Zurich University
of Applied Sciences

School of
Engineering
InES Institute of
Embedded Systems

Augmented Reality Platform using Sensor Fusion

Zurich University of Applied Sciences

Institute of Embedded Systems

Authors	Marcel Wegmann

Version	0.1
Last changes	December 17, 2021

Contact Information
c/o Inst. of Embedded Systems (InES)
Zürcher Hochschule für Angewandte Wissenschaften
Technikumstrasse 22
CH-8401 Winterthur

Tel.: +41 (0)58 934 75 25
Fax.: +41 (0)58 935 75 25

E-Mail: wegr@zhaw.ch
Homepage: `http://www.ines.zhaw.ch`

Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering
InES Institute of
Embedded Systems

Erklärung betreffend das selbständige Verfassen einer Vertiefungsarbeit/Masterarbeit im Departement School of Engineering

Mit der Abgabe dieser Vertiefungsarbeit/Masterarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat.

Der/die unterzeichnende Studierende erklärt, dass alle verwendeten Quellen (auch Internetseiten) im Text oder Anhang korrekt ausgewiesen sind, d.h. dass die Vertiefungsarbeit/Masterarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten Paragraph 39 und Paragraph 40 der Rahmenprüfungsordnung für die Bachelor- und Masterstudiengänge an der Zürcher Hochschule für Angewandte Wissenschaften vom 29. Januar 2008 sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum: Unterschrift:

…… ……..

Das Original dieses Formulars ist bei der ZHAW-Version aller abgegebenen Vertiefungsarbeiten/Masterarbeiten im Anhang mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

Abstract

This is gonna be really abstract...

Preface

Thanks! I guess....?

Contents

1 Introduction **2**
1.1 Scope . . . 2
1.2 Initial Situation . . . 2
1.3 Goals . . . 2
1.4 Target audience . . . 2

2 Motivation **3**

3 Concept **4**

4 Fundamentals **5**
4.1 3D Cameras . . . 5
4.2 Mathematics for Rotation and Translation . . . 6
4.2.1 Euler Rotations and Linear Algebra . . . 6
4.2.2 Rotation with Quaternions . . . 7
4.2.3 Singular Value Decomposition (SVD) . . . 8
4.2.4 Spatial coordinates and device coordinates . . . 10
4.3 Standard Vulkan Coordinate System . . . 10
4.4 Camera Calibration . . . 13
4.5 Kalman filter . . . 14

5 Implementation **17**
5.1 Hardware . . . 17
5.1.1 Evaluation of ToF Camera . . . 17
5.1.2 Camera Head . . . 18
5.1.3 Processing System . . . 19
5.1.4 Video inputs . . . 19
5.1.5 Unified Memory . . . 20
5.2 Software Architecture . . . 21
5.3 Motion estimation from ToF camera . . . 22
5.3.1 ToF Camera calibration . . . 23
5.3.2 SIFT feature extraction . . . 24
5.3.3 Find rotation and translation . . . 26
5.3.4 3D Random Sample Consensus (RANSAC) . . . 28
5.4 Sensor Fusion . . . 30
5.4.1 Gyroscope and Accelerometer . . . 30
5.5 Sensor Fusion with Kalman Filter . . . 31
5.5.1 Prediction . . . 31
5.5.2 Correction: Rotation . . . 33
5.5.3 Rotation Drift compensation with Accelerometer . . . 34
5.5.4 Correction: Translation . . . 34
5.6 Raspberry Pi Camera calibration . . . 35
5.7 Video display . . . 35

6 Testing and Results **36**
6.1 ToF Camera . . . 36
6.1.1 Setup . . . 36
6.1.2 Distance measurement . . . 37
6.1.3 3D Scene Reconstruction . . . 37
6.1.4 RANSAC feature matching . . . 37
6.1.5 Rotation from ToF camera . . . 39

6.1.6 Translation from ToF camera . . . 40
6.2 Inertial Measurement Unit . . . 42
6.2.1 Accelerometer . . . 42
6.3 Kalman Filter . . . 43
6.3.1 Rotation . . . 43
6.3.2 Translation . . . 45
6.4 Cross sensitivity . . . 45
6.5 Performance . . . 45

