)} = -\vec{n}_0^{(i)}$). Thus for each plane S_i, for which the normal vector points away from the camera center C the normal vector $\vec{n}_0^{(i)}$ has to be replaced by $\vec{n}_0^{*(i)}$. This results in the same as multiplying the plane equation given in equation 5.32 by (-1).

$$\begin{aligned} 0 &= a \cdot x + b \cdot y + c \cdot z + d \qquad\qquad | \cdot (-1) \\ 0 &= -a \cdot x - b \cdot y - c \cdot z - d \\ 0 &= a^* \cdot x + b^* \cdot y + c^* \cdot z + d^* \end{aligned} \tag{5.32}$$

To check if the normal vector $\vec{n}^{(i)}$ of a plane S_i points into the direction of the camera center C a vector $\vec{v}^{(i)}$ pointing from any point $P^{(i)}$ on the plane S_i to the camera center C is calculated as defined in equation 5.33.

$$\vec{v}^{(i)} = \vec{c} - \vec{p}^{(i)} \tag{5.33}$$

Afterwards, the scalar product of the vector $\vec{v}^{(i)}$ and the normal vector $\vec{n}^{(i)}$ is calculated. If this scalar product is larger or equal to zero, the normal vector $\vec{n}^{(i)}$ points into the correct direction. If it is smaller than zeros, the normal vector has to be flipped.

$$\vec{n}^{(i)} = \begin{cases} \vec{n}^{(i)} & \text{if} \quad \left(\vec{v}^{(i)}\right)^T \cdot \vec{n}^{(i)} \geq 0 \\ -\vec{n}^{(i)} & \text{else.} \end{cases} \tag{5.34}$$

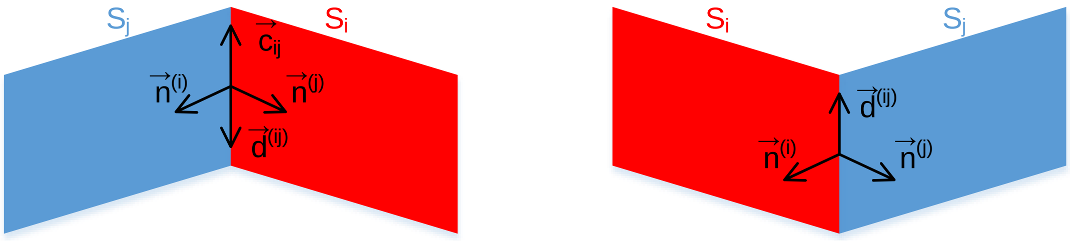
Each crossing edge $L_c^{(ij)}$ between two neighbored planes S_i and S_j has a direction vector $\vec{d}_c^{(ij)}$ with a certain direction. The pointing direction of $\vec{d}_c^{(ij)}$ was defined prior such that the plane S_i is on the left side of the crossing edge $L_c^{(ij)}$ and the plane S_j on the right side of the crossing edge in pointing direction. Thus it can be obtained out of the cross product $\vec{c}_{ij}$ of $\vec{n}^{(i)}$ and $\vec{n}^{(j)}$ if the crossing edge described by $L_c^{(ij)}$ is either convex or concave. For a convex edge the cross product $\vec{c}_{ij}$ (equation 5.36) results in a vector which points in same the direction as $\vec{d}_c^{(ij)}$. For a concave edge the cross product points into the opposite direction as $\vec{d}_c^{(ij)}$.

$$\vec{c}_{ij} = \vec{n}^{(i)} \times \vec{n}^{(j)} \tag{5.35}$$

Thus a crossing edge between S_i and S_j, defined by $L_c^{(ij)}$, is said to be convex based on the definition given in equation 5.36.

$$L_c^{(ij)} \mathrel{\widehat{=}} \begin{cases} \text{convex edge} & \text{if} \quad \vec{c}_{ij}^{\,T} \cdot \vec{d}_c^{(ij)} > 0 \\ \text{concave edge} & \text{else.} \end{cases} \tag{5.36}$$

The orientation of the normal vectors $\vec{n}^{(i)}$ and $\vec{n}^{(j)}$ as well as the resulting cross product $\vec{c}_{ij}$ is visualized



(a) Concave crossing edge between the planes S_i and S_j

(b) Convex crossing edge between the planes S_i and S_j

Figure 5.14: Concave crossing edge compared to convex crossing edge

in figure 5.14 for a concave (fig. 5.14(a)) as well as for a convex edge (fig. 5.14(b)). Algorithm 4 again states the procedure how the crossing edges $L_c^{(ij)}$ are classified into concave and convex edges. Here the set of convex edges is denoted by E_{conv} and respectively the set of concave edges by E_{conc}.

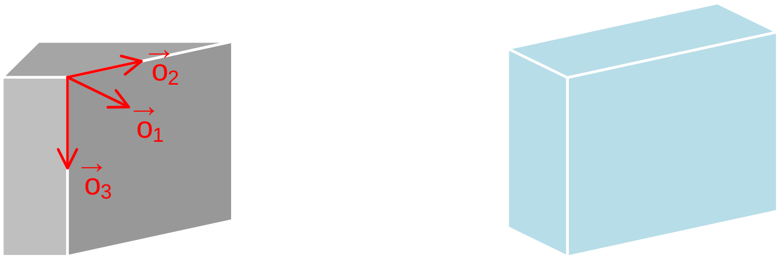
Algorithm 4 Classification of crossing edges

```
$E_{conv} \leftarrow \{\}$                                        ▷ set of convex edges
$E_{conc} \leftarrow \{\}$                                        ▷ set of concave edges
for $\forall S_i, \quad i \in \{1, 2, \dots, N_{SEG}^{new}\}$ do        ▷ Lets point all plan normals towards the camera center $\vec{c}$
    $\vec{v}^{(i)} = \vec{c} - \vec{p}^{(i)}$                          ▷ $\vec{p}^{(i)}$ is any point on the plane $S_i$
    if $\left(\vec{v}^{(i)}\right)^T \cdot \vec{n}^{(i)} < 0$ then
        $\vec{n}^{(i)} = -\vec{n}^{(i)}$
    end if
end for
for $\forall L_c^{(ij)}, \quad i, j \in \{1, 2, \dots, N_{SEG}^{new}\}$ do
    if $\left(\vec{d}_c^{(ij)}\right)^T \cdot \left(\vec{n}^{(i)} \times \vec{n}^{(j)}\right) > 0$ then
        $E_{conv} \leftarrow \{L_c^{(ij)}, E_{conv}\}$               ▷ $L_c^{(ij)}$ is assigned to the set of convex edges
    else
        $E_{conc} \leftarrow \{L_c^{(ij)}, E_{conc}\}$               ▷ $L_c^{(ij)}$ is assigned to the set of concave edges
    end if
end for
```

5.2.4.2 Building objects

After all crossing edges $L_c^{(ij)}$ are classified to be either concave ($L_c^{(ij)} \in E_{conc}$) or convex ($L_c^{(ij)} \in E_{conv}$) the objects can be build. As already mentioned, all planes, which are connected by a convex crossing edge are assigned to the same object O_k as shown in figure 5.13(b). Segments, which do not have a convex edge at all to any other plane as the bright green back plane in figure 5.13(b) build their own object.

After each plane S_i (for $i \in \{1, 2, \dots, N_{SEG}^{new}\}$) is assigned to an object O_k (for $k \in \{1, 2, \dots N_{OBJ}\}$)



(a) Plane assigned to one object before calculating bounding box

(b) Bounding box representing an object

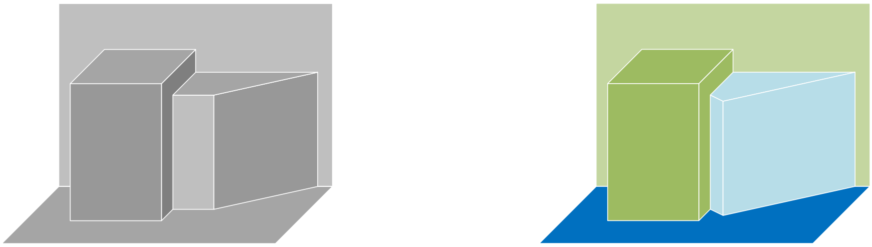
Figure 5.15: Building of a bounding box for an object consisting of three planes

for each object a bounding box is calculated. The bounding box, which is calculated will be cuboid. Therefore, the normal vector $\vec{n}^{(i)}$ of the plane $S_i^{(k)}$ that represents the segment i, which consists out of the most data points within the object O_k is taken as one orientation of the cuboid ($\vec{o}_1^{(k)} = \vec{n}^{(i)}$). The second orientation vector $\vec{o}_2^{(k)}$ results from the direction vector $\vec{d}_c^{(ij)}$ of the longest crossing edge between the plane $S_i^{(k)}$ and any neighbored plane S_j. S_j has not to be mandatory connected to $S_i^{(k)}$ by a convex edge. Since all objects will be modeled as cuboids the third orientation vector $\vec{o}_3^{(k)}$ results from the cross product of $\vec{o}_1^{(k)}$ and $\vec{o}_2^{(k)}$.

$$\vec{o}_3 = \vec{o}_1 \times \vec{o}_2 \tag{5.37}$$

The boundaries of the cuboid are chosen such that all data points $P^{(k)}$, which are assigned to the object O_k lie inside the cuboid. Figure 5.15 shows how the bounding box for the right (bright blue) object in figure 5.13(b) is build. The red vectors in figure 5.15(a) represent the orientation vectors $\vec{o}_1$, $\vec{o}_2$, and $\vec{o}_3$ of the bounding box. Figure 5.15(b) represents the resulting bounding box where all three planes shown in figure 5.15(a) are included within the cuboid. While figure 5.15 shows only the bounding box of one object, figure 5.16 shows the bounding box representation of the whole scene. The boxes of the green and the bright blue object cross each other. Nevertheless, each box is defined by a certain position and size. The bright green back plane is an object, which consists of only one plane and thus the object also will only be represented by this one plane.

The advantage of building these bounding boxes is that each object is defined as a simple cuboid shape, which can be outputted really easy. Although the shape of an object is really primitive especially its position is defined very accurate and also the dimensions of the cuboid give an impression of how big the obstacle actually is. Also very important is to mention that the cuboid representing an object never will be smaller than the object really is since always all data points assigned to the object will be within the corresponding cuboid space. Thus objects rather will be represented bigger than they really are than smaller.



(a) Segmented planes of a 3D scene

(b) Bounding box representation of a 3D scene

Figure 5.16: Modeling of bounding boxes out of a 3D plane segmented scene

After all bounding boxes are calculated there will be planes and objects, which are completely hidden behind other objects. These objects will be removed since they do not have to be considered for the obstacle illustration.

5.3 Obstacle illustration

In this section a scheme is described that gives a suitable illustration of the obstacles detected by the image processing algorithm, which was developed during this thesis. Since usually the modeling of objects results in a number of several objects, it would be overcharging to output all of the detected obstacles. This is the reason why a method was developed such that only objects which really represent a urgent danger are illustrated. Since a person walking though a room usually moves more or less forward and does not make any sudden movements sideways, it is important to illustrate objects in front of a person more detailed than objects beside a person. It is also not that important to output the accurate dimensions of an object, which is still far away from a person since the probability that the moving person never meets this object is very high.

Based on this assumptions a scheme was developed, which separates the room in front of a visually impaired person into different areas. Thus objects, which are detected within one or the other area are illustrated differently. For the scheme presented here three different areas are defined, in which obstacles will be illustrated as given in figure 5.17. Until now the areas are just build on 2D constrains. Nevertheless, for the future also a vertical division of the room is worth to be considered. In the presented scheme there are two frontal zones defined, one is the close range zone, which is marked red in figure 5.17 and the second one is the far range zone, which is marked blue in the figure. A third zone is the gray marked side zone, which is left and right of the close range zone.

