Institute for Computer Science XVII
Robotics

Bachelor's thesis

High-Precision Calibration of a Stereo Camera-Laser Scanner Rig

Jonas Günther

January 2025

First reviewer: Prof. Dr. Andreas Nüchter

Zusammenfassung

In der vorliegenden Arbeit wird ein Stereo-Kamera-Laserscanner Rig kalibriert um eine hochgenaue Einfärbung von LiDAR-Punktwolken zu ermöglichen. Hierfür werden zuerst die einzelnen Kameras individuell kalibriert und darauf folgend eine Stereokalibrierung des Kamerasystems durchgeführt. Diese wird verwendet, um eine 3D-Rekonstruktion der Aufnahmen zu erstellen, die dann mittels ICP mit der LiDAR-Punktwolke registriert wird. Abschließend wird die LiDAR-Punktwolke mittels der Fotofarbwerte eingefärbt. Auf diese Weise konnte die Punktwolke auf eine Tiefe von drei bis acht Meter bis auf wenige Zentimeter genau eingefärbt werden und so eine genaue Farbwiedergabe ermöglicht.

Abstract

In this thesis, a stereo camera laser scanner rig is calibrated to enable high-precision coloring of LiDAR point clouds. For this purpose, the individual cameras are first calibrated individually and then a stereo calibration of the camera system is performed. This is used to create a 3D reconstruction of the images, which is then registered with the LiDAR point cloud using ICP. Finally, the LiDAR point cloud is colored using the photo color values. In this way, it was possible to color the point cloud to a depth of three to eight meters with an accuracy of just a few centimeters, thus enabling accurate color reproduction.

Contents

Chapter 1

Introduction

1.1 Task Definition

This work aims to calibrate a stereo-camera system and a LiDAR, delivering information about their relative pose. This is achieved by 3D-reconstruction of calibrated image pairs and subsequent matching of resulting point clouds with LiDAR measurements. To verify the results, coloring of the rotated LiDAR point cloud is done.

1.2 Work Structure

The theoretical principals of this work are described in chapter 2, covering calibration of single cameras, as well as stereo calibration and finally point cloud matching. In chapter 3 the experimental setup is explained, results are displayed and discussed.

Chapter 2

Theoretical Background

2.1 Camera Calibration

Camera properties can be split into two factors: *intrinsics* and *extrinsics*. While the instrinsic calibration aims at identifying physical properties of a camera, such as distortion, extrinsic calibration delivers information regarding the cameras pose w.r.t. the world coordinate system.

2.1.1 Intrinsics

The so-called *pin-hole camera model* is used to describe the imaging behavior of a camera. It "*consists of the image plane $\mathcal{Q}$ and the centre of projection (COP)*"[6]. All rays pass through the COP. Between $\mathcal{Q}$ and the COP lies the focal length f, describing the distance between $\mathcal{Q}$ and the focal plane $\mathcal{F}$ which contains the center of projection. Since all projected points are inverted as they pass through the COP, most of the times a simplified model is used. Preventing the values' inversion and simplifying the geometrical model as well as computation, $\mathcal{Q}$ is placed in front of the center of projection (see Fig.2.1). As shown in Fig. 2.1, a 3-dimensional point $\boldsymbol{P}_w$ is projected into the COP (here $\mathcal{F}_]$), resulting in a corresponding pixel $\boldsymbol{p_w}$ on the image plane. This is mathematically described as:

$$s\boldsymbol{p_w} = \boldsymbol{A}[\boldsymbol{R}|\boldsymbol{t}]\boldsymbol{P_w} \tag{2.1}$$

with $\boldsymbol{A}$ being the camera's intrinsic, $\boldsymbol{R}$ the rotational and $\boldsymbol{t}$ the translational matrix. Whilst $\boldsymbol{R}$ and $\boldsymbol{t}$ are describing the transformation of the 3-dimensional world coordinates to the 3-dimensional camera coordinates, $\boldsymbol{A}$ converts the resulting 3D-point into 2D pixel coordinates: [1]

$$\boldsymbol{A} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

with $\boldsymbol{f}$ being the camera's focal length and $\boldsymbol{c}$ the principal point, each in x and y.

As the OpenCV standard library is used in this work, three types of distortion are taken into account: *radial* (k_i) (see Fig.2.2), *tangential* (p_i) (see Fig.2.3) and *thin prism* (s_i) distortion[1].
OpenCV accounts for these as follows [1]:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} f_x x'' + c_x \\ f_y y'' + c_y \end{bmatrix} \tag{2.3}$$

with $\boldsymbol{u}$ and $\boldsymbol{v}$ being the x and y pixel-coordinates respectively and:

$$\begin{bmatrix} x'' \\ v'' \end{bmatrix} = \begin{bmatrix} x' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + 2p_1 x' y' + p_2(r^2+2x'2) + s_1 r^2 + s_2 r^4 \\ y' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + p_1(r^2+2y'^2) + 2p_2 x' y' + s_3 r^2 + s_4 r^4 \end{bmatrix} \tag{2.4}$$

where

$$r^2 = x'^2 + y'^2 \tag{2.5}$$

and

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} X_c/Z_c \\ Y_c/Z_c \end{bmatrix} \tag{2.6}$$

if $Z_c \neq 0$ and with X_c, Y_c and Z_c being the point's coordinates in the camera coordinate system.

2.1.2 Extrinsic

Since generally, the camera coordinate frame does not match the world coordinate frame, coordinates have to be rotated and translated in order to achieve affine transformations.

Since homogeneous coordinates allow for linear affine transformations, a 4x4 projection matrix is used[1]:

$$\begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \tag{2.7}$$

projecting a homogeneous point P_w of the world coordinate frame onto P_c of the camera coordinate frame:

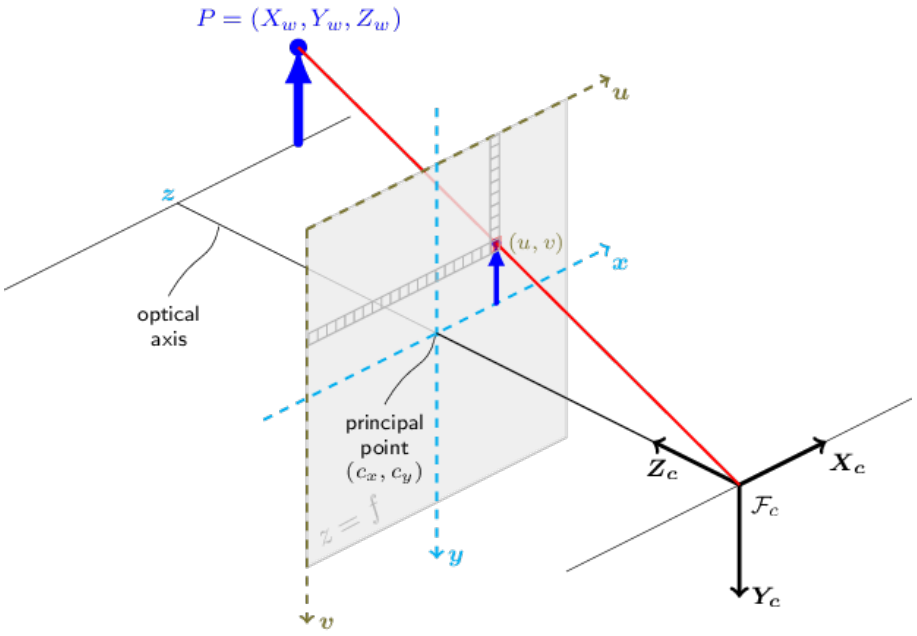

Figure 2.1: Pinhole-Camera-Model; Courtesy [1]

$$P_c = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{2.8}$$

with R encoding three Euler angles ω, ϕ, κ and t the translation in the three axes [6].

2.1.3 Calibration

Zhang's Method [12] is used for calibration. The goal is to determine the unknown parameters of (2.2) and (2.7). The following equations are closely related to [10] and [9].

(2.1) is rewritten as[1]

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2.9}$$

[1] In a more general approach, f_x is c and f_y is $c(1+m)$, with c being the principal distance and m a scaling factor. Furthermore a matrix entry cs exists, with s being a sheer factor. As no scaling is assumed and distortions are already considered in 2.1.2, $s = 0$ and $m = 0$ is assumed.

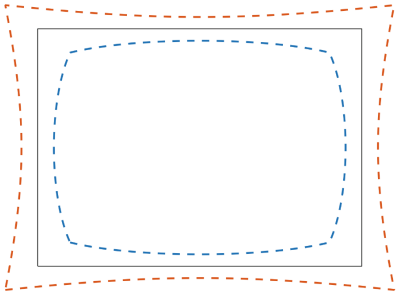

Figure 2.2: Barrel Distortion: Red positive, blue negative; Courtesy [6]

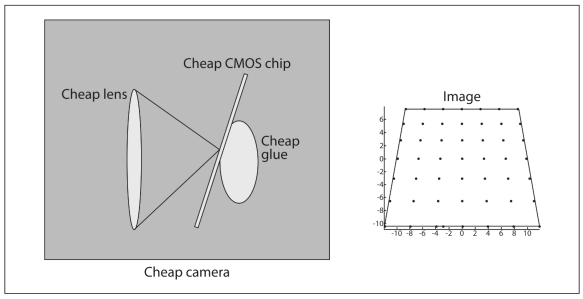

Figure 2.3: Tangential distortion caused by a non-parallel sensor chip; Courtesy [4]

Zhang's method utilizes multiple different views of calibration patterns with known properties. One such property is, that all points lie in one plane. Thus $Z = 0$ and it can be deleted from the vector, allowing for the deletion of the row $(r_{13}, r_{23}, r_{33})^T$:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ r_{31} & r_{32} & t_z \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \tag{2.10}$$

Thus the resulting estimation can be reduced from a 3x4 projection matrix to a 3x3 homography. This results in an equation for an amount of I points per image

$$\begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = \mathbf{H} \begin{bmatrix} X_i \\ Y_i \\ 1 \end{bmatrix} \tag{2.11}$$

with the homography $\mathbf{H}$

$$\mathbf{H} = \boldsymbol{A} \begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ r_{31} & r_{32} & t_z \end{bmatrix}. \tag{2.12}$$

$(r_{11}, r_{21}, r_{31})^T$ and $(r_{12}, r_{22}, r_{32})^T$ will be referred to as $\mathbf{r_1}$ and $\mathbf{r_2}$ respectively.
Simplifying the following equations, $\mathbf{H}$ (and thus (2.11)) will be rearranged:

$$\mathbf{H} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} = \begin{bmatrix} \mathbf{A}^T \\ \mathbf{B}^T \\ \mathbf{C}^T \end{bmatrix}. \tag{2.13}$$

Transforming (2.11) from homogeneous coordinates to euclidean, x_i and y_i are defined as:

$$x_i = \frac{\mathbf{A}^T\mathbf{X}_i}{\mathbf{C}^T\mathbf{X}_i}, y_i = \frac{\mathbf{B}^T\mathbf{X}_i}{\mathbf{C}^T\mathbf{X}_i} \tag{2.14}$$

From this, a system of linear equations follows:

$$x_i\mathbf{C}^T\mathbf{X}_i - \mathbf{A}^T\mathbf{X}_i = 0 \tag{2.15}$$
$$y_i\mathbf{C}^T\mathbf{X}_i - \mathbf{B}^T\mathbf{X}_i = 0 \tag{2.16}$$

To solve this system of linear equations, a singular value decomposition (SVD) is applied. Thus (2.15)/(2.16) are rewritten as:

$$\mathbf{a}_{x_i}^T\mathbf{h} = 0 \tag{2.17}$$
$$\mathbf{a}_{y_i}^T\mathbf{h} = 0 \tag{2.18}$$

with

$$\mathbf{a}_{x_i}^T = (-\mathbf{X}_i, \mathbf{0}^T, x_i\mathbf{X}_i) \tag{2.19}$$
$$\mathbf{a}_{y_i}^T = (\mathbf{0}^T, -\mathbf{X}_i, y_i\mathbf{X}_i) \tag{2.20}$$
$$\mathbf{h} = vec(H^T). \tag{2.21}$$

This allows for SVD which has the form

$$\mathbf{M}\mathbf{h} = 0 \tag{2.22}$$

where

$$\mathbf{M} = \begin{bmatrix} \mathbf{a}_{x_1}^T \\ \mathbf{a}_{y_1}^T \\ \dots \\ \mathbf{a}_{x_i}^T \\ \mathbf{a}_{y_i}^T \\ \dots \\ \mathbf{a}_{x_I}^T \\ \mathbf{a}_{y_I}^T \end{bmatrix}. \tag{2.23}$$

Using this form, multiple points can be used for estimating $\mathbf{H}$.

In reality, (2.22) is only close to 0, such that (2.22) equals to an ω close to 0. To achieve the best result, $\mathbf{h}$ is wanted, "*such that it minimizes*" [10]:

$$\begin{aligned} \Omega &= \omega^T \omega \\ \Rightarrow \widehat{\mathbf{h}} &= \arg\min_{\mathbf{h}} \omega^T \omega \\ &= \arg\min_{\mathbf{h}} \mathbf{h}^T \mathbf{M}^T \mathbf{M} \mathbf{h} \end{aligned} \tag{2.24}$$

with

$$||H||_2 = \sum_{ij} h_{ij}^2 = ||\mathbf{h}|| = 1 \tag{2.25}$$

To get the optimal solution for $\mathbf{h}$, $\mathbf{h} = \mathbf{v_9}$ is retrieved from the SVD, analogue to [10]

$$\underset{2I\times 9}{M} = \underset{2I\times 99\times 99\times 9}{U\ S\ V^T} = \sum_{i=1}^{9} s_i \mathbf{u_i} {\mathbf{v_i}}^T \tag{2.26}$$

as $\mathbf{v}_9$ is "*the singular vector belonging to the smallest singular value* s_9" [10] and as such minimizes Ω.

As the homography $\mathbf{H}$ is now estimated, the next step is to compute $\mathbf{A}$ from (2.12). For this purpose, a series of constraints are set up.