The close range zone is that zone, in which objects can be considered as dangerous obstacles. Thus objects, which are within this zone have to be outputted very precise to the blind person so that the person is able to navigate around it. Thus in this zone obstacles will be outputted with the size of their bounding box and also their 3D position. As size the projection of the object onto the x-y-plane is illustrated since this is the area the object requires from blind's perspective. The y-component of the

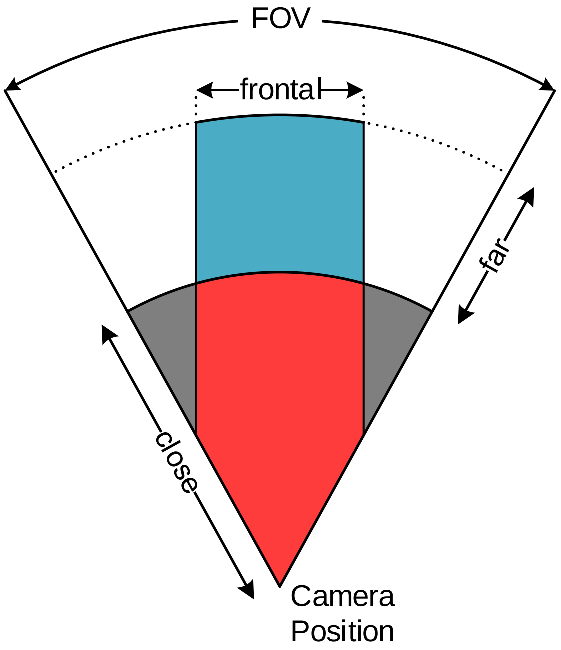


Figure 5.17: Field of view divided into zones for obstacle illustration

object is given as distance above the floor. Since objects most time are laying on the floor or are hanging far above it, the distance from the lowest edge of the object to the floor plane is of most interest and thus is outputted.

Since the far range zone is also frontal to the camera and thus also to the moving person, it is the zone, which the person reaches if she or he does not turn sidewards. Thus it is also the zone, which the person is expected to reach next after the close range zone. If there is no object within the close range zone, object within the far range zone are presented to the blind. Since objects in this range are not acute obstacles yet for the blind, they do not have to be illustrated as precise as in the close range zone. Thus these object only will be represented by their distance to the blind person. The precise position of these objects is not really of interest since the blind person will slightly change her or his direction anyways. Besides, the position already is roughly defined since the object is within the frontal area and thus directly in front of the person. The size of the object also can be more or less neglected because very small objects will not be detected as single objects in far ranges. Thus all objects detected within this range are at least above a certain size.

Objects within the side range are not obstacles at all to the moving person as long as those objects do not move into the close or far range. This would happen if the person turns left or right. To avoid that the person turns sideways and immediately runs into an obstacle the objects in the side range are just signalized as present. This means the person is informed about the presence of objects within the side range but without any position and size information.

All objects, which are outside of all three areas will not be illustrated to the blind person at all. The number of objects outside of all areas will be very little anyways since the upper boundary of the far range will be defined such that it is about the upper depth information boundary of the depth image, which is in between 3 m to 4 m. The upper boundary of the close range should be set to about 1.5 m to 2 m such that the moving person is still able to avoid upcoming obstacles. For the frontal area in first test a width of about 120 cm was considered to be suitable.

As already mentioned before, a vertical division of the scene could also be helpful. Thus very high objects could be ignored since they would be above the height of the blind person. Thereby, also very flat objects on the ground could be classified as small steps. To these steps also attention has to be payed but nevertheless they do not represent insuperable obstacles. Algorithm 5 states the presented obstacle illustration.

Algorithm 5 Obstacle Illustration

```
find the object $O_k$ which is closest to the camera center $C$
if $O_k$ is within close range then
    output: $x$- and $y$- dimensions of $O_k$
    output: min. distance of $O_k$ to camera center $C$ in $x$- and $z$-coordinates
    output: min. distance of $O_k$ to floor plane $S_f$
else
    if $O_k$ is within far range then
        output: min. distance of $O_k$ to camera center $C$
    end if
end if
if any object $O_i$ is within the left side range then
    output: potential obstacle on the left
end if
if any object $O_i$ is within the right side range then
    output: potential obstacle on the right
end if
```

Chapter 6

Software implementation

This chapter describes how the Microsoft Kinect is controlled by software and also how the algorithm presented in chapter 5 *Development of algorithms* is implemented. During the development and implementation of the algorithm it was very important to have good methods to check the results of single processing steps. Thus it was decided to implement all algorithms firstly in MATLAB, since it offers various opportunities to visualize 2D as well as 3D data. During this thesis the focused lay not on developing a image processing algorithm which works in real time, but on developing a first algorithm at all which supplies accurate and reliable results. Thus the slower computation speed which comes with MATLAB compared to C or C++ for example can be disregarded. Besides, the computation effort the Kinect library already needs is very high and thus for running both, the Kinect software and the developed image processing algorithm in parallel, fast processing units would be needed. For running the Kinect itself already a dual core CPU[1] with a 2.6 GHz clock rate is recommended by Microsoft and the processor load confirms that it is really needed.

Since it was not possible with the available hardware and in the available range of time to build a real time system, it was decided to do all processing offline. Nevertheless, the focus lay not on building a real time system, but to develop a reliable and accurate algorithm, as already mentioned. Thus a software in C# was developed to control the Kinect sensor and to read out the camera and sensor data. Afterwards the depth image data is read in by a MATLAB function and the developed algorithm is applied.

In the following two sections, first the software for controlling the Kinect is presented (section 6.1) and afterwards the MATLAB implementation of the image processing algorithm is presented briefly (section 6.2).

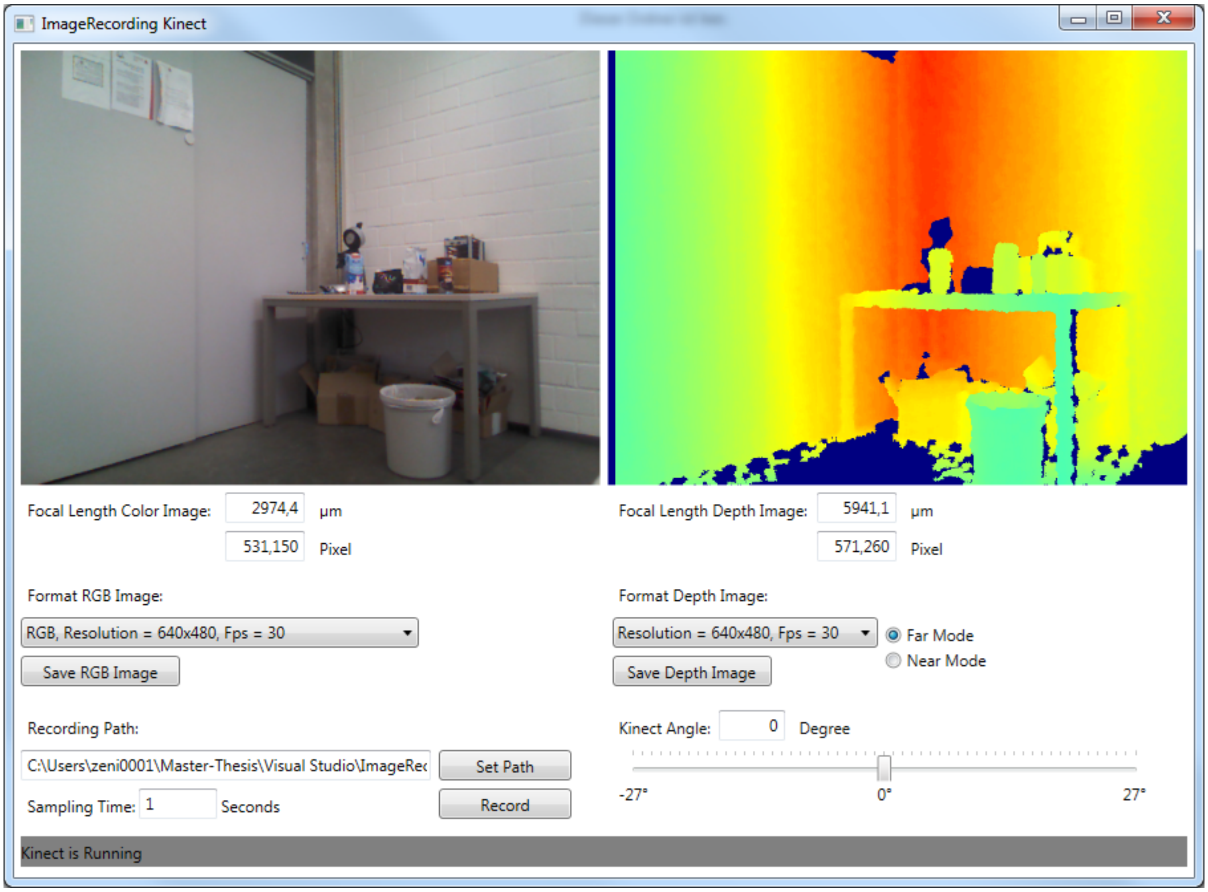
6.1 Controlling of Kinect

The software which controls the Microsoft Kinect is written in C# and was developed in Microsoft Visual Studio 2010. Since the program should contain an user interface, it was developed as a Microsoft WPF[2]

---

[1] central processing unit

[2] Windows Presentation Foundation



Focal Length Color Image: 2974,4 µm

531,150 Pixel

Focal Length Depth Image: 5941,1 µm

571,260 Pixel

Format RGB Image: RGB, Resolution = 640x480, Fps = 30

Format Depth Image: Resolution = 640x480, Fps = 30

● Far Mode

○ Near Mode

Recording Path: C:\Users\zeni0001\Master-Thesis\Visual Studio\ImageRec

Sampling Time: 1 Seconds

Kinect Angle: 0 Degree

Kinect is Running

Figure 6.1: GUI for controlling Microsoft Kinect

application. Figure 6.1 shows the GUI[3] which was built to control the Kinect.

In this thesis the Kinect software was developed by using version 1.5 of the official Microsoft Kinect SDK. Now there already exists a newer version of the SDK (v1.6). The software developed in v1.5 is compatible to the new version but not vice versa, since v1.6 supplies some new features.

The development with the Kinect SDK is really straight forward. Here the structure of a Kinect program will not be described commonly but by leading though the program developed during this thesis. Nevertheless, this structure is similar to almost any other program developed with the Microsoft Kinect SDK.

In the developed program a new class was build, which is called `MyKinect`. This class is organized such that only the Kinect functions needed are supplied. When an object of this class is build it is checked in the constructor how many Kinect devices are connected to the computer. Since we always have only one device connected, the first device which is available will be selected and thus becomes the controlled device. If more the one Kinect device have been connected, always only the first device would

---

[3] graphical user interface

be controlled by the developed software. For the selected Kinect device already exists an object of the class `KinectSensor`. This object will be stored in `MyKinect` and its methods can be call to control the Kinect device. In the following sections the most important methods implemented in `MyKinect` will be presented. All those methods are listed in table 6.1.

Name of the method	**Input variables**	**Return type**
`startKinect`	none	bool
`stopKinect`	none	bool
`IsRunning`	none	bool
`initialize_RGB`	`RGB_Format`, `ColorFrameReady`	bool
`initialize_DepthImage`	`Depth_Format`, `DepthFrameReady`	bool
`initialize_Depth_and_RGB`	`RGB_Format`, `Depth_Format`, `AllFramesReady`	bool
`saveLastColorFrame`	`path`	bool
`saveLastDepthFrame`	`path`	bool
`storeColorFrame`	none	bool
`storeDepthFrame`	none	bool
`flipPixeldata`	`Image`	byte-/short-array
`startRecording`	`path`, `sampTime`	integer
`stopRecording`	none	integer
`setDepthRange`	`range`	bool
`getFocalLengthDepth`	none	float
`getFocalLengthColor`	none	float
`setAngle`	`value`	bool
`getAngle`	none	integer

Table 6.1: Methods of the `MyKinect` class

6.1.1 Enable/Disable Kinect

The methods `startKinect` and `stopKinect` are used to get the Kinect running or to set it back to a standby mode. The advantage of setting the Kinect to a standby mode is that no further processing for the Kinect is done and the CPU power can be used for other tasks. Since the processing performed by the Kinect software needs a lot of computation effort, switching of the device can be very helpful e.g. while storing an image. Both methods `startKinect` and `stopKinect` return a Boolean variable, which states if the switching off respectively the switching on procedure was successful. The method `IsRunning` can be called to check whether the Kinect is running or not.