As $\mathbf{A}$ is invertible, (2.12) can be rewritten as

$$[\mathbf{r}_1, \mathbf{r}_2, \mathbf{t}] = \mathbf{A}^{-1}[\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3] \tag{2.27}$$

from which follows:

$$\begin{aligned} \mathbf{r_1} &= \mathbf{A}^{-1}\mathbf{h_1} \\ \mathbf{r_2} &= \mathbf{A}^{-1}\mathbf{h_2} \end{aligned} \tag{2.28}$$

"*As* r_1, r_2, r_3 *form an orthonormal basis*":[9]

$$\mathbf{r}_1^T \mathbf{r}_2 = 0 \tag{2.29}$$

$$||\mathbf{r}_1|| = ||\mathbf{r}_2|| = 1 \tag{2.30}$$

Putting (2.28) into (2.29):

$${\mathbf{h_1}}^T \mathbf{A}^{-T} \mathbf{A}^{-1} \mathbf{h_2} = 0 \tag{2.31}$$

And from constraint (2.30) follows:

$$\begin{aligned} &\mathbf{h_1}^T \mathbf{A}^{-T} \mathbf{A}^{-1} \mathbf{h_1} = \mathbf{h_2}^T \mathbf{A}^{-T} \mathbf{A}^{-1} \mathbf{h_2} \\ \iff &\mathbf{h_1}^T \mathbf{A}^{-T} \mathbf{A}^{-1} \mathbf{h_1} - \mathbf{h_2}^T \mathbf{A}^{-T} \mathbf{A}^{-1} \mathbf{h_2} = 0 \end{aligned} \tag{2.32}$$

For the following matrix decomposition a "*symmetric and positive definite*"[9] matrix $\mathbf{B}$ is defined such that $\mathbf{A}$ can be derived by $\mathbf{B}$:

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \mathbf{A}^{-T} \mathbf{A}^{-1} \tag{2.33}$$

As $\mathbf{A}$ is a triangular matrix, a vector $\mathbf{b} = [b_{11}, b_{12}, b_{13}, b_{22}, b_{23}, b_{33}]$ is defined, covering $\mathbf{A}$'s 6 degrees of freedom. Similar to (2.22), a system of linear equations

$$\underset{2\times 6}{\mathbf{V}}\,\mathbf{b} = 0 \tag{2.34}$$

using the constraints (2.29) and (2.30)

$$\mathbf{v_{12}}^T \mathbf{b} = 0 \tag{2.35}$$

$$\mathbf{v_{11}}^T \mathbf{b} - \mathbf{v_{22}}^T \mathbf{b} = 0 \tag{2.36}$$

is established.
Following (2.35) and (2.36), $\mathbf{V}$ is defined as:

$$\mathbf{V} = \begin{bmatrix} \mathbf{v}_{12}^T \\ \mathbf{v}_{11}^T - \mathbf{v}_{22}^T \end{bmatrix} \text{ with } \mathbf{v}_{ij} = \begin{bmatrix} h_{1i}h_{1j} \\ h_{1i}h_{2j} + h_{2i}h_{1j} \\ h_{3i}h_{1j} + h_{1i}h_{3j} \\ h_{2i}h_{2j} \\ h_{3i}h_{2j} + h_{2i}h_{3j} \\ h_{3i}h_{3j} \end{bmatrix} \tag{2.37}$$

For every image, one equation of the form

$$\begin{bmatrix} \mathbf{v}_{12}^T \\ \mathbf{v}_{11}^T - \mathbf{v}_{22}^T \end{bmatrix} \mathbf{b} = 0 \tag{2.38}$$

is obtained. As $\mathbf{A}$ is of dimension 6×6 and $\mathbf{V}$ is of 2×6, at least 3 pictures are required to compute $\mathbf{A}$, adding the respective homographies to $\mathbf{V}$ in the above equation.

This system of linear equations is then solved similar to (2.22) using a SVD. Since measurements are containing noise, $\mathbf{b}$ needs to be minimized as follows

$$\mathbf{b}^* = \arg\min_{\mathbf{b}} ||\mathbf{V}\mathbf{b}\text{with } ||b|| = 1 \tag{2.39}$$

As the final step, using the Cholesky decomposition knowing $\mathbf{B}$, $\mathbf{A}$ derives from the form

$$chol(B) = \mathbf{K}\mathbf{K}^T \tag{2.40}$$

$$\text{with } \mathbf{K} = \mathbf{A}^{-T} \tag{2.41}$$

Estimating $\mathbf{A}$ and $\mathbf{H}$, (2.27) can be solved with (2.29), (2.30) and

$$\mathbf{t} = \mathbf{A}^{-1}\mathbf{h}_3 \tag{2.42}$$

2.2 Stereo Calibration

Calibrating the Stereo-LiDAR-Rig not only the knowledge about the relative pose between stereo-system and LiDAR is necessary, but also the relative pose between both cameras (refer to chapter 2.3). For this a stereo calibration is necessary, calculating a rotation matrix R and translation vector $\mathbf{t}$.

2.2.1 Epipolar Geometry

Determine the relative pose the projection of object points onto the respective image planes are considered. Given a point $\mathbf{m}$ in the first camera's image, its homologous point $\mathbf{m'}$ must lie on a straight line on the second image (see 2.4), since the information about $\mathbf{m}$'s depth is lost. This is called the *epipolar constraint*[6]. The resulting line in the second image is then called the *epipolar line*[2]. Geometrically, the epipolar line is the intersection of the respective image plane with the epipolar plane, which is "determined by m, C and C'"[6], with C and C' being the respective *center of projection*. The intersection of CC' with the image planes are called *epipoles*. All epilines are running through this projection of the other camera's center of projection.

Defining now the relation between a point M_l of a point in the left camera coordinate system to a corresponding point M_r in the right camera coordinate system (so in 3-dimensional space), a few assumptions are made[4]:

- As origin, C is defined.[3]
- The location of the other COP will be denoted as $\mathbf{t} = C' - C$, giving only the direction.

[2] Sometimes also referred to as *epiline*[4]

[3] C' can also be taken as origin, the following equations would have to be adjusted accordingly.

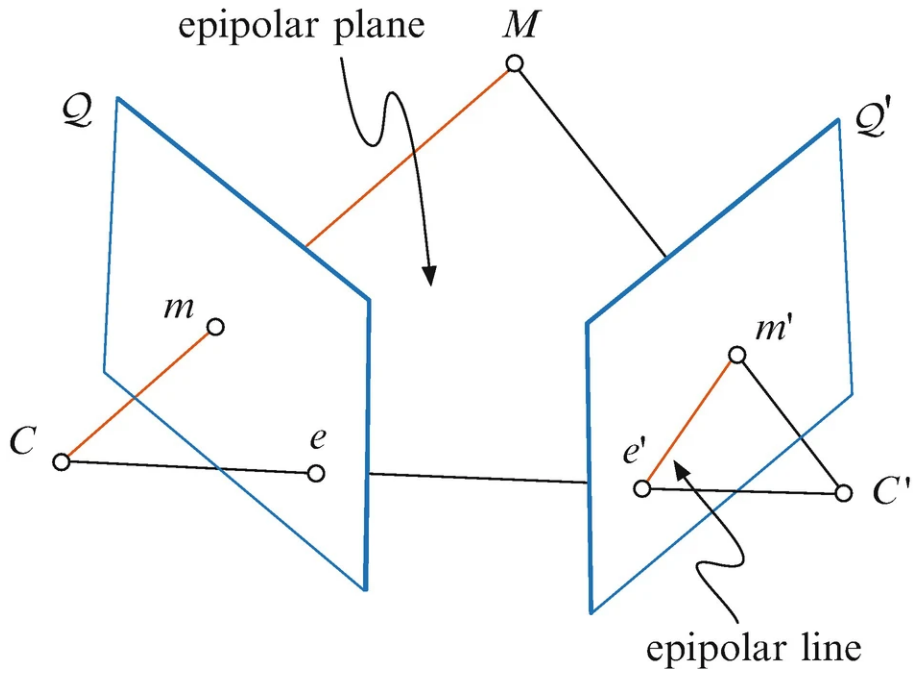

Figure 2.4: Epipolar Geometry; Courtesy [6]

Based on those, the relation between M_l and M_r follows:

$$M_r = \mathbf{R}(M_l - \mathbf{t}) \tag{2.43}$$

Utilizing the epiplane one constraint can be derived because the following equation applies for "*all points* $\mathbf{x}$ *on a plane with normal vector* $\mathbf{n}$ *and passing through point* $\mathbf{a}$"[4]:

$$(\mathbf{x} - \mathbf{a}) \cdot \mathbf{n} = 0 \tag{2.44}$$

Implementing this for this use case, as M_l and $\mathbf{t}$ lie in the epipolar plane and thus their cross product is orthogonal to each:

$$(M_l - \mathbf{t}^T)(\mathbf{t} \times M_l) = 0 \tag{2.45}$$

Now rearranging (2.43) with $\mathbf{R}^T = \mathbf{R}^{-1}$

$$(M_l - \mathbf{t})^T = \mathbf{R}^T M_r \tag{2.46}$$

and applying it to (2.45), results in

$$(\mathbf{R}^T M_r)^T (\mathbf{t} \times M_l) = 0. \tag{2.47}$$

Rewriting the cross product $\mathbf{t} \times M_l$ in matrix form

$$\mathbf{t} \times M_l = \mathbf{S} M_l \text{ with } \mathbf{S} = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \tag{2.48}$$

allows for the a concise relation between two corresponding points in the respective camera coordinate system

$$M_r^T \mathbf{R}\mathbf{S} M_l = 0. \tag{2.49}$$

RS is denoted as **E**, the essential matrix, capturing the geometrical information between two calibrated cameras.

To retrieve the relation between two points in the normalized camera coordinate system (2D-plane with physical values e.g. m, cm), M_l and M_r can be transformed analogously to (2.6) into m_l and m_r, and (2.49) can be adjusted:

$$m_r^T \mathbf{R}\mathbf{S} m_l = 0. \tag{2.50}$$

Finally relating the points in the image sensor (2D-pixel-coordinates), using

$${}^p m = \mathbf{A} m \tag{2.51}$$

with ${}^p m$ being the point in pixel coordinates and **A** the respective camera matrix, (2.49) is once more adjusted:

$${}^p m_r^T (\mathbf{A}_r^{-1})^T \mathbf{R}\mathbf{S}\mathbf{A}_l^{-1} {}^p m_l = 0. \tag{2.52}$$

$(\mathbf{A}_r^{-1})^T \mathbf{R}\mathbf{S}\mathbf{A}_l^{-1}$ is denoted as **F**, the fundamental matrix, expanding the essential matrix **E** by the camera matrices and thus the physical properties of the cameras.

With the relation between two corresponding points established, [4] proposes the following to compute **R** and **t**.

An object point M in the world coordinate system is related to M_l (analogously for M_r) in the camera coordinate system via

$$M_l = \mathbf{R}_l M + \mathbf{t}_l. \tag{2.53}$$

Together with

$$M_l = \mathbf{R}^T(M_r - \mathbf{t}) \tag{2.54}$$

where $\mathbf{R}$ and $\mathbf{t}$ describe the rotation and translation between two cameras, the following solutions for $\mathbf{R}$ and $\mathbf{t}$ arise:

$$\mathbf{R} = \mathbf{R}_r(\mathbf{R}_l)^T \tag{2.55}$$

$$\mathbf{t} = \mathbf{t}_r - \mathbf{R}\mathbf{t}_l \tag{2.56}$$

As measurements are noisy,[4] proposes the use of multiple points (and images) to compute the median values of $\mathbf{R}$ and $\mathbf{t}$ and then use "*a robust Levenberg- Marquardt iterative algorithm to find the (local) minimum*"[4], to optimize the rotation and translation.

2.3 Pointcloud Calibration

In the calibration of the Stereo-LiDAR-Rig the final step is to generate a point cloud from the stereo images to match it with the LiDAR-measurements. To achieve this 3D-reconstruction of the image pairs, the stereo normal case and furthermore matching algorithms are used.

2.3.1 Point Matching

Since 3D-reconstruction from low-textured monochromatic calibration patterns is prone to errors, from this point onward only scenes without available information are considered and therefore matching algorithms are required. This can be done in multiple ways. In chapter 3 the *Semi-Global-Block-Matching algorithm* will be used but depending on the field of application different algorithms might be applied.

To simplify the computing effort of these algorithms, the stereo images can be transformed into the *stereo normal case* (see Fig. 2.5).