6.1.2 Initialize Kinect cameras

The methods `initialize_RGB`, `initialize_DepthImage`, and `initialize_Depth_and_RGB` are used to initialize the Kinect cameras. Therefore only the image format in which the images shall be received and some event handlers have to be set. The image format is of the data type `ColorImageFormat` for the color camera and respectively `DepthImageFormat` for the depth image camera. Both formats are defined in the Kinect library and define the image resolution as well as the image encoding.

In the methods `initialize_RGB` and `initialize_DepthImage` the event handlers `ColorFrameReady`, respectively `DepthFrameReady` are submitted beside the corresponding image format. This event handlers are submitted to the Kinect object of type `KinectSensor` and thus the corresponding events are triggered when a new color, respectively depth frame is received. The method `initialize_Depth_and_RGB` can be used if both cameras want to be initialized simultaneously. Here a shared event handler `AllFramesReady` is submitted. The corresponding event is only triggered if both a color and a depth frame are received. The return value of all there methods is used to signalize whether the processing was successful or not.

6.1.3 Saving images

To save a single color image two functions are needed. This are the methods `saveLastColorFrame` and `storeColorFrame`. When the method `storeColorFrame` is called the color frame which will be received next from the Kinect is stored within the `MyKinect` object. In the next step the method `saveLastColorFrame` has to be called. This method saves the image which currently is stored within the `MyKinect` object on the hard disk of the computer. The image will be encoded as bitmap file and will be stored under the path submitted to the method as the input parameter `path`. Before the image is stored the method `flipPixeldata` is called, which mirrors the image since the one received by the Kinect device is mirror inverted.

The same procedure which is perfromed by `saveLastColorFrame` and `storeColorFrame` for the color image do `saveLastDepthFrame` and `storeDepthFrame` for the depth image. The only difference is that the depth image will not be stored in a bitmap file but in a text file (.txt). In this file the depth information is stored within a matrix pixel by pixel. Each depth value is represented by a variable of the data type `short`. In this `short` variable the three LSBs[4] are always zero. The 12 next higher bits represent the unsigned depth information. The MSB[5] signalizes whether a pixel has a valid depth information or not. For all valid depth values the MSB is equal to zero. For all pixels where the depth information could not be gained out of the IR image the MSB is equal to one and thus the `short` variables representing the corresponding depth values will contain a negative value. Pixels representing points which are above the upper depth range will be represented by the highest possible value ($2^{12}-1$).

6.1.4 Recoding image streams

The methods `startRecording` and `stopRecording` are used to record streams of images. The method `startRecording` has the input parameters `path` and `sampTime`. The parameter `path` gives the path to a directory on the hard disk where all recorded images will be stored. At the destination given in `path` a new folder is created. The folder's name represents the current date and time (*YYYY_MM_DD-HH_MM_SS*). The second parameter `sampTime` defines the sample period of the recorded image stream in seconds. Since for the current applications all images are processed offline, no high sample rates are

---

[4] least significant bits

[5] most significant bit

needed. Besides, the CPU will not be able to store streams with frame rates of several fps[6]. Thus the minimum sample time which can be selected is 1 s (max. sample rate = 1 fps). When the recording is started color images as well as depth images are stored. The images are stored by an event function, which is triggered by a timer. Thus every period, which is defined by `sampTime` one depth as well as one color image is stored in the created folder. The method `stopRecording` will stop the recording.

6.1.4.1 Different parameters and settings

Beside the main functions described above some methods are implemented to set or read out Kinect parameters. One of these methods is `setDepthRange` which is used to switch between the *near* and *default (far)* mode of the depth image camera. Therefore the variable `range` of data type `DepthRange` is submitted to the method. Variables of type `DepthRange` can either have the value *near* or *default*. The methods `getFocalLengthDepth` and `getFocalLengthColor` can be used to read out the focal length of the depth (IR) and respectively the color camera in milliliters. The focal length f of both cameras is stored in the class `KinectSensor` in pixels and is converted into millimeters based on the pixel dimensions given in the data sheets (9) and (1).

By calling the methods `setAngle` and `getAngle` the vertical tilt angle can be set or read out. This angle can be any integer in between $-27°$ and $27°$.

6.1.5 Graphical user interface

The GUI is that program part from which all the functions described in the section above are called. The complete GUI is shown in figure 6.1. It displays a live image of the Kinect's color camera (left image) as well as a color coded representation of the depth image (right image). The color code used is the HSV[7] color map which was received from MATLAB. The color map is stored in the program in a LUT[8] and is assigned to the depth image pixels based on their depth value. Both images are updated with the lower one of both sample rates defined in the two selected image formats. Since the images are read mirror inverted from the Kinect, they are also displayed on the GUI like this. Beside the displayed images there are different control items on the GUI. These items are divided into four different regions which are shown in figures 6.2 to 6.5.

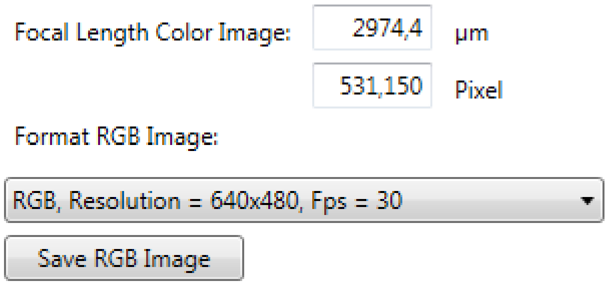
In figure 6.2 all items, which can be used to control the recording of the color image are shown. Two text boxes display the focal length f_{RGB} of the RGB camera which can be read out of the Kinect librariy. The focal length is displayed in micrometers as well as in RGB image pixels. Under *Format RGB Image* the format of the recorded image can be selected. Here one of four formats can be chosen (RGB, 1280 × 960 at 12 fps; RGB, 640 × 480 at 30 fps; YUV, 640 × 480 at 15 fps; YUV (RAW), 640 × 480 at 15 fps). The button *Save RGB Image* opens a save file dialog. In this dialog a path can be selected where the current color image is stored.

---

[6] frames per second

[7] hue, saturation, and value

[8] lookup table



Focal Length Color Image: 2974,4 µm

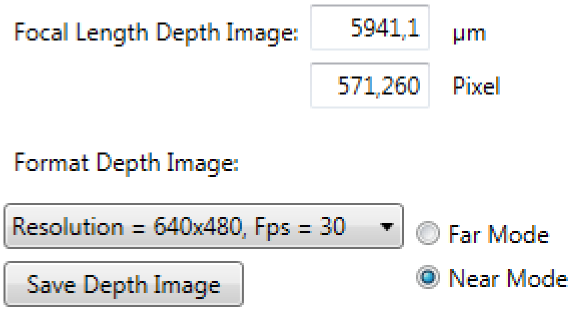
531,150 Pixel

Format RGB Image:

RGB, Resolution = 640x480, Fps = 30

Figure 6.2: Color image settings

Basically the same setting opportunities as for the color image are available for the depth image. The GUI region for this setting options is shown in figure 6.3. Here again two text boxes show the focal length f_{IR} of the IR camera in micrometers as well as in depth image pixels. For the depth image three different image formats, respectively resolution can be selected (640 × 480 at 30 fps; 320 × 240 at 30 fps; 80 × 60 at 30 fps). With the button *Save Depth Image* the last depth image, which was recorded can be stored in a text file. Additionally, for the depth image camera it can be switched between the *default (far)* mode and the *near* mode by two radio buttons.



Focal Length Depth Image: 5941,1 µm

571,260 Pixel

Format Depth Image:

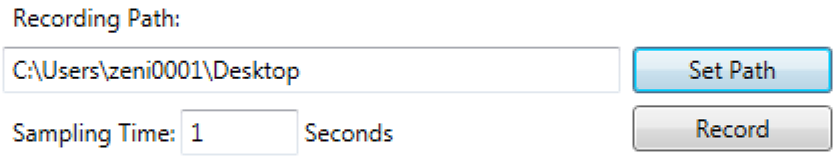
Resolution = 640x480, Fps = 30

○ Far Mode

● Near Mode

Figure 6.3: Depth image settings

Figure 6.4 shows the control items to record a stream of images. Therefore the recording path either can be typed in into the shown text box or can be selected over a dialog box by pressing the *Set Path* button. Additionally the sampling time (period of time between two images) can be set. The sampling time has to be an integer value greater than zero. By pressing the button *Record* the recording will start. This will cause that the text shown on this button changes to *Stop*. By pressing this button again the recording will stop. For each started recording a new folder is created named by the current date and time (*YYYY_MM_DD-HH_MM_SS*) where all images (depth and RGB) of one recording will be stored.

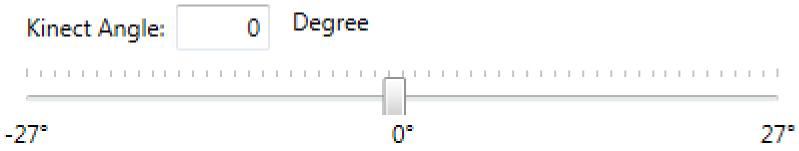


Recording Path: C:\Users\zeni0001\Desktop

Sampling Time: 1 Seconds

Figure 6.4: Image stream settings

A fourth region on the GUI can be used to control the Kinect's tilt motor. This region is shown in figure 6.5. By a slide bar the vertical angle of the Kinect can be varied from −27° to 27°. The set angle is displayed in the text box named *Kinect Angle*. The Kinect angle is not measured in relation to its pedestal but as the difference angle to a straight line orthogonal to the plumb line. This angle is measured inside the Kinect by an accelerometer.



Kinect Angle: 0 Degree

Figure 6.5: Tilt angle settings

6.2 MATLAB implementation

In this section the MATLAB implementation of the camera calibration as well as of the IP[9] algorithm is presented. In chapter 4 *Camera calibration* both RGB camera and depth camera calibration is presented. Both calibrations where also implemented in MATLAB, but since only the depth camera calibration is needed for the IP at the current development state, only this calibration will be described here. Nevertheless, the RGB camera calibration is basically the same as the depth image calibration.

6.2.1 Depth camera calibration

The depth camera calibration is realized in a function named `KinectCalibrationDepth`. This function has no input variables at all since all parameters will be submitted to the function by user commands into the MATLAB console. In this function the user can select, whether she or he wants to record new measurement points or just wants to use an existing set of data points to perform the calibration. If an existing set shall be used, the user has to enter the destination where this data set is stored and the transformation matrix $\hat{\boldsymbol{A}}^*_{Depth}$ will be estimated out of the stored measurement points. For estimating $\hat{\boldsymbol{A}}^*_{Depth}$, exactly the method described in Section 4.4.2 *Depth calibration approaches 2* was implemented. If no existing measurements shall be used, a new set of measurement points has to be recorded first. Therefore the user is asked to enter the path of a calibration image. This image should look like the one given in figure 6.6. On this image each calibration plate must have a defined position within the 3D

---

[9] image processing

world coordinate system. Then the user is asked how many calibration plates are within the image and at which positions they are. Out of these information for each plate the 3D world coordinates of the four corners are calculated. Afterwards the user is requested to mark the corresponding corner positions in the depth image by setting cursor points. Since the edges and corners of the calibration plates in the depth image are very blurry, for each corner two points are set. One point representing the exact corner coordinates and one representing a reliable depth value as shown in figure 6.6. The user can load as many depth images as wanted to collect a large number of calibration points. After all calibration points are recorded the set of calibration points will be stored and the estimation of the transformation matrix $\hat{\boldsymbol{A}}^*_{Depth}$ is performed. After finishing the estimation the function `KinectCalibrationDepth` returns the variable `Adepth` containing the transformation matrix $\hat{\boldsymbol{A}}^*_{Depth}$.