The stereo normal case is geometrically described as: "*Both cameras are ideal and identical, the viewing directions are parallel and orthogonal to the base vector, and the x'- and x''-axes are parallel to the basis.*"[11]

following[8] the mathematical terms can be described:

- Image planes are parallel and not rotated w.r.t. each other:

$$\mathbf{R}' = \mathbf{R}'' = \mathbf{I}_3 \tag{2.57}$$

- Offset only in x direction and y-parallaxes are zero for all points:

$$\mathbf{b} = \begin{bmatrix} B_x \\ 0 \\ 0 \end{bmatrix} \tag{2.58}$$

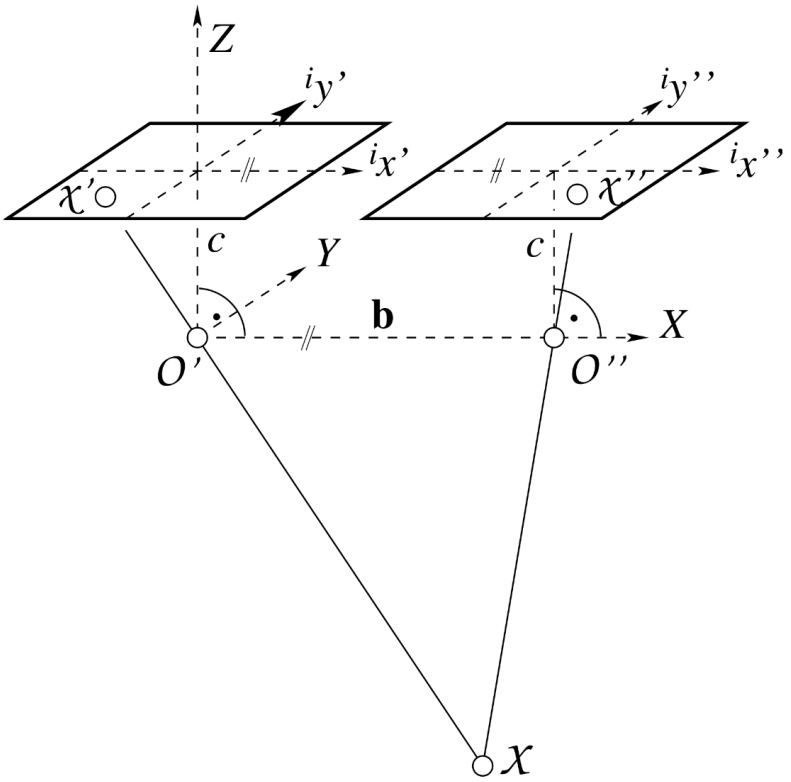

Figure 2.5: Stereo Normal Case; Courtesy [11]

- Accordingly the essential matrix is defined as:

$$\mathbf{E} = \mathbf{S}_b\mathbf{R}^T = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -B_x \\ 0 & B_x & 0 \end{bmatrix} \tag{2.59}$$

- And identical camera matrices are assumed[4]

$$\mathbf{A} \doteq \mathbf{A}' = \mathbf{A}'' = \begin{bmatrix} c & 0 & 0 \\ 0 & c & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ with } c \text{ being the principal point.} \tag{2.60}$$

Since the stereo normal case is barely achieved in reality, it has to be generated. For this, similar to 2.2, homographies can be used to describe the relation between the original point x and the rotated ${}^n x$ in the normal case[8]:

[4]The relationship between c and f is described as "*the assumed principal distance c, which may be approximated by the focal length*"[11] ($f \approx f_x \approx f_y$ is assumed).

$$ {}^{n}x' = {}^{n}\mathbf{H}'x' \qquad {}^{n}x'' = {}^{n}\mathbf{H}''x'' \tag{2.61} $$

Since

$$ x' = \mathbf{P}'X = \mathbf{A}'\mathbf{R}'[\mathbf{I}_3|\mathbf{X}_{\mathbf{O}'}]\mathbf{X} \tag{2.62} $$
$$ x'' = \mathbf{P}''X = \mathbf{A}''\mathbf{R}''[\mathbf{I}_3|\mathbf{X}_{\mathbf{O}''}]\mathbf{X} \tag{2.63} $$

A and **R** are defined, such that

$$ {}^{n}x' = {}^{n}\mathbf{P}'X = \mathbf{A}\mathbf{R}[\mathbf{I}_3|\mathbf{X}_{\mathbf{O}'}]\mathbf{X} \tag{2.64} $$
$$ {}^{n}x'' = {}^{n}\mathbf{P}''X = \mathbf{A}\mathbf{R}[\mathbf{I}_3|\mathbf{X}_{\mathbf{O}''}]\mathbf{X}, \tag{2.65} $$

Because only the camera and rotation matrices differ between (2.62)/(2.63) and (2.64)/(2.65) respectively, the following relation can be established:

$$ {}^{n}H' = \mathbf{A}\mathbf{R}\mathbf{R}'^{T}\mathbf{A}'^{-1} \tag{2.66} $$
$$ {}^{n}H'' = \mathbf{A}\mathbf{R}\mathbf{R}''^{T}\mathbf{A}''^{-1} \tag{2.67} $$

With the image planes now in stereo normal case, the search for corresponding points is reduced to 1D. Thus the deviation occurs only in x-direction and for all y-parallaxes $y_p = 0$ is true (w.r.t. 2.2.1: the epipoles lie at infinity and all epilines are parallel), the search for corresponding points is reduced to 1D.

2.3.2 3D Reconstruction

To reproject 2-dimensional points back to 3D, the information gathered in the previous sections and chapters can be used. Using the homographies (2.66) and (2.67), the image planes are brought into stereo normal case, allowing to compute the disparity d.[5] d is then defined as

$$ d = x' - x''. \tag{2.68} $$

Looking at 2.6´ it becomes clear, that the disparity is inversely proportional to the depth of a point. Triangulation[4] delivers

$$ Z = \frac{fT}{x' - x''} \tag{2.69} $$

[5]No distortion assumed/already taken into account, see (2.4).

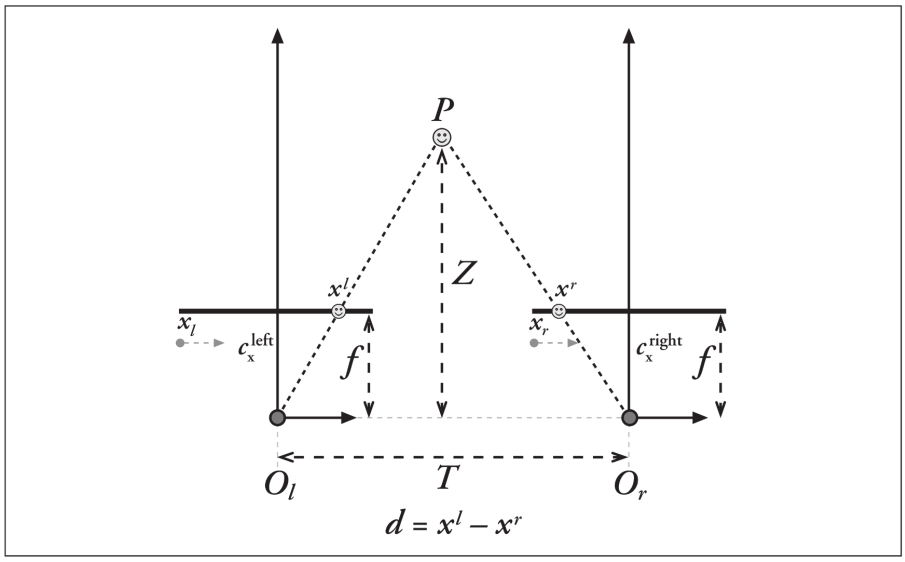

Figure 2.6: Stereo Normal Case used to compute disparity d; Courtesy [4]

To now reconstruct the 3D points, [4] introduces a reprojection matrix Q[6]

$$Q = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & f \\ 9 & 0 & \frac{-1}{T_x} & 0 \end{bmatrix} \tag{2.70}$$

Using Q, the 2-dimensional image points can now be projected to 3D:

$$Q \begin{bmatrix} x \\ y \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} \tag{2.71}$$

To finally obtain the 3D coordinates, the homogeneous coordinates have to be transformed back to euclidian ones via $\frac{X}{W}, \frac{Y}{W}$ and $\frac{Z}{W}$.

With this, a set of points $\mathbf{D_i}$ can be created.

2.3.3 ICP

The last step to calibrate the LiDAR with the stereo setup is to match the PCLs from LiDAR input and depth estimation from the stereo recording as described in the previous section. For

[6] [4]' matrix Q has lower right matrix entry $\frac{c_x - c'_x}{T_x}$ but states, that "*[i]f the principal rays intersect at infinity, then $c_x = c'_x$ and the term in the lower right corner is 0*". As the stereo normal case is computed in this work, the entry here is subsequently set to 0.

this, the *Iterative Closest Point* [2] algorithm is used. ICP computes for each point $\mathbf{M_i}$ of a set of points $\mathcal{M}$ the closest point $\mathbf{D_i}$ of a second set of points $\mathcal{D}$[6]:

$$min_{R,\mathbf{t}} \sum_{i=1}^{n} \|\mathbf{D}_i - (R\mathbf{M}_i + \mathbf{t})\|^2 \tag{2.72}$$

ICP then iteratively applies changes to the rotation matrix R and translation vector $\mathbf{t}$ as to minimize (2.72).

Chapter 3

Experiment and Evaluation

3.1 Experimental Setup

3.1.1 The Rig

The Stereo-Camera-LiDAR-Rig consists of two *U3-30C0CP Rev.2.2 iDSI* cameras and a *Livox Avia LiDAR* mounted on a handheld device as described in [3]. The cameras are equipped with a global shutter system, a resolution of 2.35MP/1936 x 1216p and a Sony IMX392LQR-C sensor. Additionally, both cameras "*are equipped with 4 mm lenses, which results in a [...] FoV of 77.3°×61.9°.*"[3]. The Livox Avia LiDAR has a detection range of up to 450m, range precision of 2cm at 20m with 80% reflectivity. The FoV is 70.4°×77.2° when using a non-repetitive scanning pattern, as done in this work. The data processing is done with ROS.[1] The computing is done offline and not in real-time. For this, the Rosbag-files had to be extracted, containing both cameras' images and the LiDAR measurements.

3.1.2 Camera Calibration

For the following calibration tasks, the software framework OpenCV[2] is used. It provides methods implementing most of the individual steps in chapter 2. Moreover, the fundamental methods will be mentioned in the following sections. The code can be found in the Appendix A.1/A.2.

For Zhang's method of camera calibration (see Sec. 2.1.3), a ChArUco board is used. It is a combination of the classical chessboard pattern and ArUco patterns[7]. Whilst chessboard calibration patterns are a simple method to identify key elements, it is prone to occlusion and has to be fully visible at any time. ChArUco benefits from the advantages of chessboard patterns, but adds ArUco identifiers for each corner allowing not only for a fixed orientation of the board observed (an information lost on chessboards, there is no "up" or "down"), but also works despite occlusions or partial recordings of the pattern, as seen in Fig. 3.3.

As discussed in Sec. 2.1.3, multiple images of various different angles were taken to compute the camera matrix of each camera. For Sec. 3.2, over 140 image pairs are used for calibration.

[1] https://www.ros.org/
[2] https://opencv.org/

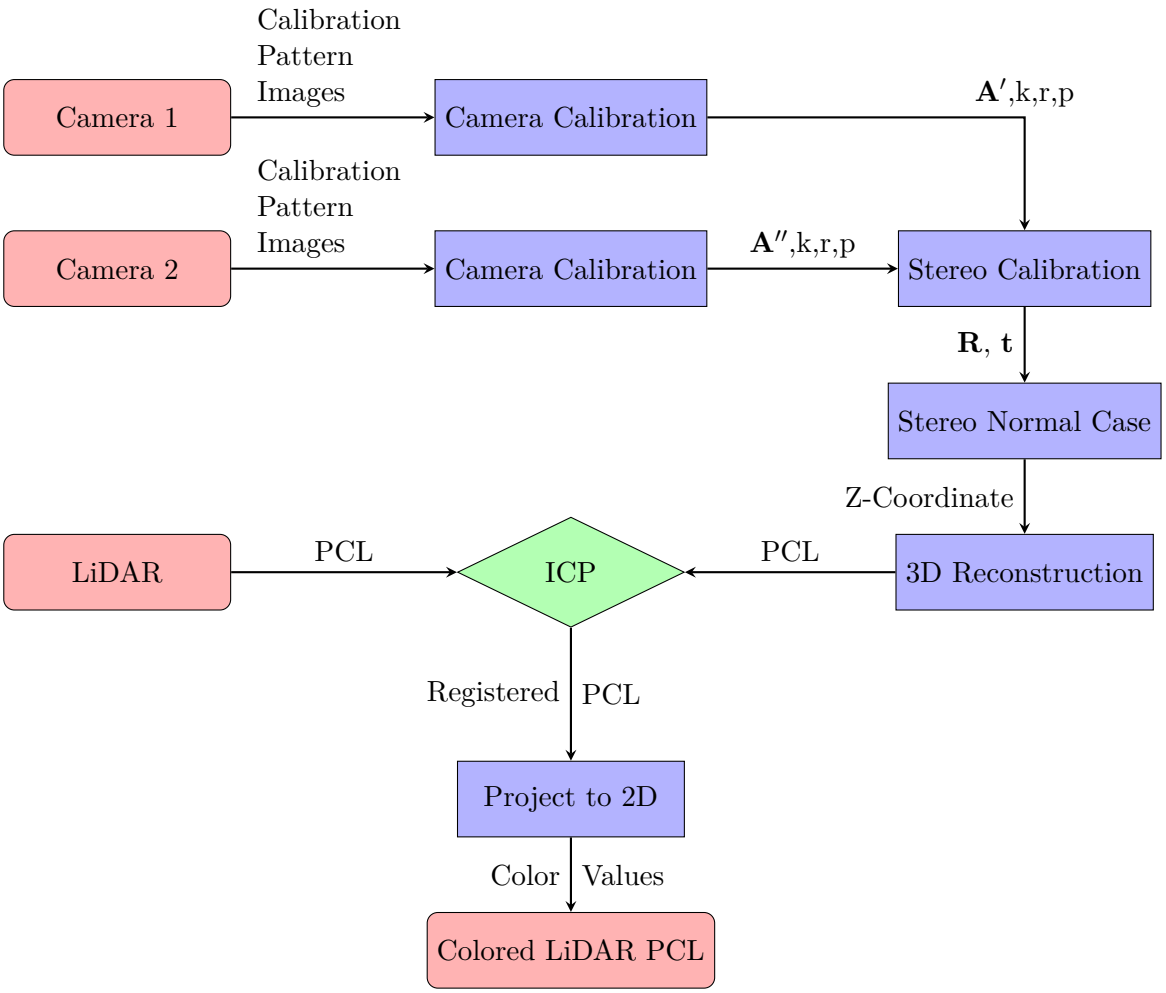

Figure 3.1: Simplified Workflow Overview

Using `detectBoard()` of the class `CharucoDetector`, pattern detection is done and using `matchImagePoints` of class `CharucoBoard`, the identified object points (3D) are matched with the detected image points (2D).

This is repeated for every image taken. `calibrateCamera` then combines all object and image points gathered and calculates the camera matrix as well as the distortion coefficients from (2.4), reducing the reprojection error via the Marquardt-Levenberg algorithm similar to Sec. 2.2.

3.1.3 Stereo Calibration

With each camera matrix calibrated in the previous step, the already gathered image and object points are both used to compute the relative pose via `stereoCalibrate`. Analogue to `cameraCalibrate`, the reprojection error is computed using a Levenberg-Marquardt algorithm, minimizing the error of $\mathbf{R}$ and $\mathbf{t}$.

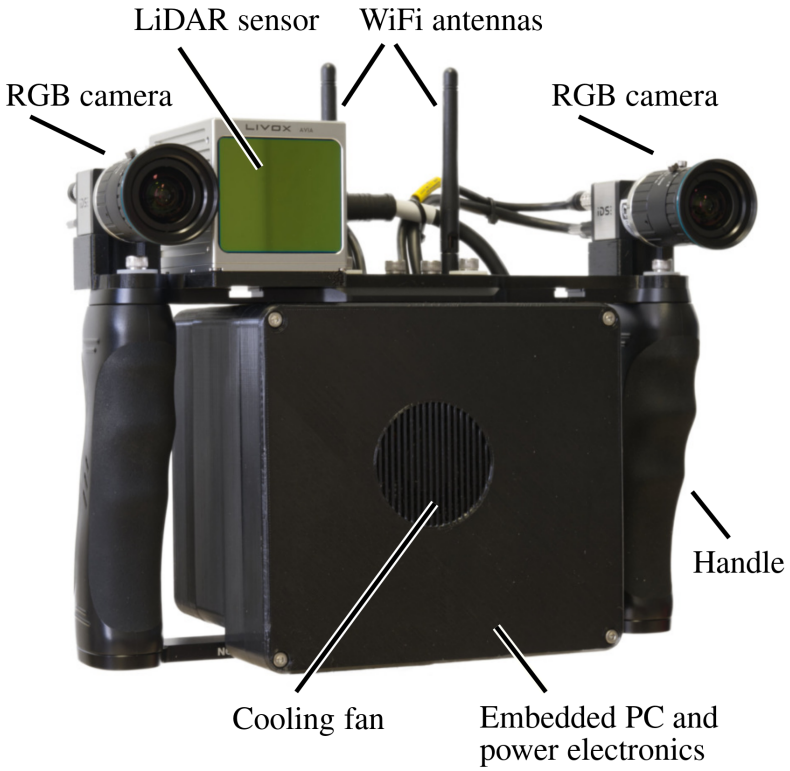

Figure 3.2: The Rig. Courtesy [3]

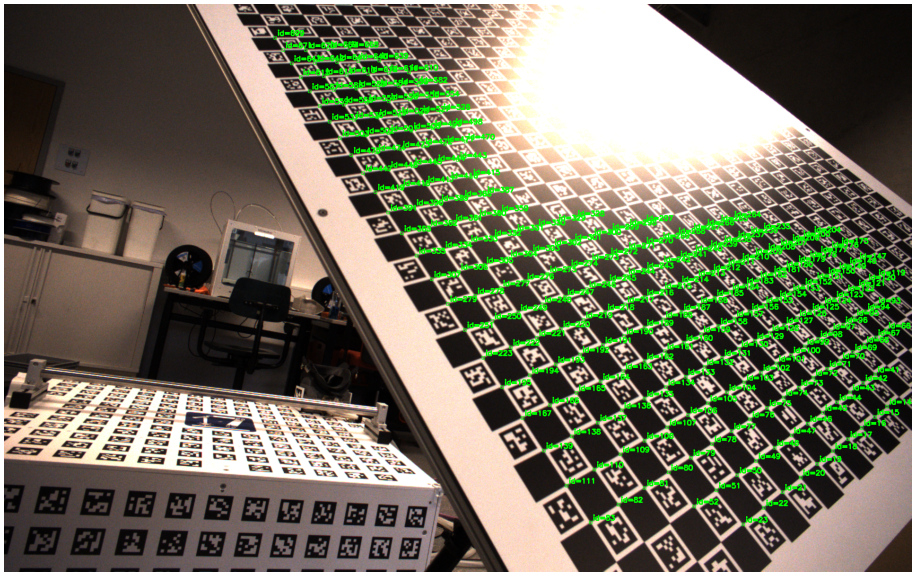

Figure 3.3: The Charuco pattern occluded by lighting; Green labels identify each corner uniquely.

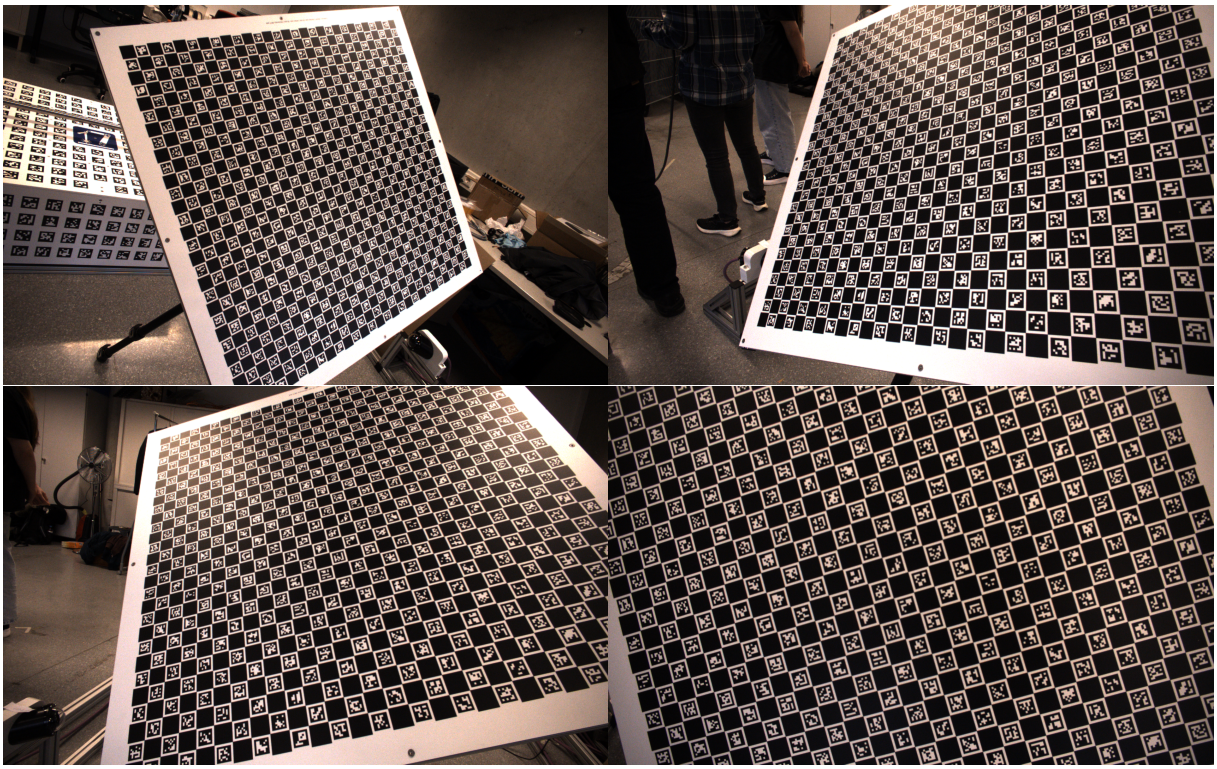

Figure 3.4: Different poses used for calibration.

3.1.4 PCL Calibration

With the relative pose of the stereo setup computed, the 3D reconstruction can be done.

As discussed in Section 2.2.1, the images have to be brought to stereo normal case, transforming both image planes to one common viewing direction, as well as assigning one common camera matrix, among others. As the real camera images are distorted, and both cameras need to be "*ideal*"[11], OpenCV first applies the distortion factors retrieved in 3.1.2 to the camera matrices via `initUndistortRectifyMap`, adjusting each camera matrix to achieve an ideal mapping behaviour, as well as transforming the images to stereo normal case (see Fig. 3.5).

In the next step, depth estimation is done. *Semi-Global Block Matching*, as provided by OpenCV, is used for this. It is based on the H. Hirschmüller SGM algorithm. `StereoSGBM` returns a disparity map, displaying the estimated depth of each point/block.

Since the color values need to differ to reliably identify correspondences, the depth estimation relies on highly textured/non-monochromatic samples. Otherwise working with a calibration pattern will not succeed, as the resulting depth map is highly distorted. Also surfaces with high reflectivity or glass may cause flaws, as demonstrated in Fig. 3.6. Furthermore, the Livox Avia does not register reliably points closer than roughly 3m. As for the final test scene, a playground is chosen (see Fig. 3.7). Having a highly textured climbing wall, many different, unique structures and lacking a high amount of small-scaled elements, which may cause distortion, it provides an ideal sample. The scaling of the structures also enables scanning from a distance greater than

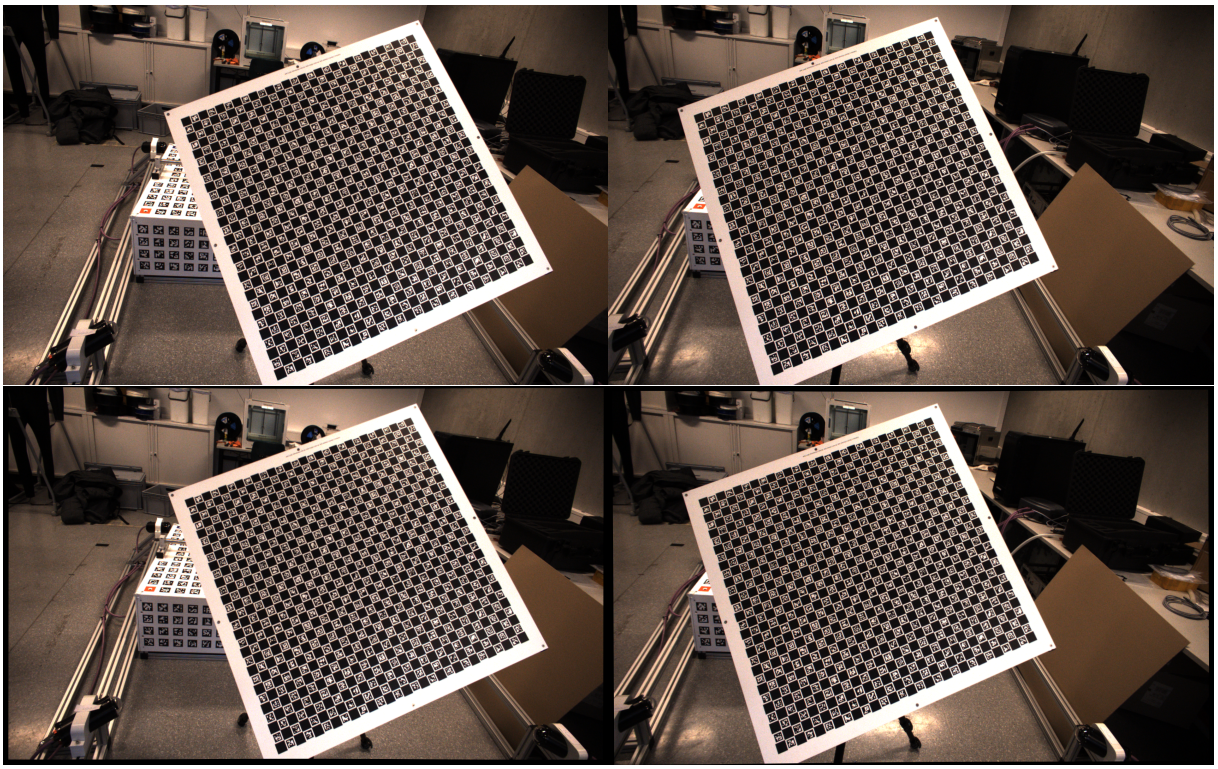

Figure 3.5: First row: original. Second Row: rectified. Only a slight difference can be determined, as the cameras seem to provide a good inherent calibration.

3m. The rig setup is not moved during recording to allow for LiDAR measurements which do not need to be registered afterwards to compensate for movement, preventing unnecessary errors. Furthermore choosing cloudy weather for recording is recommended since uniform light exposure provides better results.

After calculating the 3D coordinates via `reprojectImageTo3D` and using the reprojection matrix Q from (2.70), the point cloud can be created.

The LiDAR scans are simply combined, since the static setup makes sure that no further processing of each scan is needed.

For displaying the PCL's and running ICP, 3DTK - The 3D Toolkit [3] is used. It has to be noted, that, differing from the conventions of ROS and OpenCV, 3DTK works in a left-handed coordinate system (x-right, y-up, z-forward). ROS works in a right handed-system with x-forward, y-left and z-up[4], while OpenCV uses the following convention x-right, y-down and z-forward, as the origin of the image coordinates lie on the upper left corner of each image [4]. To account for this, both LiDAR(ROS) and stereo(OpenCV) point clouds have to be converted to the left-hand coordinate system. While for OpenCV, simply inverting the y-axis is sufficient, values obtained via ROS need to be rotated. This can be done via interchanging the coordinates

[3] https://slam6d.sourceforge.io/

[4] ROS Enhancement Proposal 103: https://www.ros.org/reps/rep-0103.html

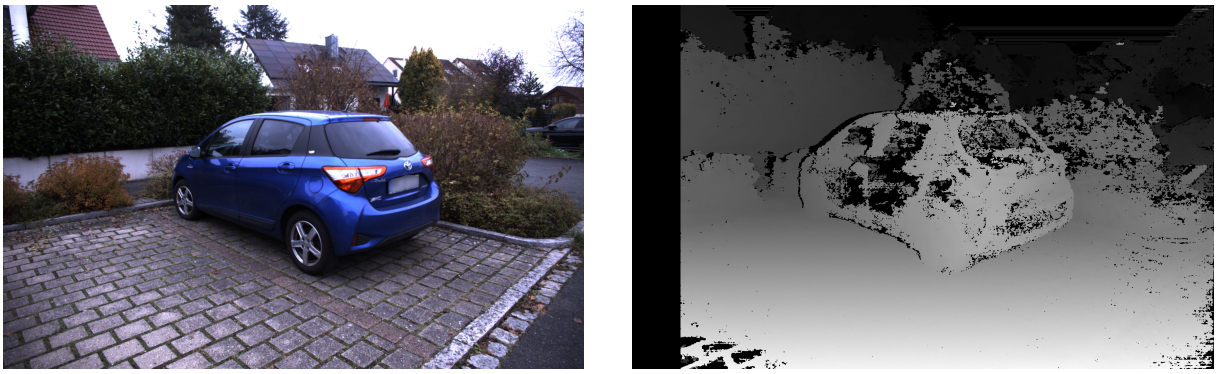

Figure 3.6: Depth estimation using a car as test scene. The reflection on the side front causes a wrong depth and glass leads to partial recognition of the surface and objects behind it, leading to high distortion.