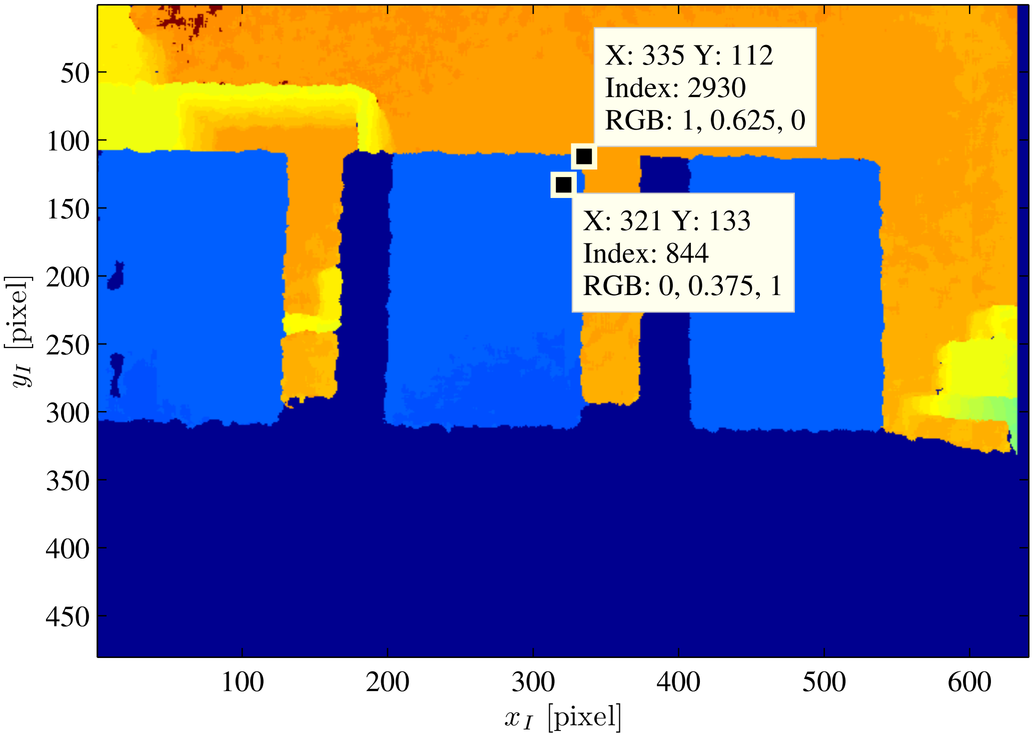


Figure 6.6: Calibration depth image with includes cursor points

6.2.2 IP algorithm

The implementation of the IP algorithm in MATLAB is realized such that only one function has to be called to perform the whole processing. This function is called `obstacle_detection` and has the input parameters `path`, `depth_image`, and `plot` as given in table 6.2. The parameter `path` is used to submit the path of file to the function containing the depth image. The name of the file is written in `depth_image`. The third input parameter `plot` is used to signalized whether results of single sub processing steps shall be plotted or not. Thus, if the results are supposed to be plotted `plot` has to contain the string *plot*, otherwise the variable must not be given to the function. The only thing `obstacle_detection` does is to call two different functions. First the function `DepthSegmentation` is called, which performs the depth image processing and object modeling. This objects then are given to the second called function (`obstacle_output`) which realizes a textual obstacle output as described in section 5.3 *Obstacle illustration*. In the function `DepthSegmentation` the depth image file is read. Additionally, the transformation matrix `Adepth` is read from a file. Since in the development process the IP algorithm was changed and extended very often, in `DepthSegmentation` again a function is called which performs the actual image processing. This function is named by `DepthImageSegmentation_YYYY_MM_DD` where `YYYY_MM_DD` represents the date of development of the function. This allows to change functions within the algorithm without overwriting the old ones. All versions of `DepthImageSegmentation_YYYY_MM_DD` must have the same input and output parameters and thus just can be replaced by each other.

Variable name	**Data type**
`path`	string
`depth_image`	string
`plot`	string

Table 6.2: Input parameters of `obstacle_detection`

6.2.2.1 Segmentation

The function `DepthImageSegmentation_YYYY_MM_DD` has basically only two input parameters as given in table 6.3, which are used for the image processing. A third and fourth one can be submitted to either close all open MATLAB figures or to signalize whether sub processing results shall be plotted. The matrix `DepthImage` contains the depth image recorded by the Kinect, which is a matrix of integer values. The size of the matrix is equal to the image resolution since each pixel is represented by one depth value. `Adepth` contains the estimation of the transformation matrix $\boldsymbol{A}^*_{Depth}$ as defined in section 4.4.2 *Depth calibration approaches 2* in *double* precision. The return parameters of `DepthImageSegmentation_YYYY_MM_DD` are given in table 6.4. The structure-array `objects` contains all objects which were build by the image processing algorithm. Each object again contains a structure-array of up to the length six defining all planes, which describe the object's boundary box. Each of these planes again is described by its corner points. The structure of one object element of the `objects`-array is shown in table 6.5. The variable `field_of_view` bounds the area seen by the depth camera. This

Variable name	**Data type**
`DepthImage`	Integer (Matrix of size $N \times M$)
`Adepth`	Double (Matrix of size 4×4)

Table 6.3: Input parameters of `DepthImageSegmentation_YYYY_MM_DD`

Variable name	**Data type**
`objects`	Structure-array
`field_of_view`	Structure

Table 6.4: Return parameters of `DepthImageSegmentation_YYYY_MM_DD`

field of view can be calculated out of the chip size of the camera sensor as well as the camera's focal length. Thus, its position in world coordinates is dependent on the transformation matrix $\boldsymbol{A}^{*}_{Depth}$. Since the field of view is only defined for displaying purpose it was adjusted manually to fit to the recorded image. If in the image a floor plane is detected, this floor plane represents the lower margin of the field of view. The field of view is defined in `field_of_view` basically in the same way as an object in the `objects`-array. The field of view consists of four planes consisting out of exactly four points, which are connected to each other.

Element	**Sub element**	**Sub sub element**
Object(i)	Plane(1)	Point(1) = $\{x_W^{(i11)}, y_W^{(i11)}, z_W^{(i11)}\}$
		Point(2) = $\{x_W^{(i12)}, y_W^{(i12)}, z_W^{(i12)}\}$
		$\vdots$
	Plane(2)	Point(1) = $\{x_W^{(i21)}, y_W^{(i21)}, z_W^{(i21)}\}$
		Point(2) = $\{x_W^{(i22)}, y_W^{(i22)}, z_W^{(i22)}\}$
		$\vdots$
	$\vdots$	$\vdots$

Table 6.5: Structure of one Object (Object (i))

All thresholds and parameters needed for the image processing are defined at the beginning of the function `DepthImageSegmentation_YYYY_MM_DD` and thus can be changes easily. The most important parameters are shown in table 6.6 and will be described below.

A precise description of the function `DepthImageSegmentation_YYYY_MM_DD` and its sub functions will not be given in this thesis, since the algorithm was described in detail in chapter 5 *Development of algorithms* and the MATLAB implementation is an one-to-one conversion of the described algorithm. The above given functions provide all knowledge needed to use the implemented MATLAB program and all important adjustable parameters are presented int the following paragraphs (table 6.6).

Parameter	**Variables**
Step threshold T_{Step}	`th_step`
Size of median filter N_{MED}	`N_med`
Gradient threshold T_g	`th_grad`
Depth threshold T_d	`th_depth`
RANSAC Threshold T_{RAN}	`th_ran`
Minimum segment size	`min_seg`
Optical camera center C	`camera.center`
Camera focal length f_{Depth}	`camera.f`
Camera sensor size	`camera.chip_size`
Camera orientation Matrx	`camera.R`
Distance from C to floor plane S_f	`floor.dist`
Normal vector of floor plane	`floor.normal`

Table 6.6: image processing parameters

Step threshold T_{Step} If the absolute value of the depth image gradient $\|\vec{g}_d(x_I, y_I)\|$ is above the threshold T_{Step}, stored in `th_step`, the corresponding pixel $p(x_I, y_I)$ is considered to be a step within the depth image.

Size of median filter N_{MED} In `N_med` the size/order of the used median filter is stored. The median filter, which is used is a 2D squared filter with the dimension $N_{MED} \times N_{MED}$.

Gradient threshold T_g T_g is the gradient threshold for the gradient based segmentation described in paragraph 5.2.1.2 *Gradient based segmentation.* The value T_g is stored in the variable `th_grad`.

Depth threshold T_d The depth threshold stored in `th_depth` is other than T_d defined in paragraph 5.2.1.2 *Gradient based segmentation* not a constant value. The variable is a vector of the length two defining the threshold T_d as a linear function of the depth information $d(x_I, y_I)$ as given in equation 6.1. In this equation `th_depth` is denoted as the vector $\vec{t}_{depth}$.

$$T_d(x_I, y_I) = \left(\vec{t}_{depth}\right)^T \cdot \begin{pmatrix} d(x_I, y_I) \\ 1 \end{pmatrix} = \begin{pmatrix} \texttt{th\_depth(1)} & \texttt{th\_depth(2)} \end{pmatrix} \cdot \begin{pmatrix} d(x_I, y_I) \\ 1 \end{pmatrix} \tag{6.1}$$

Thus, T_d has to be recalculated for each pixel $p(x_I, y_I)$. This is done because the accuracy of the depth information decays with increasing depth values $d(x_I, y_I)$. Thus, for higher depth values also the threshold T_d has to be higher to avoid a to fine segmentation at far distances.

RANSAC threshold T_{RAN} The RANSAC threshold is also defined by two parameters `th_ran(1)` and `th_ran(2)`, which define the threshold T_{RAN} as a linear function of the z-component of the world coordinate system z_W. The purpose to do this is the same as described for the depth threshold T_d. In the function performing the RANSAC algorithm the depth information $d\left(x_I, y_I\right)$ is not known, but since z_W is considered to be proportional to it, this values is used to define the linear function.

Minimum segment size The variable `min_seg` defines the number of points/pixels that a segment has to consist of at least. All segments smaller than the value `min_seg` are considered to be outliers and are erased.

Camera parameters Four different camera parameters are used within the segmentation function. The focal length f (`camera.f`) as well as the chip size (`camera.chip_size`) were received from the Kinect SDK respectively the IR camera data sheet (1). The other two camera parameters, which are the optical camera center C in world coordinates (`camera.center`) as well as the orientation matrix of the depth camera within the world coordinate system (`camera.R`) where obtained by trail and error. These parameters are needed to define the field of view and thus do not have to be really precise.

Floor parameters `floor.dist` and `floor.normal` define the position and orientation of the floor plane within the world coordinate system. `floor.dist` defines the minimum distance from the floor plane to the optical camera center C and `floor.normal` the normal vector $\vec{n}_f$ of the floor plane in world coordinates. A plane S_i is considered to be a floor plane if both parameters are within a certain range.

Chapter 7

Experiments and evaluation

In this chapter results of the developed image processing algorithm will be presented. First, for one sample image the results of the image processing are presented, including sub procedure results. Afterwards, some comments are given about time consumption and based on other recorded sample images the pros and cons of the camera system as well as the developed algorithm are presented and discussed.

7.1 Sample processing

In this section the depth image segmentation and object modeling of the scene shown in figure 5.1 is presented. Figure 7.1 shows the depth image recorded by the Kinect of the same scene as shown in figure 5.1. In the depth image shown in figure 7.1 for most of the image pixels a depth value is defined and thus the capabilities of the developed algorithm can be presented very well. Only the dark blue parts in the image represent pixels without any depth information. In the scene some objects exist, which are really important to be detected. The table in the back as well as the trash bin are very large objects and collisions with both of them have to be avoided. The blue book laying on the ground also has to be detected since a person can stumble over it. This is very easy in the color image, since the color of the book is different from that of the floor. Nevertheless, to detect the book in the depth image is very difficult, since both the floor and the upper side of the book have basically the same plane normal vector and only a sightly different height.