Figure 3.7: The test scene.

Figure 3.8: The test setup.

as following:

$$\begin{bmatrix} x_L \\ y_L \\ z_L \end{bmatrix} = \begin{bmatrix} -y_R \\ z_R \\ x_R \end{bmatrix} \tag{3.1}$$

Alternatively this can be done via rotation:

$$\mathbf{p}_L = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \mathbf{p}_R \tag{3.2}$$

When both PCL's are transformed to the same coordinate system, 3DTK provides the `slam6D` function, to apply ICP and register points. It iteratively matches two sequential scans

Figure 3.9: Sample Picture from the Left Camera

until a set threshold is met (standard ϵ=1.0000000000000001$*10^{-5}$) or until a set amount of iterations is reached. Furthermore it allows for octree based point reduction (for this particular test scene, 5cm as voxel size is used), speeding up the process. And lastly a maximum distance is set with 10cm up to which point-to-point correspondence can be assumed.

3.1.5 PCL Coloring

To simplify evaluation of the results, the LiDAR point cloud is colored, using the RGB values of one of the rectified images. To achieve this, the rotation and translation computed by the ICP-algorithm in the previous section is applied to the LiDAR-PCL.[5] Via OpenCV's `projectPoints` and the rotation, translation and camera matrix obtained in chapter section 2.1.3 as well as the distortion coefficients computed in 3.1.2. This projects the 3D point cloud onto a 2D plane, similar to the real camera. Then, iterating over each pixel of the raw image matrix (non-rectified, in this case 1920×1200p), every point is colored accordingly. Since the LiDAR has a different FoV than the camera, outliers are colored black as 3DTK background color is black. Once again it is important to point out differing conventions: OpenCV works with BGR color order. To work with 3DTK, this has to be changed to RGB when saving the color values.

3.2 Results

To not rely solely on the subjective impression of the coloring, physical context and limitations have to be clear. [5] provides the means to calculate the quality of the resulting 3D points. To calculate the error in Z-direction/depth, the following applies:

[5] It is important to note, that the LiDAR-PCL is rotated on the stereo-PCL, not the other way round!

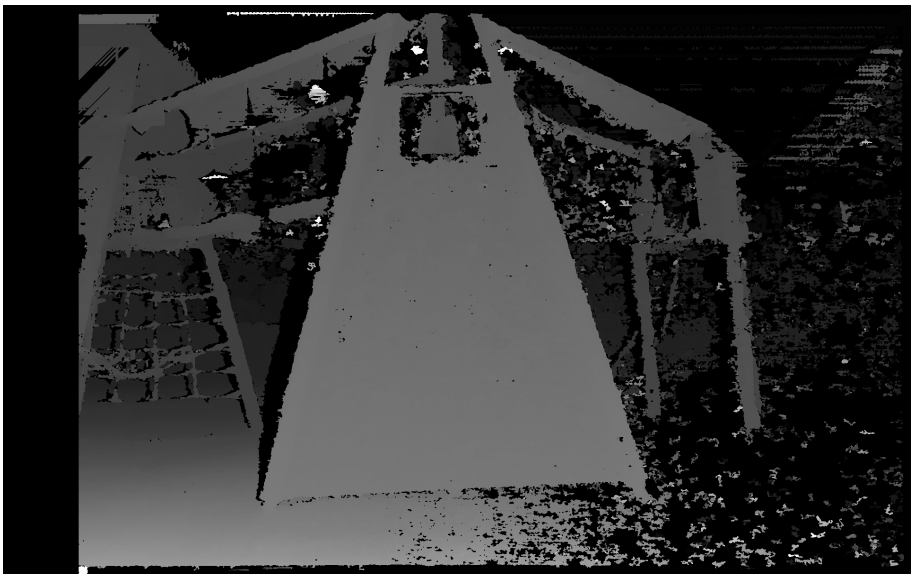

Figure 3.10: Playground Disparity Map

$$s_Z = \frac{Z^2}{Tc} s_{px'} = qms_{px'} \tag{3.3}$$

with height-to-base ratio $q = \frac{Z}{T}$, image scale $m = \frac{Z}{c}$ and $s_{px'}$ the error of the x-parallax. As T and c remain constant in a rigid setup, the error s_Z increases proportionally with the square of Z. The error of the parallax is defined as

$$s_{px'} = \frac{s_{x'}}{\sqrt{2}}. \tag{3.4}$$

with $s_{x'}$ being the error of the x-coordinate in the left/first picture.
To compute the error in X- and Y-direction [5] provides

$$\begin{aligned} s_X &= \sqrt{(\frac{x'}{c} s_z)^2 + (ms_{x'})^2} \\ s_Y &= \sqrt{(\frac{y'}{c} s_z)^2 + (ms_{y'})^2} \end{aligned} \tag{3.5}$$

According to Luhmann[5] the dominant term is the second summand inside the square root allowing for the following estimation:

$$s_X = s_Y = ms_{x'y'} \tag{3.6}$$

According to (3.3), the depth error increases quadratically whilst the error in X and Y increase linearly(3.6). Moreover, the error always relies on the baseline. Increasing the baseline also increases accuracy. The in 2.2 computed baseline is approx. 26cm.

Before coloring the LiDAR PCL, both registered clouds can be compared, displaying each with a distinct color. As seen in Fig.3.12 there is a clear offset in x direction (prominently recognizable by the left post of the climbing wall) as well as an offset in z-direction (made visible by the now pink climbing wall as well as the shifting of the upper right horizontal post). Also outliers can be seen especially in the upper left corner as well as along the right edge. These can be accredited to distortions in the depthmap caused by the overexposure of the sky (see Fig.3.9).

To visualize: the climbing wall measures around 275cm at the base and is about 350cm tall. Each post has the same diameter of roughly 14cm. The ropes seen in the left background measure ca. 1.5cm. The hanging bars left to the climbing wall measure 55cm×23cm×4.5cm (L×W×H).

The distance from the rig setup to the lower beginning of the climbing wall was measured to be around 430cm. In contrast, the 3D model by the stereo images returns a distance of roughly 410cm. At the same time the post on to the right of the climbing net is measured to be about 810cm from the setup whilst reconstruction returns values between 755cm and 786cm.

Since this is only the precision of the image reconstruction, the colored LiDAR PCL (see Fig.3.13) will be considered to evaluate the calibration. The first key element considered are the boulders in Fig.3.14. These are accurately colored with an approx. average of 1cm in x and y at a depth of 411cm to 480cm. This error worsens with increasing depth (and thus height) to up to 3cm mainly in y-direction. Figure 3.15 also shows little error with the background "bleeding" into the wall from the left. In contrast to Fig.3.14 the error is constant along the edge. The right edge of the wall 3.16 shows clear deviation, being partially colored by the background and floor as well (around 5cm). Differing from 3.15, the post is also projected onto the points behind having the same x-coordinate but different depth. This is caused as multiple points with different depth values are projected onto the same point in 2D (compare to 2.2.1 epipolar geometry/epilines). At the right hand side are three posts (see Fig.3.17), whereas one is almost completely occluded. All three are accurately colored with barely visible deviations. The colorings of the two swings in the top right corner is also precise. With the swing on the right having even its red ropes painted accordingly with little error. In contrast, the blue net between the three posts and the wall are too distorted to evaluate the coloring. This could be caused by the ropes being barely registered in the depthmap and thus having little influence to correct the ICP process. When looking at the hanging bars to the left (see Fig.3.18) the error in y-direction is obvious. This appears to originate from the LiDAR PCL, as the thickness/height is allegedly around 13.5cm according to those measurements in contrast to the manually retrieved 4.5cm. The colored part on the other hand is correctly 4.5cm thick. This error could be explained by edge distortions of the laser signals. Going further left, the net in Fig.3.19 appears to show the same distortion in y. Whilst the blue coloring is present at every rope, according to LiDAR these are 12cm of diameter as opposed to 1.5cm. The coloring is 4-6cm wide. What is striking is, that the errors dominating in 3.18 and 3.19 are present along the upper borders of the playground, especially along the horizontal posts in Fig.3.20. This appears to be a systematic error in the LiDAR measurements.

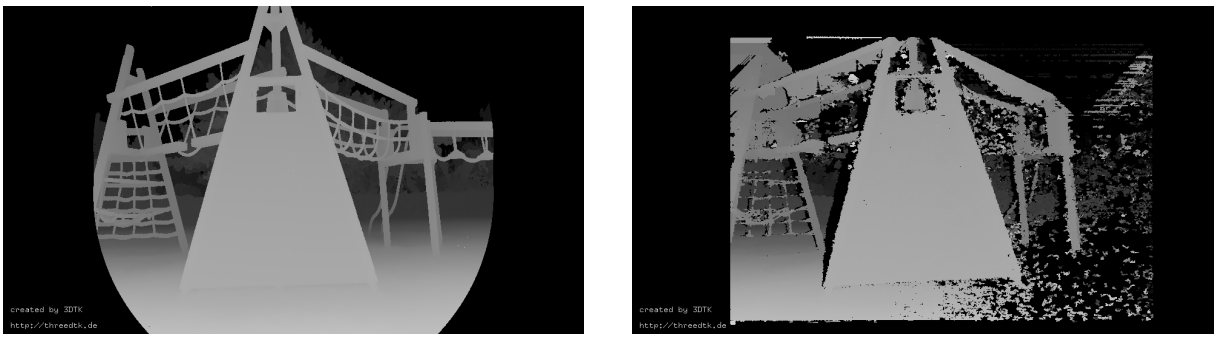

Figure 3.11: LiDAR PCL and Stereo PCL.

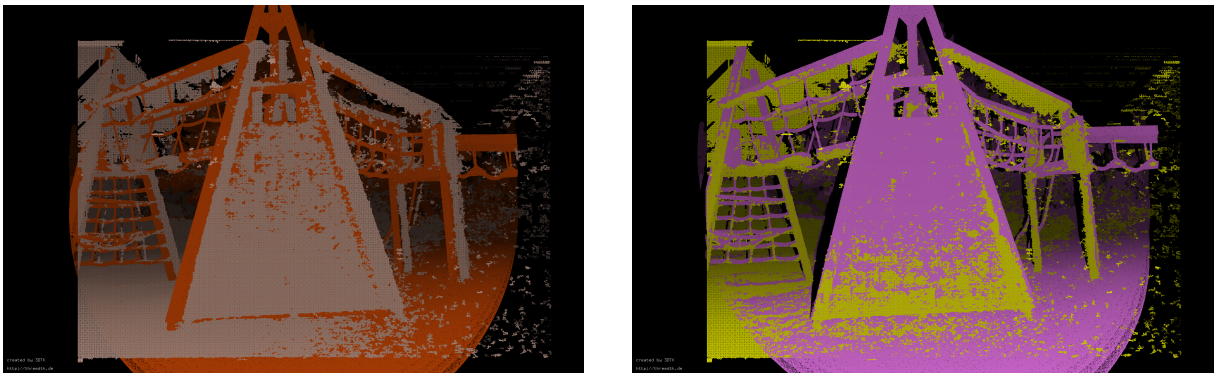

Figure 3.12: LiDAR(orange/pink) and Stereo(gray/yellow) point clouds before and after registration via ICP.