The first step which is performed is to calculate the gradient $\vec{g}_d\left(x_I, y_I\right)$ of the image shown in figure 7.1, as described in section 5.2.1.2 *Gradient based plane segmentation.* In figure 7.2 the absolute value of the gradient over the depth image $\|\vec{g}_d\left(x_I, y_I\right)\|$ is shown. In this figure all gradient values above $\|\vec{g}_d\left(x_I, y_I\right)\| = 90$ are limited to 90. Those are the gradient values which are considered as steps in the image and are represented as dark red pixels within the gradient image. In the figure it can be seen that the gradient is not really continuous. Especially on the right wall in the image the gradient's absolute value changes from zero to some value unequal to zero and back again. This results from the quantization of the depth information and also from the next neighbor interpolation performed by the

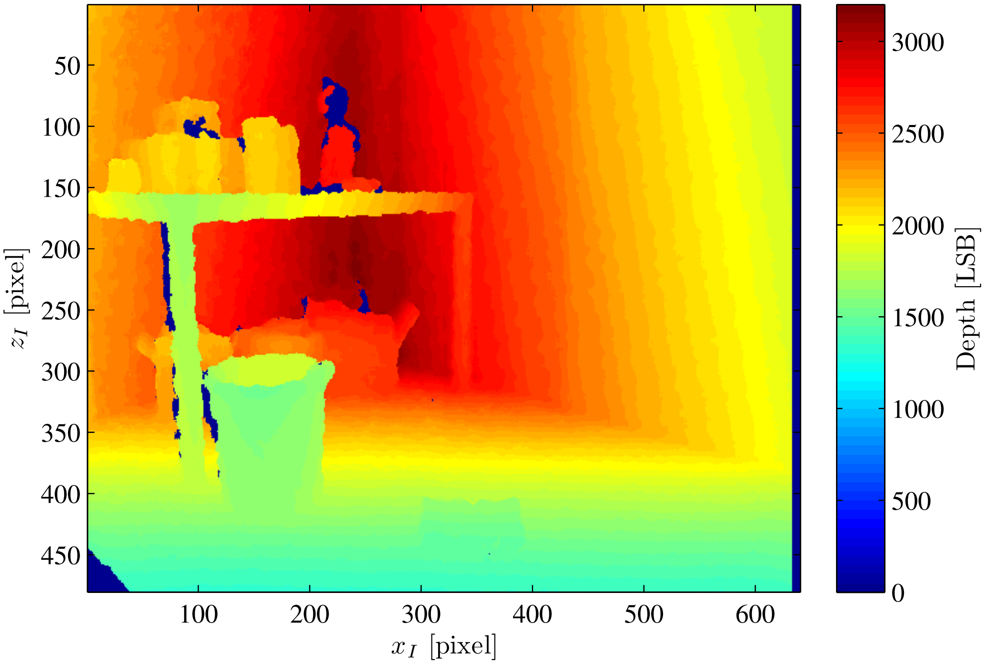


Figure 7.1: Depth image of recorded scene

Kinect. This effect increases with rising depth, because the further objects are away from the Kinect the more the IR pattern is stretched. Thus the resolution of the depth information decreases which is equalized by interpolation. Nevertheless, based on the gradient, especially the book laying on the floor can be seen much better than in the depth image itself. Also the border between the right leg of the table and the right side wall can be seen very well in the gradient image.

To get rid of outliers within the gradient image caused by interpolation and quantization artifacts, as well as noise, the gradient vector $\vec{g}_d\,(x_I, y_I)$ is filtered by a quadratic median filter. For the filtering all gradient values which were prior detected as steps are set to zero to reduce their effect on the filtering result. Figure 7.3 shows the absolute value of the filtered gradient $\|\vec{g}_{d(MED)}\,(x_I, y_I)\,\|$. In figure 7.3 again the pixels representing steps are set to the maximum absolute value (= 60) and are represented in dark red. In the filtered gradient image large outliers are erased compared to the unfiltered image and the gradient became more continuous. Nevertheless, at the right side wall and the wall behind the table almost no diversity in the gradient is left and the gradient is equal to zero for almost the whole area . This lack of diversity of neighbored pixels is the main reason why beside the gradient based segmentation also the RANSAC algorithm is applied. The received depth information of neighbored pixels is not accurate and significant enough to decide whether they correspond to the same plane or not. An advantage of the gradient based method for example is extracting the book from the floor plane, since they are separated by high gradient values.

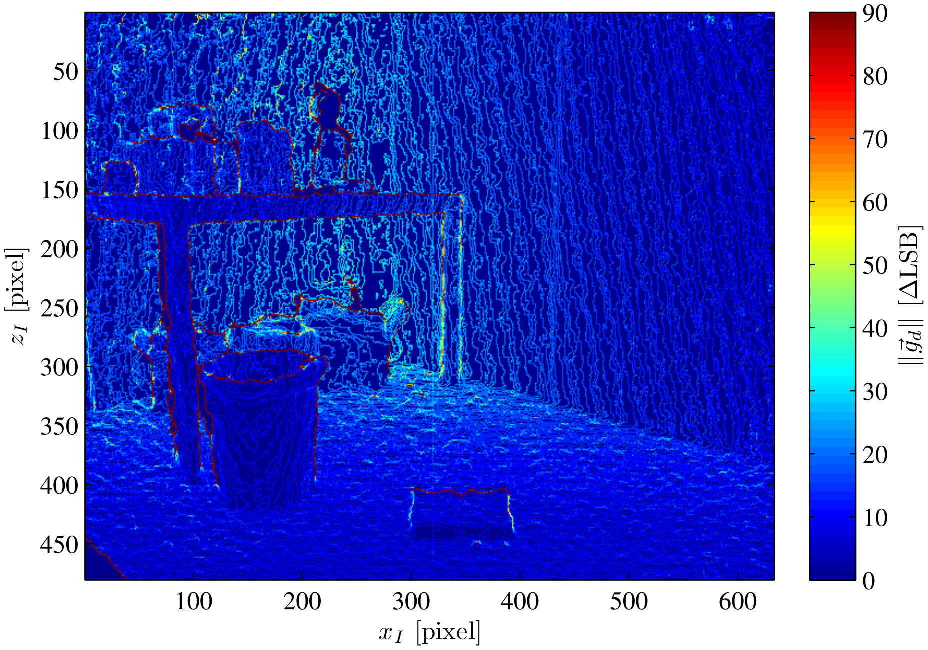


Figure 7.2: Absolute value of the depth image gradient vector field $\|\vec{g}_d\,(x_I, y_I)\,\|$

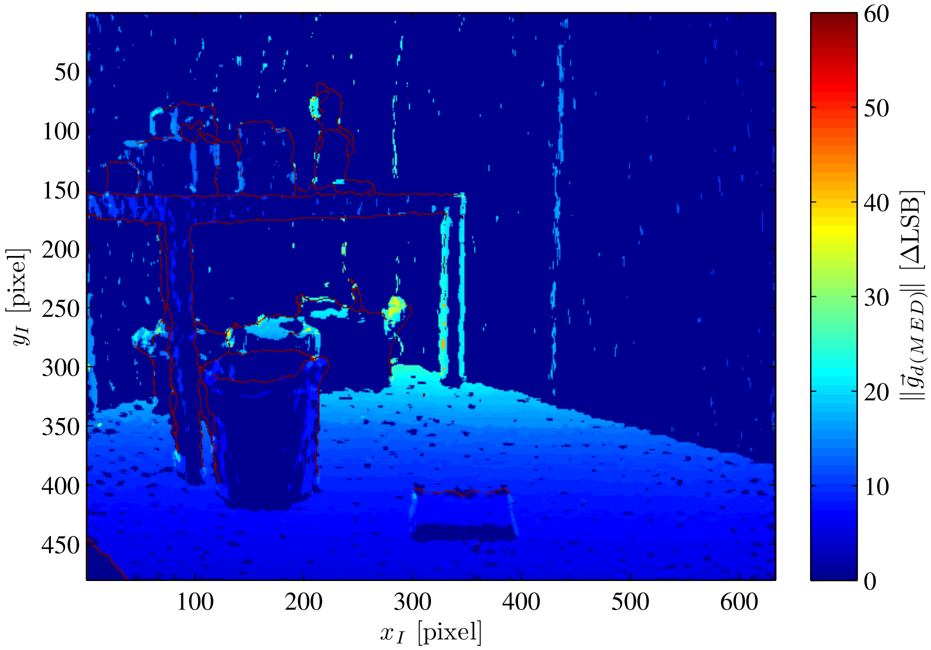


Figure 7.3: Absolute value of the filtered depth image gradient vector field $\|\vec{g}_{d(MED)}\,(x_I, y_I)\,\|$

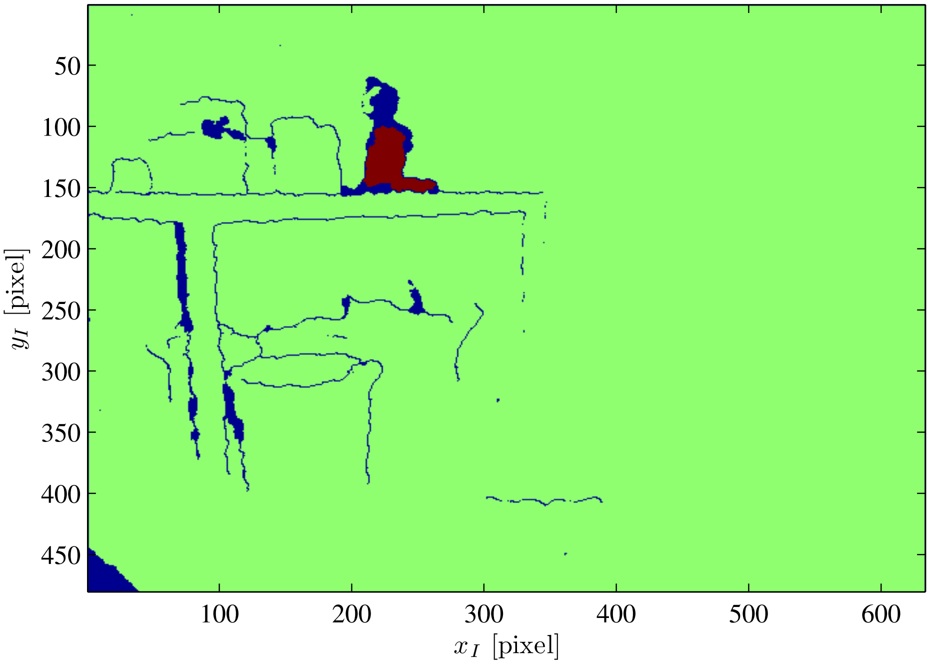


Figure 7.4: Clusters built within the depth image based on steps

Based on the dark red marked steps within the gradient image clusters are build within the image. In the case of the presented depth image this results in two different clusters, since only the body of the coffee maker is totally surrounded by step pixels. The result of the clustering is show in figure 7.4. Here the two clusters are represented by the green and the dark red area. All blue marked pixels are either pixels without depth information, pixels which were classified as steps, or pixels which were assigned to a too small cluster (number of pixels smaller than the minimum segment size).

For each cluster the gradient based segmentation is performed separately as described in section 5.2.1.2 *Gradient based plane segmentation.* As already mentioned is the gradient based segmentation especially useful for objects or planes divided by sharp edges, as they are between the floor and the walls or between the floor and the book for example. Especially at large depth values the gradient based segmentation supplies really bad results, since the depth resolution decays. Thus the threshold for the gradient based method is adjusted for increasing depth values to avoid a to fine segmentation, as e.g. dividing the walls into several segments. Out of this adjustment results that both walls in the image shown in figure 7.1 result in one segment, which has to be separated by the RANSAC algorithm.

Figure 7.5 shows the result after the RANSAC based plane segmentation. Here each segment is represented by a different color. Although the RANSAC algorithm is applied on the 3D point cloud of the recorded image, the segments here are shown in a 2D image representation, since the results can be visualized better. The advantage of the RANSAC algorithm is that large planes are separated very accurate from each other. Thus the crossing edge between those planes is defined very well. For example the walls in figure 7.1, which were still be considered to be one segment after the gradient based

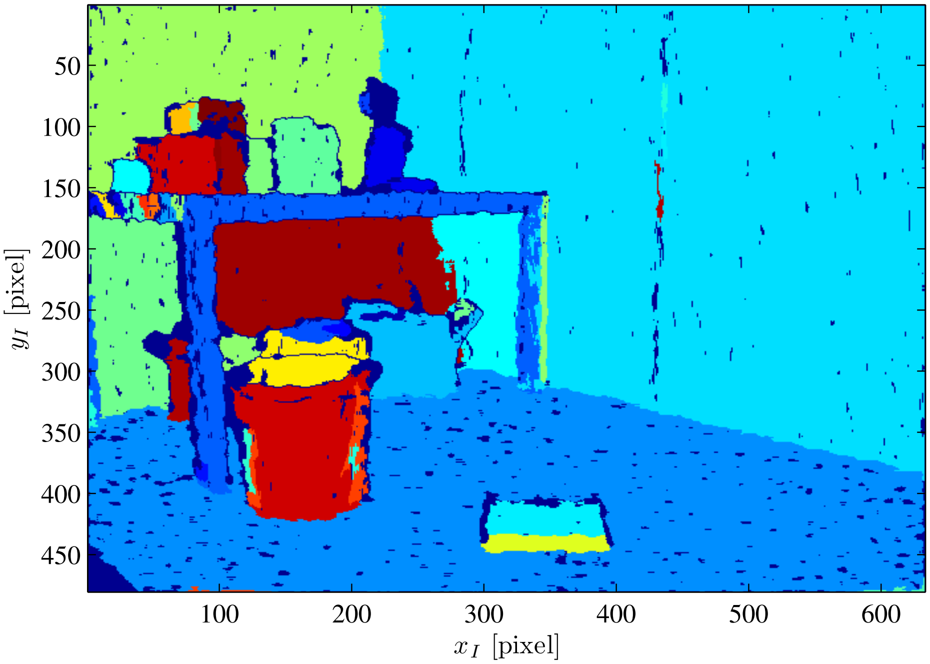


Figure 7.5: 2D representation of plane segments after gradient based and RANSAC segmentation

segmentation are now separated.

A rather bad segmentation result shows the left edge of the table which is separated into several small planes. This results not from the RANSAC algorithm but from the gradient based approach. From figure 7.3 it can be seen that the gradient at this edge is strongly varying and thus the gradient based approach separates this edge into several single segments. To combine those segments again to one plane, the color image segmentation would be helpful. Since in the color image segmentation the whole edge consists of only one segment and since all little planes have almost the same normal vector they could certainly be considered as one plane.

After all the plane segmentation is finished crossing edges between neighbored planes are defined as described in section 5.2.2 *Crossing edges between planes*. Based on this edges the planes are combined to objects as described in section 5.2.4 *Object modeling*. Figure 7.6 shows the 2D image where each pixel is assigned to a specific object. The color of a pixel represents the object to which the pixel is assigned to. Additionally before combining planes to objects the floor plane is extracted since it represents an object which is not an obstacle. Even though the front of the table and the floor are represented by almost the same color, they are not one but two separate objects. Pixels which were after the calculation of the crossing edge on the wrong side of the edge were assigned to the respectively other plane. Thus proper margins between two crossing segments are received as it can be seen in figure 7.6, e.g. between the blue floor and the yellow wall segments. In figure 7.6 it already can be seen that all depth image pixels are assigned to the objects very well. The floor as well as the walls (except the parts which are bordered

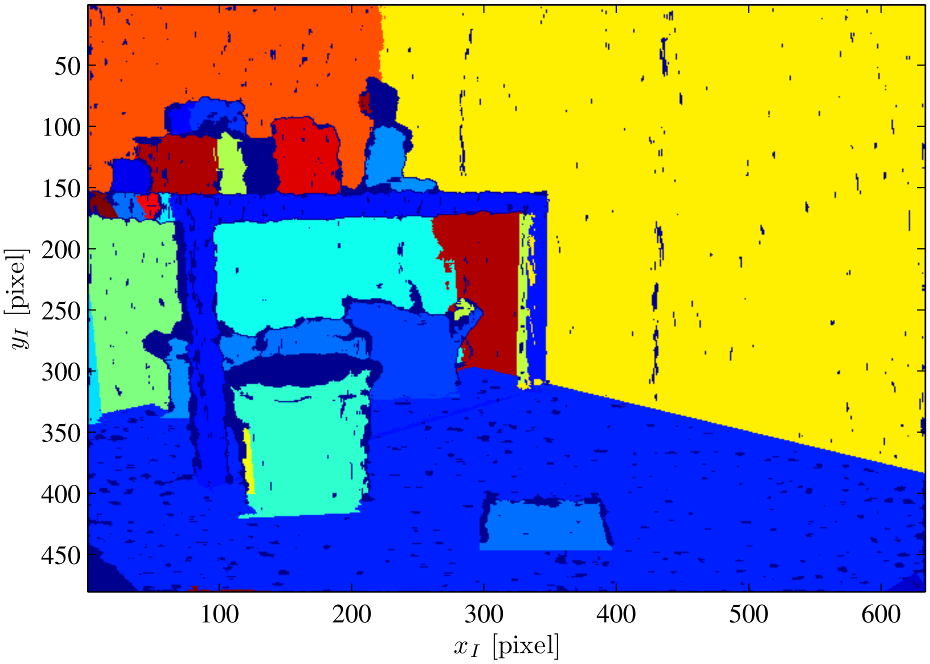


Figure 7.6: 2D representation of pixels connected to objects

by the table) are represented by single objects. The book on the floor as well as the trash bin are also represented by single objects. The only con is that the left side of the table is separated into many small objects. Nevertheless, the front of the table represents one large object.

After combining planes to objects the bounding boxes for the objects are calculated. This is done based on the method described in section 5.2.4.2 *Building objects*. Figure 7.7 shows a 3D representation of the calculated bounding boxes. In this figure the cyan plane represents the floor plane which is the lower boundary of the scene. The whole scene is surrounded by black solid lines which define the FOV. Since outside the FOV can not be captured by the camera, all objects are bounded to the FOV.

Out of the objects shown in figure 7.7 the textual output is generated based on the scheme developed in section 5.3 *Obstacle illustration*. The generated output is presented in table 7.1. For the output presented here a close range of 250 cm, a far range of 400 cm and a side range of ±60 cm were defined. The first two lines in table 7.1 represent the closest obstacle in the frontal zone. This obstacle is the book laying on the floor. Since the book is within the close range it is described by its size (height and width) as well as its position relatively to the camera. Since the obstacle is right in front of the Kinect camera the x-component is represented by the words *in front of you* and not by some value in centimeter. Since the trash bin as well as the right side wall are within the side ranges, they are mentioned by the third line in table 7.1.

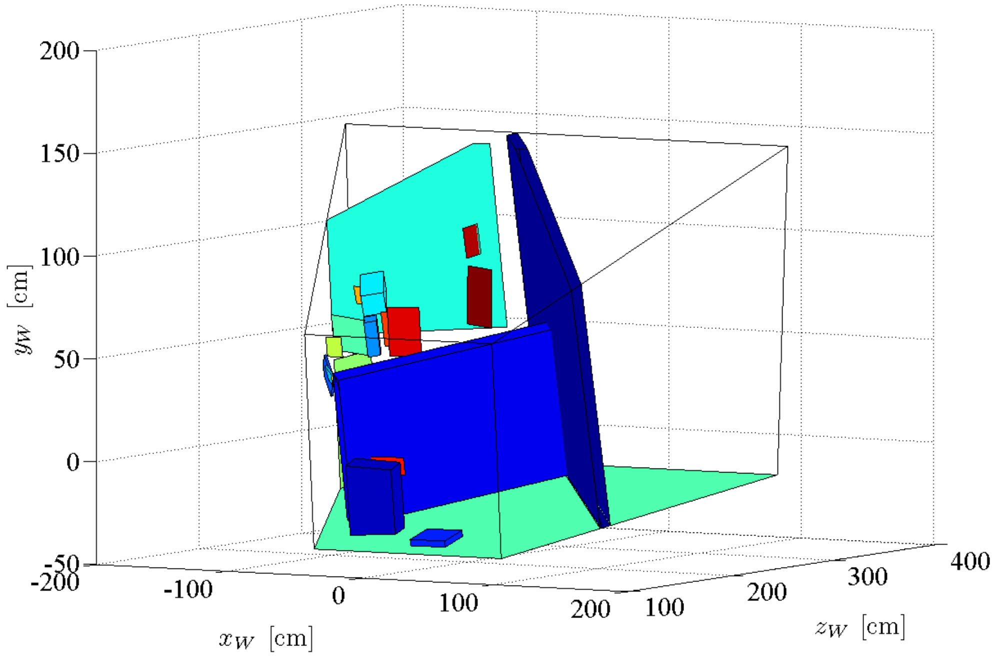


Figure 7.7: Final result of the obstacle detection algorithm

```
The next object is 157 cm in front of you, right on the floor.
The object is 4 cm high and 26 cm wide.
There are potential objects left and right of you.
```

Table 7.1: Textual output of obstacle detection

The example presented in this section showed that the developed image processing algorithms in combination with the Kinect camera system performs very well. All important obstacles were detected by the algorithm. Nevertheless, also some disturbing effects were observed, e.g. the separation of the left side of the table.

7.2 Some poor effects

Section 7.1 *Sample processing* presents a very good result of the developed image processing algorithm. Even though this result is probably representative for the most scenes, there are also some scenarios the Kinect camera as well as the algorithm can not handle very well. All scenarios and effects stated here are based on the figures given in appendix A *Results of IP algorithm.*

The developed algorithm has especially problems to model objects when the recorded depth image is divided into areas of depth information which are separated from each other by undefined pixels. This is especially the case for scenes with many small objects which are shadowing each other in the depth image. This objects usually can be divided into planes but can not be combined to objects afterwards. Figure A.1 shows such a scene. The chairs on the left side of the image result in a fragmentary depth image as shown in figure A.2. Another disadvantage of Kinect's depth image camera is that from some surfaces no depth information can be gained (e.g. the floor in figure A.2). This is the case for mat black surfaces as well as for glossary surfaces. Especially from LCD[1] screens no depth information at all can be gained, since they absorb the whole IR pattern. Also no depth information can be extracted when too intensive sun light is present. Thus, outdoor no depth information at all can be gained but also a scene which is under direct solar radiation, e.g. through a window, will result in a lack of depth information. This can be seen in figures A.7 and A.8. Here for the whole bright area on the floor no depth information is received (dark blue area in figure A.8). In the cases described above the depth information received from the Kinect sensor is insufficient for the algorithm to perform well. This results mostly in a too fine segmentation and a very large number of objects as shown in the resulting 3D scene reconstructions shown in appendix A.

7.3 Time consumption

The time consumption of the algorithm is very high at the current development stage. This come especially form the MATLAB implementation but also the algorithm has to be improved to reach a system which operates in real time. Until now the processing of one image is about 42 s. This value is only a rough average since the time consumption is hardly varying from one processed depth image to the next, depending e.g. on the number of segments built during the processing. The bottle neck functions in the algorithm are especially the segmentation functions and searching routines but also the reshaping of the segments after crossing edges are calculated needs about a quarter of the complete processing time.

Especially the searching routines which basically include the gradient based segmentation as well as the reshaping have to be improved by implementing more efficient algorithms. More complicated will be to improve the computation effort of the RANSAC algorithm which is also very high. But since the RANSAC function used for the segmentation is already a really efficient implementation no great improvements in computation time can be expected here, without paying with segmentation accuracy. Thus, even though the algorithm's time consumption still can be improved by better implementation also faster processing units will be needed to reach a sub-second processing time.

7.4 Discussion

The result presented in section 7.1 *Sample processing* showed that based on a coherent depth image a very good segmentation result can be received. Figure A.3 shows a result of the algorithm where only

---

[1] liquid crystal display

minor mistakes were made in in the segmentation. Even small steps like the book laying on the floor were detected. Figure A.6 shows another image processing result, of a recorded stairs and figure A.4 the corresponding color image. The 3D representation models the recorded stairs very well. Only the side wall is divided into small planes, since they are separated by the handrail.