Figure 3.13: Colored LiDAR PCL

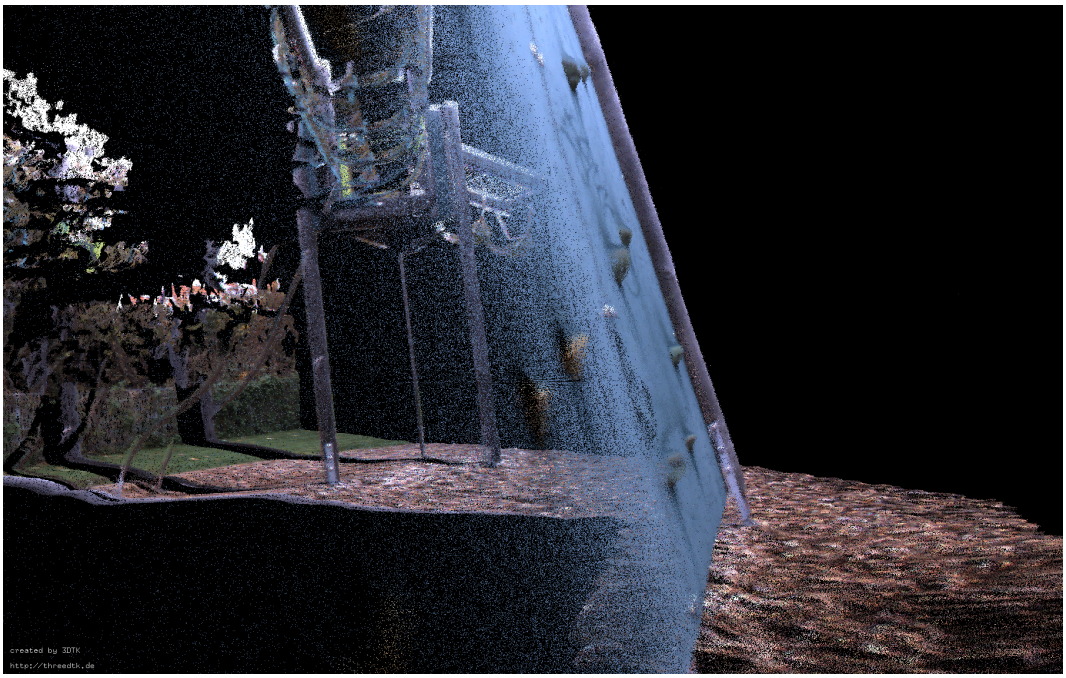

Figure 3.14: Accurately colored climbing stones

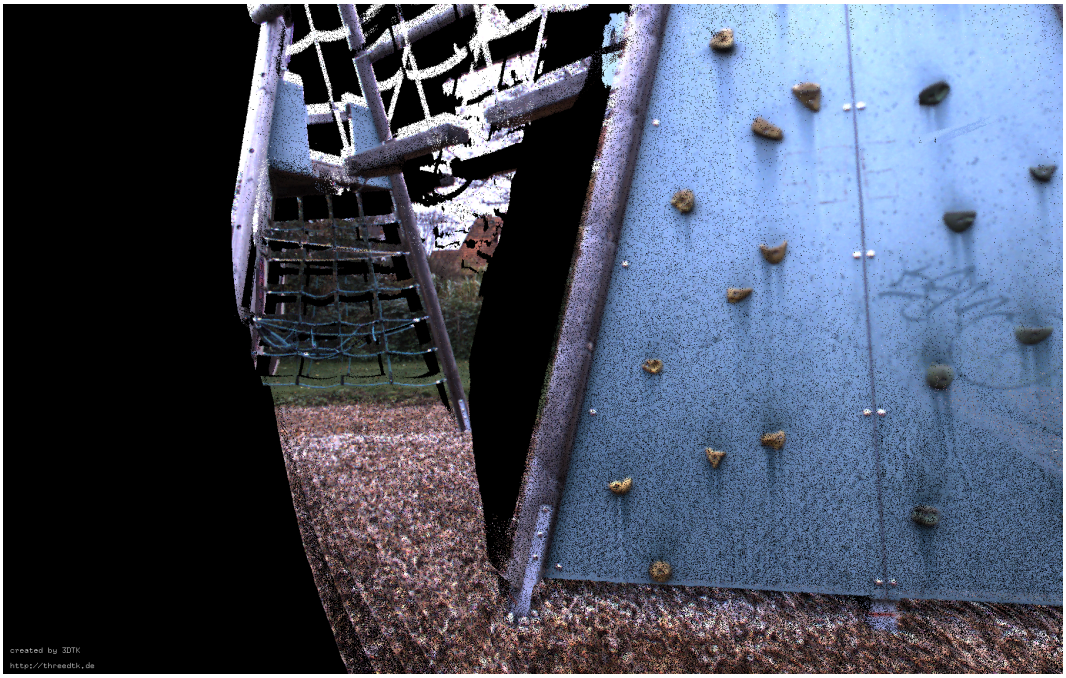

Figure 3.15: Small deviation at the left edge of the climbing wall

Figure 3.16: Big deviation at the right edge of the climbing wall

Figure 3.17: Accurately colored posts

Figure 3.18: Big deviation in y at the hanging bars.

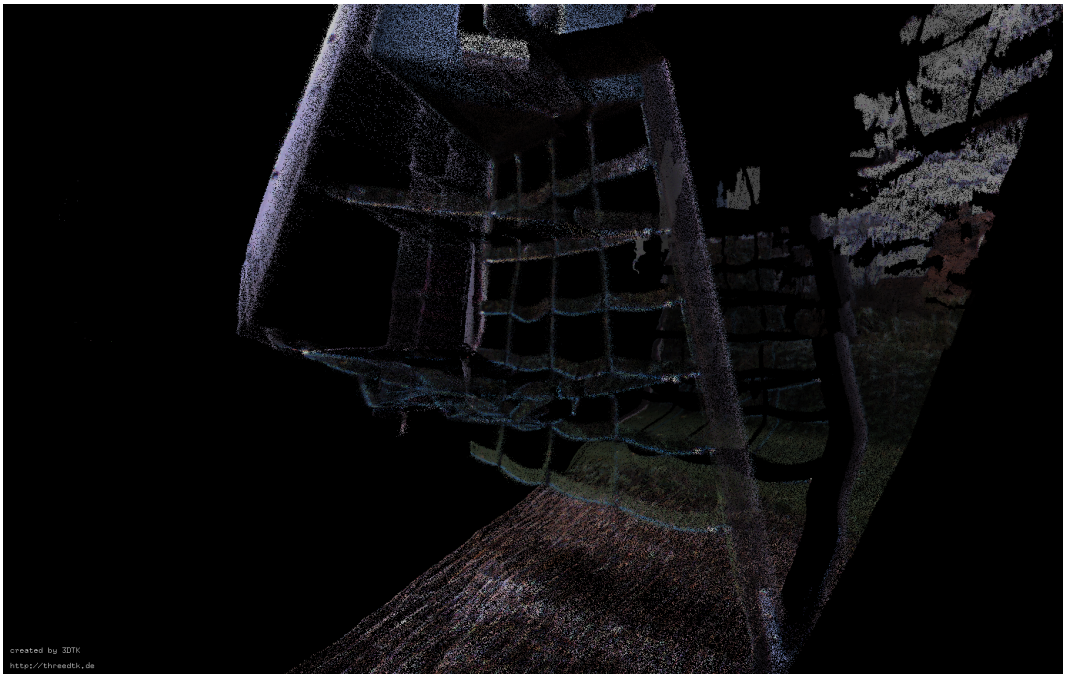

Figure 3.19: Distorted ropes.

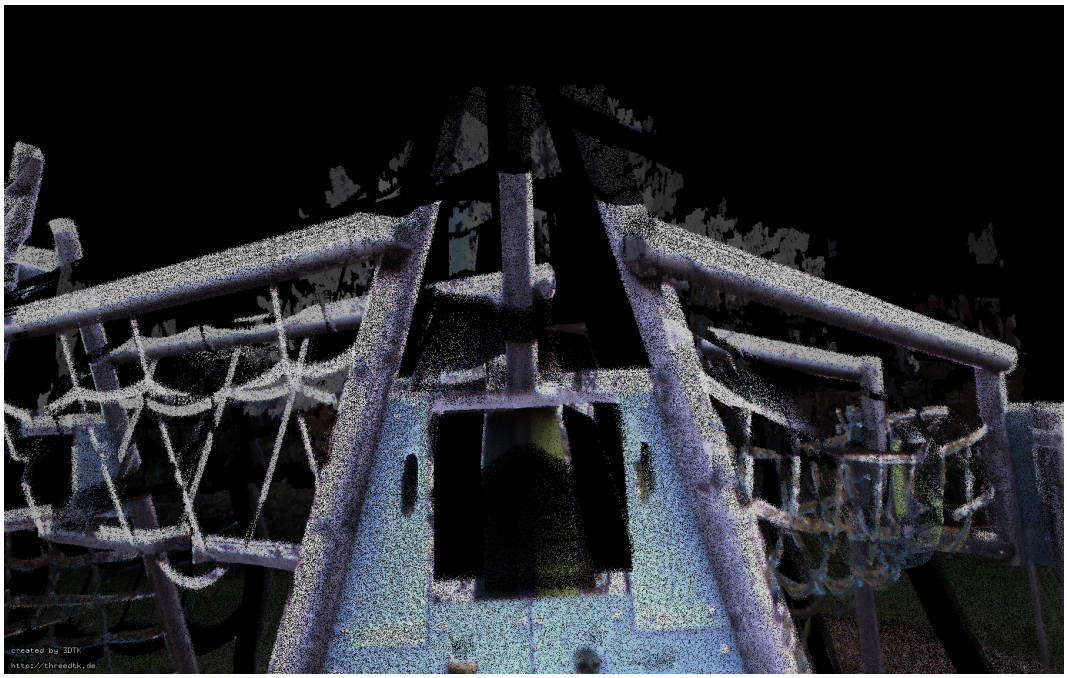

Figure 3.20: Constant deviation along the upper parts of the playground.

3.3 Conclusion

The work at hand aimed to calibrate a stereo camera setup with a LiDAR, which makes it possible to color the point cloud of the latter with high precision. This is partially achieved by calibrating each camera individually in Sec. 2.1.3 and applying the results in Sec. 2.2, retrieving the relative pose of the stereo setup. Furthermore this is used to perform depth estimation and thus 3D reconstruction of each scene viewed. The resulting point cloud is then registered with the LiDAR measurements using ICP. The resulting rotation and translation matrices are used to transform the LiDAR PCL into the camera coordinate sytem and project them onto the camera image plane, where coloring of the point cloud takes place.

The coloring at the points closer to the setup are colored accurate with errors in the low centimeter range. Smaller scaled elements cause great distortion, as seen in Fig.3.17, while bigger elements are still colored more precisely (see Fig.3.17). Occlusions lead to faulty coloring of points of same x and y in 2D, but different depth values. The Livox Avia delivered partially distorted values, a possible indication of systematic errors. Whether these stem from the LiDAR itself or are caused by the presented method needs further investigation. Another source of error is the small baseline of the handheld device, measuring ca. 26cm. As explained in Sec.3.2, the error in z is proportional to the size of the baseline. Comparing different baselines could be subject of future work. Alternatively, depending on the use case, a LiDAR with the ability to register points closer than 3m should be used, fitting the application in e.g. interiors. Moreover avoiding false coloring due to occlusion is also a topic for further studies. Since this work relied on the OpenCV standard library, implementations in `Matlab` or similar could be compared. Furthermore, different matching algorithms, such as `libSGM`[6] can be implemented and compared, although this specific example requires GPUs to run, trying to minimize outliers which may distort the ICP.

[6] https://github.com/fixstars/libSGM

Glossary

COP Center of Projection. 3, 10

FoV Field of View. 19, 25

ICP Iterative Closest Point. 17, 23–25, 27, 28, 33

LiDAR Light Detection and Ranging. 1, 10, 13, 16, 19, 23, 25, 27, 28, 33

PCL Point Cloud. 16, 23–25, 27, 28, 33, 39, 53

Bibliography

[1] Opencv documentation. https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html.

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

[3] M. Bleier, Y. Yuan, and A. Nüchter. A handheld stereo vision and LiDAR system for outdoor dense RGB-D mapping using depth map completion based on learned priors. In *Proceedings of LowCost3D 2024*, ISPRS Int. Archives Photogrammetry and Remote Sensing, Spatial Inf. Sci., XLVIII-2/W1-2024, Brescia, Italy, 12 2024.

[4] G. Bradski and A. Kaehler. *Learning OpenCV*. O'Reilly Media, Inc., 2008.

[5] T. Luhmann et. al. *Close-Range Photogrammetry and 3D Imaging*. De Gruyter, Inc., 2013.

[6] Andrea Fusiello. *Computer Vision: Three-dimensional Reconstruction Techniques*. Springer Nature Switzerland AG, 2024.

[7] S. Garrido-Jurado, R. Muñoz-Salinas, F.J. Madrid-Cuevas, and M.J. Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280–2292, 2014.

[8] Cyrill Stachniss. Photogrammetry ii - 03b - epipolar geometry and essential matrix (2015/16). https://www.youtube.com/watch?v=8pV-1GFsavA, 2015.

[9] Cyrill Stachniss. Camera calibration using zhang's method (cyrill stachniss). https://www.youtube.com/watch?v=-9He7Nu3u8s, 2020.

[10] Cyrill Stachniss. Direct linear transform for camera calibration and localization (cyrill stachniss). https://www.youtube.com/watch?v=3NcQbZu6xt8,, 2020.

[11] B. P. Wrobel W. Förstner. *Photogrammetric Computer Vision*. Springer International Publishing AG, Cham, Switzerland, 2016.

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

Appendix A

Code

The following codes use OpenCV 4.10.x and CMake version 3.30.