Nevertheless, the depth image received by the Kinect is very often not coherent and sometimes also insufficient to let the algorithm perform well. Thus the algorithm has to be improved to perform also well for this kind of depth images. For this cases the color image of the scene can be useful, since it shows some object connections better than the depth image. This would also be helpful for planes which have a very steep angle to the image plane (e.g. the left edge of the table in figure A.2). For those planes the difference of the depth information of neighbored pixels is very high. But since the depth information is quantized and the depth of neighbored pixels is also gained by zeroth order interpolation, this planes result in a stepped changing of the depth information. The gradient based segmentation separates those steps into several small planes. Thus the algorithm should be improved such that those plane can be combined later again to receive one connected plane. Therefor also the color image can be taken into account to figure out whether planes of neighbored segments are from the same plane or not. Even though the algorithm still can be improved the Kinect will not be able to handle all kind of scenes. Especially in images with a great lack of depth information it will not be possible to reconstruct the scene very well. This is the case for the scenarios described in section 7.2 *Some poor effects*, where because of some surface properties, too many small overlapping objects or also because of the presents of sunlight no depth information can be gained.

Another disadvantage of the Kinect cameras is the FOV, since for the presented purpose a large angle of aperture would be useful to detect also objects right in front of a person. Besides the range limit of the depth camera is also insufficient for the presented purpose, since no objects further away than 3 m to 4 m from the blind person can be detected.

Nevertheless, under consideration of the given limitations the developed algorithm shows good results which have potential for further development, either with Kinect as camera system or with a different camera system. Even though from some scenes not enough depth information can be gained to remodel it very well, important potential obstacles are still detected as shown in appendix A.

Chapter 8

Conclusion and prospects

8.1 Conclusion

The goal of this thesis was to develop an algorithm for reconstructing a 3D scene by geometrical objects. Based on this representation a blind person should be warned about obstacles in her or his way. The focus of this project lay not on building a complete system which is able to run in real time, but to develop some reliable algorithms. For developing those algorithms the Microsoft Kinect was used, since according to its price it represents a very attractive device for the first development steps. A control software for the Kinect was developed by using the official Microsoft Kinect SDK to receive depth as well as color images from the camera system. For developing the IP algorithm MATLAB was used, since it offers much opportunity to receive data and results from sub functions within the program.

The algorithm presented in this thesis has the ability to reconstruct a recorded scene by 3D objects as defined in the objectives. Besides, an output scheme was developed which informs the blind user about upcoming obstacles. For the IP first different, already existing plane segmentation approaches were analyzed which led to the RANSAC algorithm. This algorithm later was used in combination with some novel approach developed during this thesis to receive a reliable and accurate segmentation algorithm. In chapter 5 *Development of algorithms* the developed algorithm is presented in detail. Until now all image processing is only performed based on the depth image. Nevertheless, some color segmentation algorithms were already analyzed. One of those algorithms was presented in section 5.1 *Color image segmentation*. The presented algorithm probably will be implemented into the whole image processing in further development steps.

The results of the developed algorithm presented in chapter 7 *Experiments and evaluation* showed that the algorithm is able to perform the 3D scene reconstruction as demanded in the thesis objectives. Some experiments showed that the algorithm recognizes all important objects and represents them very well in 3D coordinated. The algorithm also showed the ability to detect very small steps and thus can keep blind people from stumbling over it. Nevertheless, the algorithm in combination with the Kinect also showed problems with special scenarios. For some scenarios the Kinect does not supply a coherent depth image. Instead the image often consists of areas with depth information which are separated by undefined areas which comes e.g. from sunlight irradiation or from inappropriate surfaces. In this scenarios the algorithm has problems to build connected object, since the scene is not described very well. Besides, the Kinect supplies a very stepped depth image which in some cases is not handled very well by the algorithm, since it cases some planes to be divided into small segments.

In conclusion it can be said, that the developed algorithm offers a very good first approach to meet the demanded objectives. Nevertheless, there are still some areas which can be improved in future work.

8.2 Prospects

The developed system already shows very good results but still is not reliable in all kind of scenarios. Before the algorithms can be implemented within a real time system, which really can be used by a blind person, much more research and development has to be done. There exist already a bunch of ideas how the developed algorithms presented in this thesis can be improved. In this section the major ideas which are proposed to be realized in future development steps will be presented.

8.2.1 Implementation of color image segmentation

Introducing color information to the segmentation approach will be one of the most important following steps. The easiest thing would be to combine the color segmentation algorithm presented in section 5.1 *Color image segmentation* with the developed depth image based algorithm. This would give the ability to combine too fine segmented planes again to one plane based on the information received from the color segments. It is not advisable to divide plane segments based on the color information since a plane can consist of many color segments. Nevertheless, it is often likely that one color segment is not part of more than one object.

8.2.2 Image sequence estimation and image stabilization

Since the blind person using the obstacle detection system is usually moving around another development step would be to estimate this motion by the support of accelerometers. Thus the scene of following images can be estimated out of the sensor data and the current image which will reduce the computation effort. The accelerometer data can also be used to realize an image stabilization to reduce the effect of shocks. Additionally obstacle can be detected earlier such that the person can navigate around it.

8.2.3 Obstacle classification

Another step would be to classify the detected obstacles. For example represent stairs not an obstacle for the blind person but an object to which attention has to be paid. Thus stairs should be classified as stairs and presented as such to the blind person.. On the other side stairs going downwards in front of the blind person is not represented by an object. Nevertheless, attention has to be paid to it and thus such stairs has to be classified too.

8.2.4 Analyzing of different camera systems

The analysis of different camera systems would be another development assignment. Section 7 *Experiments and evaluation* stated some disadvantages of the Kinect and thus another step would be to compare other camera systems to the Kinect. The biggest disadvantage of the Kinect is that the depth camera does not work outdoor and from some kinds of surface materials no depth information can be gained. Other systems do not have this disadvantage and also offer maybe better depth resolution. Nevertheless, some other systems need much more computation effort. Thus different camera systems have to be compared to each other to find the best suitable one. Light-field camera, TOF[1] camera, or stereo camera for example would be alternative systems which are worth to be taken into consideration.

---

[1] time of flight

Appendix A

Results of IP algorithm

A.1 Depth image sample 1

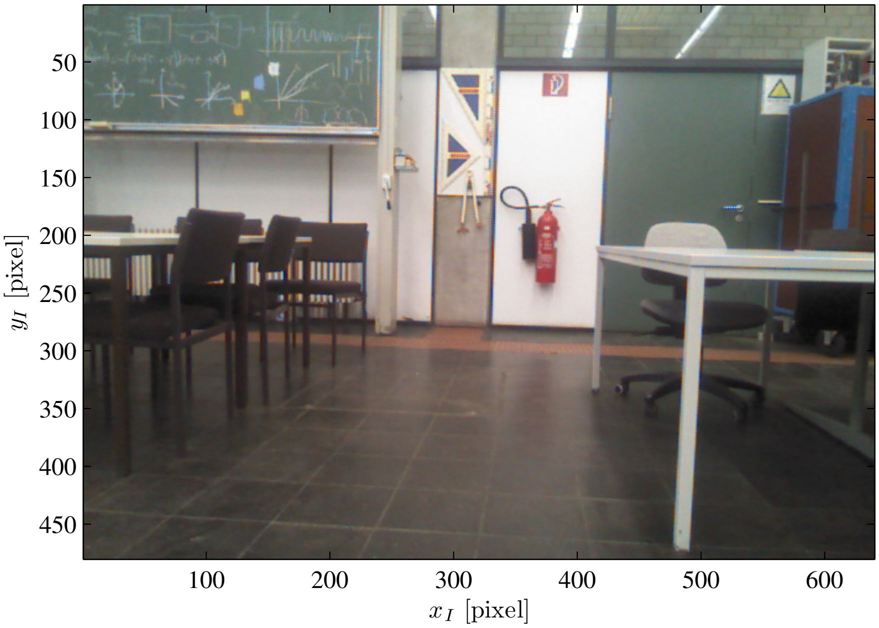


Figure A.1: RGB image of recorded scene (Sample 1: Lack of depth information)

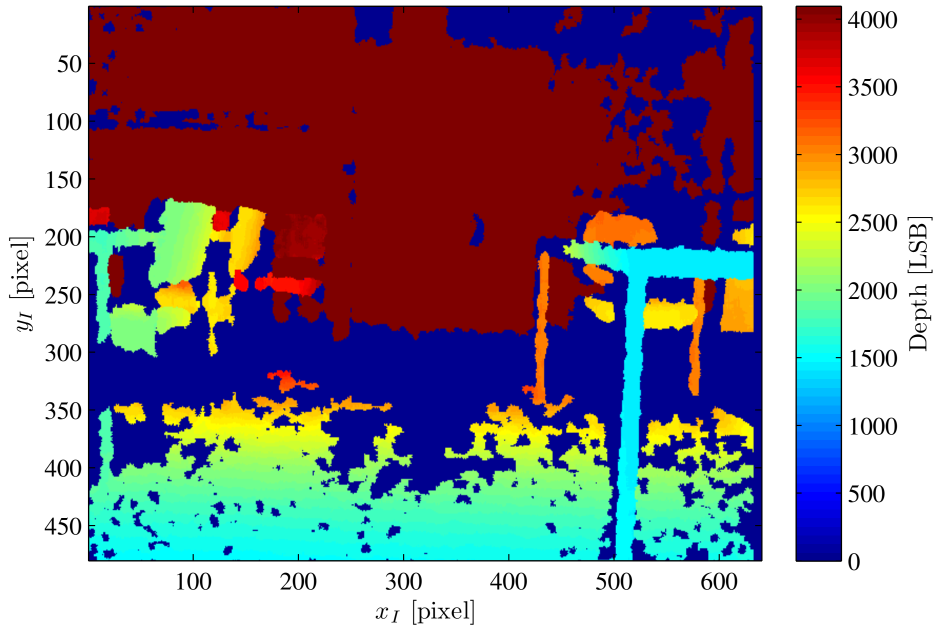


Figure A.2: Depth image of recorded scene (Sample 1: Lack of depth information)

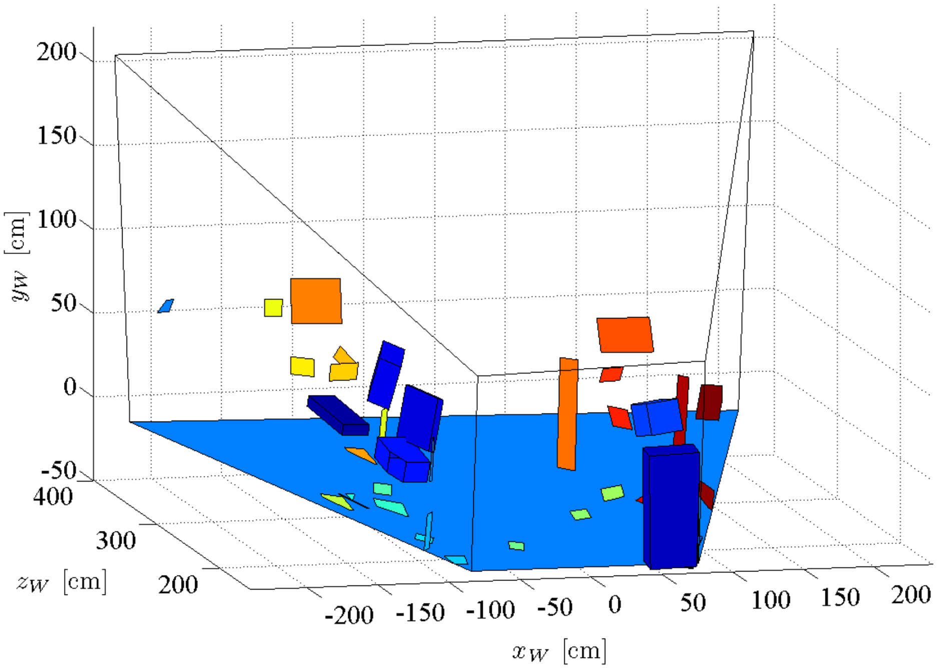


Figure A.3: 3D representation of recorded scene (Sample 1: Lack of depth information)

A.2 Depth image sample 2

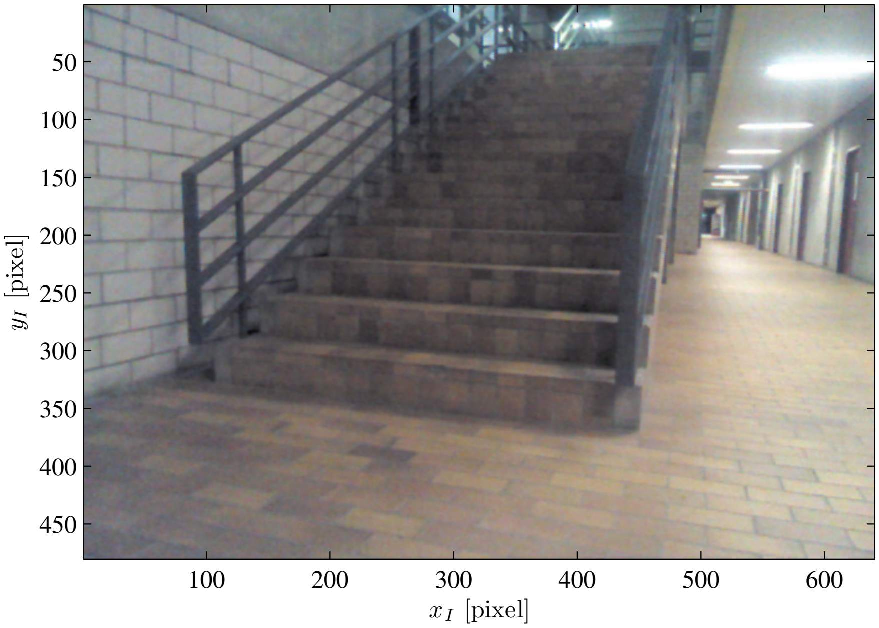


Figure A.4: RGB image of recorded scene (Sample 2: Stairs)

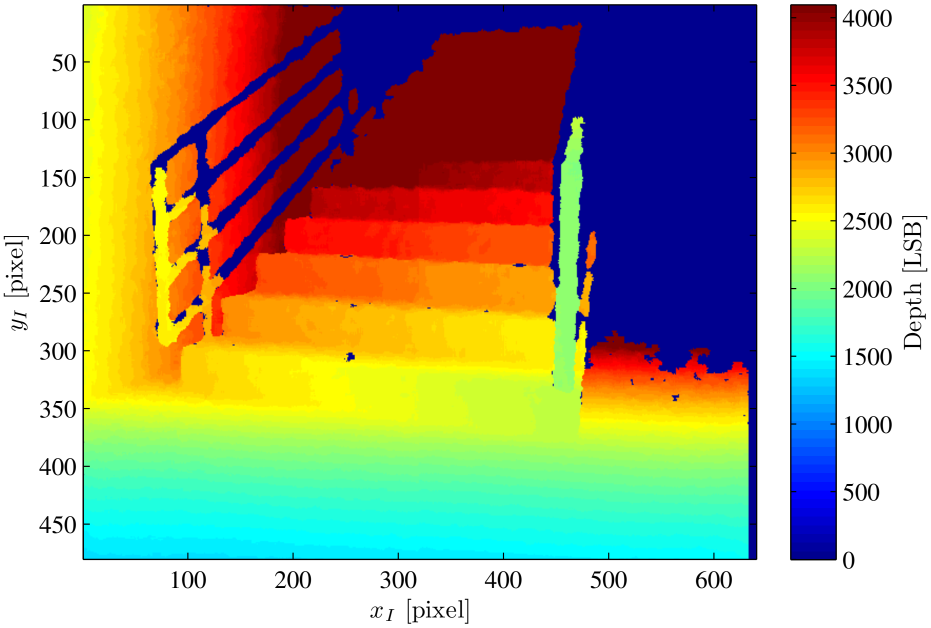


Figure A.5: Depth image of recorded scene (Sample 2: Stairs)

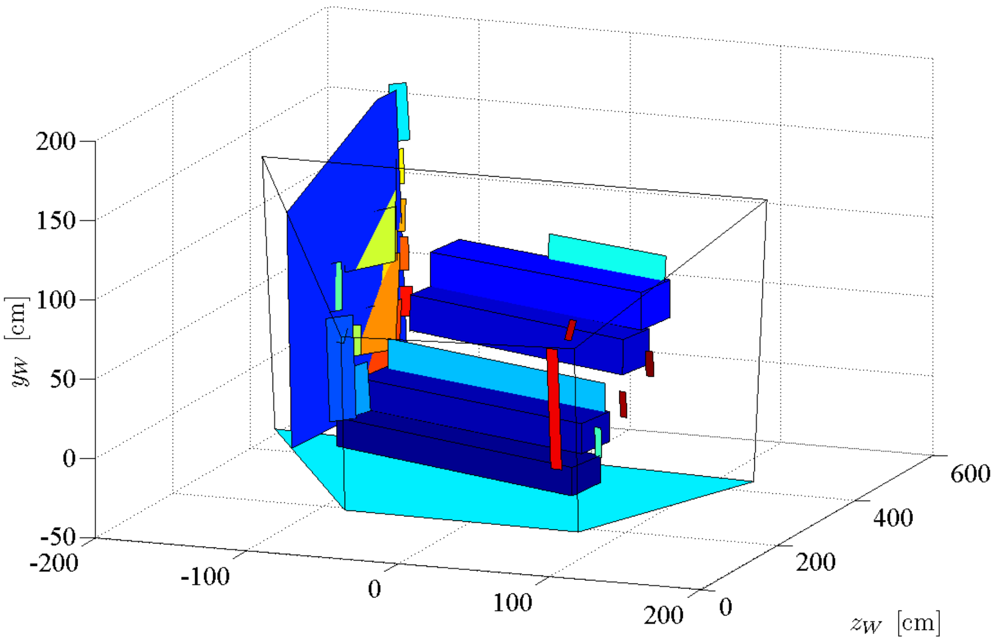


Figure A.6: 3D representation of recorded scene (Sample 2: Stairs)

A.3 Depth image sample 3

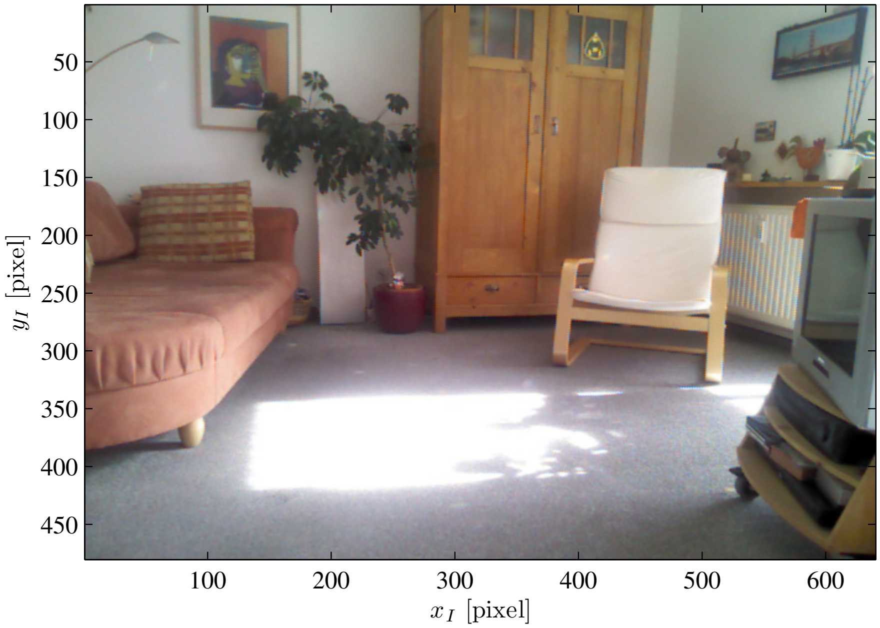


Figure A.7: RGB image of recorded scene (Sample 3: Solar irradiation)

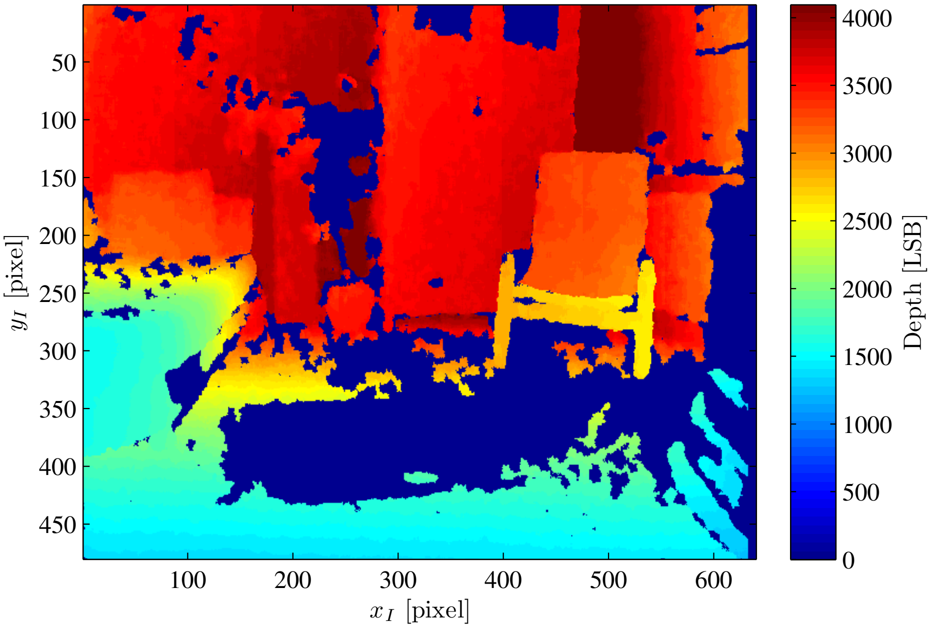


Figure A.8: Depth image of recorded scene (Sample 2: Solar irradiation)

Bibliography

[1] Aptina Imaging, 3080 North 1st Street, San Jose, CA 95134, United States. *MT9M001 Monochrome Image Sensor*, 2011.

[2] M. Barkat. *Signal detection and estimation.* Artech House, INC., 2005.

[3] N. H. Bingham and J. M. Fry. *Regression - Linear Models in Statistics.* Springer-Verlag, 2010.

[4] D. Dakopoulos and N.G. Bourbakis. Wearable obstacle avoidance electronic travel aids for blind: A survey. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 40(1):25 –35, jan. 2010.

[5] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient graph-based image segmentation. *International Journal of Computer Vision*, 59:2004, 2004.

[6] M. A. Fischler and R. 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.

[7] J.L. Gonzalez-Mora, A. Rodriguez-Hernandez, E. Burunat, F. Martin, and M.A. Castellano. Seeing the world by hearing: Virtual acoustic space (vas) a new space perception system for blind people. In *Information and Communication Technologies, 2006. ICTTA '06. 2nd*, volume 1, pages 837 –842, 0-0 2006.

[8] K. Khoshelham. Accuracy analysis of Kinect depth data. In *ISPRS Workshop Laser Scanning*, volume XXXVIII, pages 133–138, Aug. 2011.

[9] Micron, 8000 S. Federal Way, P.O. Box 6, Boise, ID 83707-0006, United States. *MT9M112 CMOS Image Sensor System-on-Chip*, 2005.

[10] J. Schares, L. Hoegner, and Stilla U. Geometrische Untersuchung zur Tiefengenauigkeit des Kinect-Sensorsystems. In *Publikationen der Deutschen Gesellschaft fr Photogrammetrie, Fernerkundung und Geoinformation (DGPF) e.V.*, volume 21, pages 372–380, 2012.

[11] J. Smisek, M. Jancosek, and T. Pajdla. 3d with Kinect. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 1154–1160, Nov. 2011.

[12] R. Tsai. A versatile camera calibration technique for high-accuracy 3d machine vision metrology using off-the-shelf tv cameras and lenses. *Robotics and Automation, IEEE Journal of*, 3(4):323 –344, august 1987.

[13] Z. Zhang. A flexible new technique for camera calibration. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(11):1330 – 1334, nov 2000.