A.1 Code 1

Listing A.1: Programme to calibrate single cameras, performing stereo calibration and create PCL from such stereo recordings.

```
#include <iostream>
#include <vector>
#include <filesystem>
#include <opencv2/calib3d.hpp>
#include <opencv2/highgui.hpp>
#include <charuco_detector.hpp>
#include <opencv2/core/utility.hpp>
#include <aruco_samples_utility.hpp>
#include <opencv2/objdetect/charuco_detector.hpp>
#include <opencv2/imgproc.hpp>
#include "opencv2/imgcodecs.hpp"



using namespace std;
using namespace cv;

namespace {
    const char* about =
            "Intrinsic and Extrinsic Calibration of Stereo Cameras
              ↪ using Charuco\n";

    const char* keys  =
            "{w        |        | Number of squares in X direction }"
```

```
            "{h        |        | Number of squares in Y direction }"
            "{sl       |        | Square side length (in meters) }"
            "{ml       |        | Marker side length (in meters) }"
            "{d        |        | dictionary: DICT_4X4_50=0,
               ↪ DICT_4X4_100=1, DICT_4X4_250=2,"
            "DICT_4X4_1000=3, DICT_5X5_50=4, DICT_5X5_100=5,
               ↪ DICT_5X5_250=6, DICT_5X5_1000=7, "
            "DICT_6X6_50=8, DICT_6X6_100=9, DICT_6X6_250=10,
               ↪ DICT_6X6_1000=11, DICT_7X7_50=12,"
            "DICT_7X7_100=13, DICT_7X7_250=14, DICT_7X7_1000=15,
               ↪ DICT_ARUCO_ORIGINAL = 16}"
            "{@outfile |cam.yml| Output file with calibrated camera
               ↪ parameters }"
            "{cd       |        | Input file with custom dictionary }"
            "{dp       |        | File of marker detector parameters
               ↪ }"
            "{rs       | false | Apply refind strategy }"
            "{ipl      |        | Get path to left camera image files
               ↪ }"
            "{ipr      |        | Get path to right camera image
               ↪ files }"
            "{dis      |        | Path to disparity outputfile}"
            "{pfl      |        | Path to reconstruction pictures
               ↪ left}"
            "{pfr      |        | Path to reconstruction pictures
               ↪ right}";
}
```

```
void instrinsicCalib (aruco::CharucoBoard &board,
   ↪ aruco::CharucoDetector &detector, vector<String> filenames,
   ↪ Mat &cameraMatrix,
Mat &distCoeffs, double &repError, vector<vector<Point2f>>
   ↪ &allImagePoints, vector<vector<Point3f>> &allObjectPoints,
   ↪ Size &imageSize) {

    int calibrationFlags = 0;  //Standard OpenCV value

    // Collect data from each frame
    vector<Mat> allCharucoCorners, allCharucoIds;
```

```
vector<Mat> allImages;



size_t totalFiles = filenames.size();

for (size_t i = 0; i < totalFiles; i++) {
    Mat image, imageCopy;
    image = imread(filenames[i]);


    //if (!chessboard) {
        vector<int> markerIds;
        vector<vector<Point2f>> markerCorners;
        Mat currentCharucoCorners, currentCharucoIds;

        //Detect Charuco Board
        detector.detectBoard(image, currentCharucoCorners,
            ↪ currentCharucoIds);
        if(currentCharucoCorners.total() > 3) {
            vector<Point2f> currentImagePoints;
            vector<Point3f> currentObjectPoints;


            //Match image points
            board.matchImagePoints(currentCharucoCorners,
                ↪ currentCharucoIds, currentObjectPoints,
                ↪ currentImagePoints);
            if(currentImagePoints.empty() ||
                ↪ currentObjectPoints.empty()) {
                cout << "Point matching failed at " << i << endl;
                continue;
            }

            //cout << "Image matched" << endl;
            allCharucoCorners.push_back(currentCharucoCorners);
            allCharucoIds.push_back(currentCharucoIds);
            allImagePoints.push_back(currentImagePoints);
            allObjectPoints.push_back(currentObjectPoints);
            allImages.push_back(image);

            imageSize = image.size();
```

```
                double progress = static_cast<double>(i + 1) /
                    ↪ (double) totalFiles * 100;

                std::cout << "Matching points for intrinsics: "<<
                    ↪ progress << "% (" << (i + 1) << "/" <<
                    ↪ totalFiles << ")\r";
                std::cout.flush();

            } else cout << "Not enough Chaurco Corners detected" <<
                ↪ endl;


        //}
    }

    cout << endl;
    cout << "Calculating re-projection error..." << endl;


    // Calibrate camera using ChArUco
    repError = calibrateCamera(allObjectPoints, allImagePoints,
        ↪ imageSize, cameraMatrix, distCoeffs,
                                    noArray(), noArray(),
                                        ↪ noArray(), noArray(),
                                        ↪ noArray(),
                                        ↪ calibrationFlags);

    cout << "Intrinsic reprojection error: " << repError << endl;

}




void extrinsicsCalib (vector<vector<Point2f>>
    ↪ &allImagePointsLeft, vector<vector<Point2f>>
    ↪ &allImagePointsRight,
     vector<vector<Point3f>> &allObjectPointsLeft,
        ↪ vector<vector<Point3f>> &allObjectPointsRight) {

    int intersection;

    for (int i = 0; i < allObjectPointsLeft.size(); i++) {
```

```
vector<Point2f> currentImagePointsRight,
    ↪ currentImagePointsLeft;
vector<Point3f> currentObjectPoints;


//Checking which array is bigger
if (allObjectPointsLeft.at(i).size() <=
    ↪ allObjectPointsRight.at(i).size()) {
    for (int j = 0; j < allObjectPointsLeft.at(i).size();
        ↪ j++) {
        //Checks if outlier point
        if (allObjectPointsLeft.at(i).at(0) ==
            ↪ Point3f(-1,-1,0)){
            break;
        }
        //Searches for matching points
        for (intersection = 0; intersection <
            ↪ allObjectPointsRight.at(i).size();
            ↪ intersection++) {
            if (allObjectPointsLeft.at(i).at(j) ==
                ↪ allObjectPointsRight.at(i).at(intersection))
                ↪ break;
        }
        //if matching point found and for-loop did not run
            ↪ through, pushing back the "good" points
        if (intersection !=
            ↪ allObjectPointsRight.at(i).size()) {
            currentObjectPoints.push_back(allObjectPointsLeft.at(i).at(j))
            currentImagePointsLeft.push_back(allImagePointsLeft.at(i).at(j
            currentImagePointsRight.push_back(allImagePointsRight.at(i).at
        }
    }
} else {
    for (int j = 0; j < allObjectPointsRight.at(i).size();
        ↪ j++) {
        //Checks for outlier point
        if (allObjectPointsRight.at(i).at(j) ==
            ↪ Point3f(-1,-1,0)) {
            break;
        }
        //Searches for matching points
        for (intersection = 0; intersection <
            ↪ allObjectPointsLeft.at(i).size();
            ↪ intersection++) {
```

```
                    if (allObjectPointsRight.at(i).at(j) ==
                        ↪ allObjectPointsLeft.at(i).at(intersection))
                        ↪ break;
                }
                //if matching point found and for-loop did not run
                    ↪ through, pushing back the "good" points
                if (intersection !=
                    ↪ allObjectPointsLeft.at(i).size()) {
                    currentObjectPoints.push_back(allObjectPointsRight.at(i).at
                    currentImagePointsLeft.push_back(allImagePointsLeft.at(i).a
                    currentImagePointsRight.push_back(allImagePointsRight.at(i)
                }
            }
        }

        if (currentObjectPoints.size() >= 10) {
            allObjectPointsLeft.at(i) = currentObjectPoints;
            allImagePointsLeft.at(i) = currentImagePointsLeft;
            allImagePointsRight.at(i) = currentImagePointsRight;


        } else {
            allObjectPointsLeft.erase(allObjectPointsLeft.begin()
                ↪ +i);
            allObjectPointsRight.erase(allObjectPointsRight.begin()
                ↪ +i);
            allImagePointsLeft.erase(allImagePointsLeft.begin() +i);
            allImagePointsRight.erase(allImagePointsRight.begin()
                ↪ +i);

            if (allObjectPointsLeft.size() <= 0) {
                allObjectPointsLeft[0].clear();
                allObjectPointsRight[0].clear();
                break;
            }

            //decrement i++ as one entry was deleted
            i--;
        }

    }
```

```
}


void convertImages (const Mat& imageLeft, const Mat&
    ↪ cameraMatrixLeft, const Mat& distCoeffsLeft, const Mat& R1,
    ↪ const Mat& P1,
     const Mat& imageRight, const Mat& cameraMatrixRight, const Mat&
        ↪ distCoeffsRight, const Mat& R2, const Mat& P2,
     Mat& imageLeftRect, Mat& imageRightRect, const Size& imageSize) {

     Mat map11, map12, map21, map22;
     initUndistortRectifyMap(cameraMatrixLeft, distCoeffsLeft, R1,
        ↪ P1, imageSize, CV_16SC2, map11, map12);
     initUndistortRectifyMap(cameraMatrixRight, distCoeffsRight, R2,
        ↪ P2, imageSize, CV_16SC2, map21, map22);

     remap(imageLeft, imageLeftRect, map11, map12, INTER_LINEAR);
     remap(imageRight, imageRightRect, map21, map22, INTER_LINEAR);
};

//See OpenCV stereo_match.cpp
static void saveXYZ(const char* filename, const Mat& mat)
{
     const double max_z = 1.0e4;
     FILE* fp = fopen(filename, "wt");
     for(int y = 0; y < mat.rows; y++)
     {
          for(int x = 0; x < mat.cols; x++)
          {
               Vec3f point = mat.at<Vec3f>(y, x);
               if(fabs(point[2] - max_z) < FLT_EPSILON ||
                  ↪ fabs(point[2]) > max_z) continue;
               fprintf(fp, "%f %f %f \n", point[0], point[1] * -1.,
                  ↪ point[2]); //X, Y, Z
          }
     }
     fclose(fp);
}



int squaresX, squaresY;
float squareLength, markerLength;
```

```
String outputFile;

aruco::Dictionary dictionary;
aruco::CharucoParameters charucoParams;
aruco::DetectorParameters detectorParams;
vector<string> goodImageListLeft;
vector<string> goodImageListRight;

Mat cameraMatrixLeft, distCoeffsLeft;
Mat cameraMatrixRight, distCoeffsRight;
double repErrorLeft, repErrorRight;
bool chessboard;
String parentDirectory;


int main(int argc, char *argv[])
{
    //Reads terminal input
    CommandLineParser parser(argc, argv, keys);
    parser.about(about);

    //Checks, if more than 8 arguments are handed over
    if(argc < 11) {
        parser.printMessage();
        return 0;
    }

    squaresX = parser.get<int>("w");
    squaresY = parser.get<int>("h");
    squareLength = parser.get<float>("sl");
    markerLength = parser.get<float>("ml");
    outputFile = parser.get<string>(0);

    //Get Image paths
    auto folderpathLeft = parser.get<string>("ipl");
    auto folderpathRight = parser.get<string>("ipr");
    auto disparityFolder = parser.get<string>("dis");
    //String pclFolder = parser.get<string>("pcl");
    auto probeFilesLeft = parser.get<string>("pfl");
    auto probeFilesRight = parser.get<string>("pfr");

    parentDirectory = outputFile;
    auto pose = parentDirectory.rfind('/');
    parentDirectory = parentDirectory.substr(0, pose);
```

```
vector<String> filenamesLeft, filenamesRight;
vector<String> reconstrLeft, reconstrRight;

//Reading filenames to iterate later
for (const auto& filename :
    ↪ filesystem::directory_iterator(folderpathLeft)) {
    filenamesLeft.push_back(filename.path().string());
}
for (const auto& filename :
    ↪ filesystem::directory_iterator(folderpathRight)) {
    filenamesRight.push_back(filename.path().string());
}
for (const auto& filename :
    ↪ filesystem::directory_iterator(probeFilesLeft)) {
    reconstrLeft.push_back(filename.path().string());
}
for (const auto& filename :
    ↪ filesystem::directory_iterator(probeFilesRight)) {
    reconstrRight.push_back(filename.path().string());
}

sort(filenamesLeft.begin(), filenamesLeft.end());
sort(filenamesRight.begin(), filenamesRight.end());
sort(probeFilesLeft.begin(), probeFilesLeft.end());
sort(probeFilesRight.begin(), probeFilesRight.end());


detectorParams = readDetectorParamsFromCommandLine(parser);
dictionary = readDictionatyFromCommandLine(parser);
bool refindStrategy = parser.get<bool>("rs");

if(!parser.check()) {
    parser.printErrors();
    return 0;
}

if(refindStrategy) {
    charucoParams.tryRefineMarkers = true;
}

// Create charuco board object and CharucoDetector
```

```
aruco::CharucoBoard board(Size(squaresX, squaresY),
    ↪ squareLength, markerLength, dictionary);
aruco::CharucoDetector detector(board, charucoParams,
    ↪ detectorParams);

//Get camera instrinsics
vector<vector<Point2f>> allImagePointsLeft, allImagePointsRight;
vector<vector<Point3f>> allObjectPointsLeft,
    ↪ allObjectPointsRight;
Size imageSize;

cout << "Calculating left intrinsics..." << endl;
instrinsicCalib(board,detector, filenamesLeft, cameraMatrixLeft,
    ↪ distCoeffsLeft,repErrorLeft,
     allImagePointsLeft, allObjectPointsLeft, imageSize);

FileStorage fs(parentDirectory + "/leftIntrinsics.yml",
    ↪ FileStorage::WRITE);
if (fs.isOpened()) {
     cout << "Storing left intrinsics..." << endl;
     fs << "M1" << cameraMatrixLeft << "D1" << distCoeffsLeft <<
        ↪ "Rep1" << repErrorLeft;
     fs.release();
}

cout << "Calculating right intrinsics..." << endl;
instrinsicCalib(board,detector, filenamesRight,
    ↪ cameraMatrixRight, distCoeffsRight, repErrorRight,
     allImagePointsRight, allObjectPointsRight, imageSize);

fs.open(parentDirectory + "/rightIntrinsics.yml",
    ↪ FileStorage::WRITE);
if (fs.isOpened()) {
     cout << "Storing right intrinsics..." << endl;
     fs << "M2" << cameraMatrixRight << "D2" << distCoeffsRight
        ↪ << "Rep2" << repErrorRight;
     fs.release();
}


//Get Camera extrinsics
aruco::CharucoBoard boardLeft(Size(squaresX, squaresY),
    ↪ squareLength, markerLength, dictionary);
```

```
aruco::CharucoDetector detectorLeft(boardLeft, charucoParams,
    ↪ detectorParams);

aruco::CharucoBoard boardRight(Size(squaresX, squaresY),
    ↪ squareLength, markerLength, dictionary);
aruco::CharucoDetector detectorRight(boardRight, charucoParams,
    ↪ detectorParams);


if(filenamesLeft.size() == filenamesRight.size() &&
    ↪ reconstrLeft.size() == reconstrRight.size()) {
     Mat E, R, t, F;

     //Stereo Calibration, matching each image point first
     extrinsicsCalib(allImagePointsLeft, allImagePointsRight,
        ↪ allObjectPointsLeft, allObjectPointsRight);


     cout << "\nSo many correct image pairs: " <<
        ↪ allObjectPointsLeft.size() << endl;

     cout << "Calculating extrinsic reprojection error..." <<
        ↪ endl;

     double repErr = stereoCalibrate(allObjectPointsLeft,
        ↪ allImagePointsLeft, allImagePointsRight,
         cameraMatrixLeft, distCoeffsLeft, cameraMatrixRight,
         distCoeffsRight, imageSize, R, t, E, F,
            ↪ CALIB_FIX_INTRINSIC); //CALIB_FIX_INTRINSIC is
            ↪ set, as intrinsics do not change

     cout << imageSize << endl;
     cout << "R: " << R << endl;
     cout << "t: " << t << endl;


     cout << "Reprojection error: " << repErr << endl;

     Mat R1, R2, P1, P2, Q;

     cout << "Rectifying..." << endl;

     stereoRectify(cameraMatrixLeft,
        ↪ distCoeffsLeft, cameraMatrixRight,
```

```
        distCoeffsRight,imageSize, R, t, R1, R2, P1, P2, Q,
            ↪ CALIB_ZERO_DISPARITY, 1); // flags and alpha value
            ↪ are standard OpenCV values


    fs.open(parentDirectory + "/extrinsics.yml",
        ↪ FileStorage::WRITE);
    if (fs.isOpened()) {
        cout << "Storing extrinsics..." << endl;
        fs << "R" << R << "t" << t << "R1" << R1 << "R2" << R2
            ↪ << "P1" << P1 << "P2" << P2 << "Q" << Q << "Size"
            ↪ << imageSize << "F" << F;;
        fs.release();
    }

    for (int i = 0; i < reconstrLeft.size(); i++) {
        //Converting images to Matrices
        //Flag -1 as done in stereo_match by OpenCV for SGBM, 1
            ↪ if BM
        Mat imgLeft = imread(reconstrLeft[i], -1);

        Mat imgRight = imread(reconstrRight[i], -1);

        //applying rectification
        Mat imgLeftRect, imgRightRect;

        convertImages(imgLeft, cameraMatrixLeft, distCoeffsLeft,
            ↪ R1, P1, imgRight, cameraMatrixRight,
        distCoeffsRight, R2, P2, imgLeftRect, imgRightRect,
            ↪ imageSize);




        //Matching
        Mat disp, disp8;
        int numDisparities = 16*10; //standard OpenCV value is
            ↪ 16*8, 16*10 delivered slightly better results
        Ptr<StereoSGBM> sgbm =
            ↪ StereoSGBM::create(0,numDisparities,3); //standard
            ↪ values from OpenCV
```

```
sgbm->setP1(8*imgLeftRect.channels()*sgbm->getBlockSize()*sgbm->getBlo
    ↪ //standard values from OpenCV
sgbm->setP2(32*imgLeftRect.channels()*sgbm->getBlockSize()*sgbm->getB
    ↪ //standard values from OpenCV

//standard values from OpenCV
sgbm->setUniquenessRatio(10);    //10
sgbm->setSpeckleWindowSize(100);     //100
sgbm->setSpeckleRange(32);       //32
sgbm->setDisp12MaxDiff(1);       //1
int64 timeElapsed = getTickCount();

sgbm->compute(imgLeftRect, imgRightRect, disp);
timeElapsed = getTickCount() - timeElapsed;

double progress = static_cast<double>(i + 1) /
    ↪ reconstrLeft.size() * 100;

std::cout << "Creating Disparitymap and Pointcloud: " <<
    ↪ progress << "% (" << (i + 1) << "/" <<
    ↪ reconstrLeft.size()
<< ")\tTime elapsed " <<
    ↪ timeElapsed*1000/getTickFrequency() << "
    ↪ milliseconds<< "  << "\r";
std::cout.flush();


disp.convertTo(disp8, CV_8U,
    ↪ 255/(sgbm->getNumDisparities()*16.)); //standard
    ↪ OpenCV values

//Save disparity map
if (!disparityFolder.empty()) {
    String dispFile = disparityFolder +"/dispmap" +
        ↪ to_string(i+1) + ".png";
    imwrite(dispFile, disp8);
}

if(!outputFile.empty())
{
    String tempFileName;
    if (i < 10) {
        tempFileName = outputFile + "/scan00" +
            ↪ to_string(i) + ".3d";
```

```
                } else if (i < 100) {
                    tempFileName = outputFile + "/scan0" +
                        ↪ to_string(i) + ".3d";
                } else {
                    tempFileName = outputFile + "/scan" +
                        ↪ to_string(i) + ".3d";
                }
                cout << tempFileName << endl;
                Mat xyz;
                Mat floatDisp;
                Mat pose = Mat::zeros(2, 3, CV_32F);
                disp.convertTo(floatDisp, CV_32F, 1.0f/16.0f);
                    ↪ //standard OpenCV values
                reprojectImageTo3D(floatDisp, xyz, Q, true);
                saveXYZ(tempFileName.c_str(), xyz);
            }
        }


    } else cout << "Directories vary in size. Please make sure you
        ↪ only have image pairs" << endl;
        cout << endl;
        cout << "Done" << endl;
    return 0;
}
```

A.2 Code 2

Listing A.2: Programme to transform left-handed 3DTK PCLs to right handed (OpenCV convention) and color it with an input image.

```
#include <iostream>
#include <vector>
#include <filesystem>
#include <opencv2/calib3d.hpp>
#include <opencv2/highgui.hpp>
#include <aruco_samples_utility.hpp>
#include "opencv2/imgcodecs.hpp"
#include <fstream>


using namespace std;
using namespace cv;

bool readPointsFromFile(const string &filename, vector<Point3f>
    ↪ &points) {
    ifstream file(filename, std::ios::binary);
    if (!file.is_open()) {
        cerr << "Error: Could not open file " << filename << endl;
        return false;
    }

    if (file) {
        file.seekg(0, std::ios::end);
        std::cout << "File size: " << file.tellg() << " bytes" <<
            ↪ std::endl;
        file.seekg(0, std::ios::beg);
    }


    size_t lineCount = 0;
    string line;
    while (getline(file, line)) {
        ++lineCount;
    }
    cout << "There are so many points: " << lineCount << endl;
    file.clear();
    file.seekg(0, std::ios::beg);

    float x = 0, y = 0, z = 0;
```

```
    cout << "Reading object points..." << endl;

    while (file >> x >> y >> z) {
        // Read x, y, z values from the file
        points.push_back(Point3f(x, y, z));
    }
    file.close();
    return true;
}

void turnMatRight(const Mat &matIn, Mat &matOut) {
    cout << "Converting to right-handed coordinate system..." <<
       ↪ endl;
    matOut.at<float>(0, 0) = matIn.at<float>(1, 1);
    matOut.at<float>(1, 0) = -matIn.at<float>(1, 2);
    matOut.at<float>(2, 0) = -matIn.at<float>(1, 0);
    matOut.at<float>(3, 0) = -matIn.at<float>(1, 3);
    matOut.at<float>(0, 1) = -matIn.at<float>(2, 1);
    matOut.at<float>(1, 1) = matIn.at<float>(2, 2);
    matOut.at<float>(2, 1) = matIn.at<float>(2, 0);
    matOut.at<float>(3, 1) = matIn.at<float>(2, 3);
    matOut.at<float>(0, 2) = -matIn.at<float>(0, 1);
    matOut.at<float>(1, 2) = matIn.at<float>(0, 2);
    matOut.at<float>(2, 2) = matIn.at<float>(0, 0);
    matOut.at<float>(3, 2) = matIn.at<float>(0, 3);
    matOut.at<float>(0, 3) = -matIn.at<float>(3, 1);
    matOut.at<float>(1, 3) = matIn.at<float>(3, 2);
    matOut.at<float>(2, 3) = matIn.at<float>(3, 0);
    matOut.at<float>(3, 3) = matIn.at<float>(3, 3);
}

void turnPointRight(const vector<Point3f> &pointsIn, vector<Point3f>
   ↪ &pointsOut) {
    for (Point3f point: pointsIn) {
        Point3f temp;

        temp.x = point.x;
        temp.y = -point.y;
        temp.z = point.z;
        pointsOut.push_back(temp);
    }
}

void readMatrixFromFile(const string &filename, Mat &mat) {
```

```
cout << "Reading Matrix from file..." << endl;
ifstream file(filename, std::ios::binary);
if (!file.is_open()) {
    cerr << "Error: Could not open file " << filename << endl;
} else {
    file.clear();
    file.seekg(0, std::ios::end);

    std::string lastLine;
    char z;

    int y = 0;
    // Move backward until the start of the last line is found
    for (file.seekg(-1, std::ios::cur); file.tellg() > 0;
       ↪ file.seekg(-1, std::ios::cur)) {
        file.get(z);
        if (z == '\n') {
            if (y == 1) break; // Stop at the second newline
               ↪ character
            y++;
        }
        file.seekg(-1, std::ios::cur); // Move one step back
           ↪ again
    }

    std::getline(file, lastLine);

    file.close();

    istringstream lineStream(lastLine);

    float a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p;
    lineStream >> a >> b >> c >> d
               >> e >> f >> g >> h
               >> i >> j >> k >> l
               >> m >> n >> o >> p;

    if (lineStream.fail()) cerr << "Error: Could not parse the
       ↪ values of the last line" << endl;

    mat.at<float>(0, 0) = a;
    mat.at<float>(1, 0) = b;
    mat.at<float>(2, 0) = c;
    mat.at<float>(3, 0) = d;
```

```
            mat.at<float>(0, 1) = e;
            mat.at<float>(1, 1) = f;
            mat.at<float>(2, 1) = g;
            mat.at<float>(3, 1) = h;
            mat.at<float>(0, 2) = i;
            mat.at<float>(1, 2) = j;
            mat.at<float>(2, 2) = k;
            mat.at<float>(3, 2) = l;
            mat.at<float>(0, 3) = m;
            mat.at<float>(1, 3) = n;
            mat.at<float>(2, 3) = o;
            mat.at<float>(3, 3) = p;
        }
    }

    void readStereoValsFromFile(const string &fileIntrinsics, const
        ↪ string &fileExtrinsics, Mat &cameraMatrix,
                                    Mat &distCoeffs, Size size, bool right) {
        if (!fileIntrinsics.empty()) {
            // reading intrinsic parameters
            FileStorage fs(fileIntrinsics, FileStorage::READ);
            if (!fs.isOpened()) cerr << "Failed to open file " <<
                ↪ fileIntrinsics.c_str() << endl;
            cout << "Reading camera parameters from file..." << endl;
            if (!right) {
                fs["M1"] >> cameraMatrix;
                fs["D1"] >> distCoeffs;
            } else {
                fs["M2"] >> cameraMatrix;
                fs["D2"] >> distCoeffs;
            }

            cout << "Camera Matrix: " << cameraMatrix << endl;
            cout << "Distortion Coefficients: " << distCoeffs << endl;

            fs.open(fileExtrinsics, FileStorage::READ);
            if (!fs.isOpened()) cerr << "Failed to open file " <<
                ↪ fileExtrinsics.c_str() << endl;

            fs["Size"] >> size;
        } else cerr << "File empty " << fileIntrinsics.c_str() << endl;
    }

    void rotatePCL(const Mat &rotMat, const vector<Point3f> &pointsIn,
```

```
    ↪ vector<Point3f> &pointsOut) {
    cout << "Rotating point cloud..." << endl;

    for (const Point3f &point: pointsIn) {
        Mat temp = (Mat_<float>(4, 1) << point.x, point.y, point.z,
            ↪ 1);
        Mat rotatedPCLMat = rotMat * temp;


        Point3f rotPoint;
        rotPoint.x = rotatedPCLMat.at<float>(0, 0);
        rotPoint.y = rotatedPCLMat.at<float>(1, 0);
        rotPoint.z = rotatedPCLMat.at<float>(2, 0);

        pointsOut.push_back(rotPoint);
    }
}


void colorLidarPCL(const Mat &image, const Mat &cameraMatrix, const
    ↪ Mat &distCoeffs, const Mat &R, const Mat &t,
                   const vector<Point3f> &objectPoints,
                       ↪ vector<Vec3b> &coloredVals) {
    vector<Point2f> imagePoints;
    //Converting Matrix R to Vector r for projectPoints()
    Mat r = Mat::zeros(3, 1,CV_32F);
    Rodrigues(R, r);

    //Rotating Lidar PCL to camera coordinate system (3D->2D)
    projectPoints(objectPoints, r, t, cameraMatrix, distCoeffs,
        ↪ imagePoints);


    cout << "Coloring Objectpoints..." << endl;
    for (auto &imagePoint: imagePoints) {
        int u = round(imagePoint.x);
        int v = round(imagePoint.y);

        //If point is within image resolution, add color, if not,
            ↪ paint black (or white)
        if (u >= 0 && u < image.cols && v >= 0 && v < image.rows) {
            //cout << "Colored Objectpoint: " << u << " " << v <<
                ↪ endl;
            const Vec3b& colorBGR = image.at<Vec3b>(v, u);
```

```
            Vec3b colorRGB;
            colorRGB[0] = colorBGR[2];
            colorRGB[1] = colorBGR[1];
            colorRGB[2] = colorBGR[0];
            coloredVals.push_back(colorRGB);
        } else {
            coloredVals.push_back(Vec3b(0, 0, 0));
        }
    }
}

void saveRotandColPCL(const vector<Point3f> &objectPoints, const
    ↪ vector<Vec3b> &coloredVals, const char *fileName) {
    cout << "Saving point cloud..." << endl;
    if (objectPoints.size() == coloredVals.size()) {
        FILE *fp = fopen(fileName, "wt");
        for (size_t i = 0; i < objectPoints.size(); i++) {
            Point3f p = objectPoints[i];
            const Vec3b& color = coloredVals[i];

            fprintf(fp, "%f %f %f %u %u %u \n", p.x, p.y, p.z,
                ↪ color[0], color[1], color[2]);
        }
    } else cerr << "Sizes differ!" << endl;
}

int main() {
    vector<Point3f> pointsOGLeft, pointsRotLeft, pointsRotRight;
    vector<Vec3b> coloredVals;
    Mat rotMatLeft = Mat::zeros(4, 4, CV_32F);
    Mat rotMatRight = Mat::zeros(4, 4, CV_32F);
    String fileIntrinsics = "/path/to/intrinsics.yml";
    String fileExtrinsics = "/path/to/extrinsics.yml";
    Mat cameraMatrix, distCoeffs;
    Mat R = Mat::eye(3, 3, CV_32F);
    Mat t = Mat::zeros(3, 1, CV_32F);
    Size size;

    String picFile =
            "/path/to/image/";
    Mat image = imread(picFile);

    readPointsFromFile("/path/to/LiDARPCL/",
                       pointsOGLeft);
```

```
    readMatrixFromFile(
        "/path/to/ICPvalues/", rotMatLeft);
    rotatePCL(rotMatLeft, pointsOGLeft, pointsRotLeft); //Applying
        ↪ ICP rotation
    turnPointRight(pointsRotLeft, pointsRotRight); //Transforming
        ↪ pointcloud to right-handed coordinate system
    readStereoValsFromFile(fileIntrinsics, fileExtrinsics,
        ↪ cameraMatrix, distCoeffs, size, false);
    colorLidarPCL(image, cameraMatrix, distCoeffs, R, t,
        ↪ pointsRotRight, coloredVals); //Accessing color values of
        ↪ image and mapping it to 3d points
    saveRotandColPCL(pointsRotLeft, coloredVals,
                     "/path/to/savefile");
    cout << "Done!" << endl;

    return 0;
}
```

Proclamation

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

Würzburg, January 2025