7 Conclusion **46**
7.1 Possible improvement . . . 46
7.1.1 Proposal: Position Estimation by Camera . . . 46
7.2 Outlook . . . 46

Verzeichnisse **47**
Bibliography . . . 47
List of Figures . . . 49
List of Tables . . . 51
Abkürzungsverzeichnis . . . 52

8 Appendix **53**
8.1 Structure of the Git-Repository . . . 53
8.1.1 Documentation . . . 53
8.1.2 Source . . . 53
8.1.3 Literature . . . 53
8.2 Used software . . . 53
Listings . . . 53

1 Introduction

This introduction is very doge.

1.1 Scope

The scope is much wow.

1.2 Initial Situation

The initial situation is kinda sus.

1.3 Goals

There are so many goals!

1.4 Target audience

This document is targeted at readers with a basic understanding of computer science and computer vision. Prior knowledge in linear algebra is beneficial.

2 Motivation

Multiple consulting companies like Deloitte and KPMG described virtual, augmented, and extended reality as possibly the biggest source of digital disruption since the smartphone [1] and the next big thing of the digital environment [2].
While augmented reality platforms already exist in consumer products, the know-how is developed within the walls of multi-billion tech companies like Apple or Microsoft. According to Bloomberg, Facebooks AR and VR, and hardware teams account for more than 6000 employees [3].
The AR game Pokemon Go became a massive success in 2016 and remained the most famous AR game to date. Its success even brought investors to buy the stock without prior research. Nintendo's market value almost doubled before a warning given by Nintendo themselves that their economic stake in Pokemon Go is limited. [4]
Another example is the infamous Google Glass, a specialized AR goggle used to browse the web or take photographs. [5] The unwillingness of some users of Google Glass to take off the goggles during a conversation or secretly filming people in public sparked criticism to the extent that the word "Glasshole" emerged. [6] Google themselves reacted by writing guidelines for Google Glass' early adopters. [6]
Microsoft opted for a bulkier design for the HoloLens [7] and targeted it towards a professional environment, for example, to support Airbus technicians at maintenance [8].
Apple did not unveil any AR goggles yet but demonstrates their advancements in augmented reality on their LiDaR equipped iPhones and iPads. [9]
While these companies provide specialized hardware and programming interfaces, the mechanisms behind them are corporate secrets. Leveraging the power of off-the-shelf compute modules and sensors allows creating a minimal working prototype for augmented reality.

3 Concept

A minimal working prototype of an AR system consists of two key components: Awareness regarding the environment and awareness regarding the motion of the camera.
Scanning the environment is required for a system to know what parts of the image could be enhanced by a virtual object. This prototype will focus on motion estimation, omitting surface detection.
Tracking the exact motion of the camera allows adjusting the position of the projected display accordingly. The virtual projection then appears to be fixed to the wall, no matter how the camera head is moved.
The prototype is developed on a Nvidia Jetson Xavier AGX 8GB system, to which an accelerometer- and gyroscope-sensor, a time of flight (ToF) camera, and a Raspberry Pi Camera are attached.
The ToF camera provides three dimensional information that can be used for both finding flat walls and giving motion information on a frame by frame basis. The accelerometer complements the motion information given by the camera, especially helping whenever the motion is too fast for a sharp image.
The Raspberry Pi camera acts as a viewfinder, in which the virtual projection is visible.

TODO

Figure 3.1: Concept

4 Fundamentals

The following chapter describes methods and technologies that are used within this thesis.

4.1 3D Cameras

In the field of 3D mapping, two expressions often get mentioned: LiDaR sensors and ToF cameras. As the basic principle in both technologies relies on measuring the Time of Flight (ToF) and is in both cases Light-Detection and Ranging, both expressions are technically ambivalent.
A LiDaR sensor is often referred to work together with a moving laser, that scans its surroundings. [10] The mechanical mounting of such a device is too bulky to be embedded in a modern smartphone, which is why solid-state LiDaR sensors are used. A solid-state LiDaR sensor projects a grid of laser dots onto the scene, as seen in image 4.1. The time of flight for each dot is measured individually.

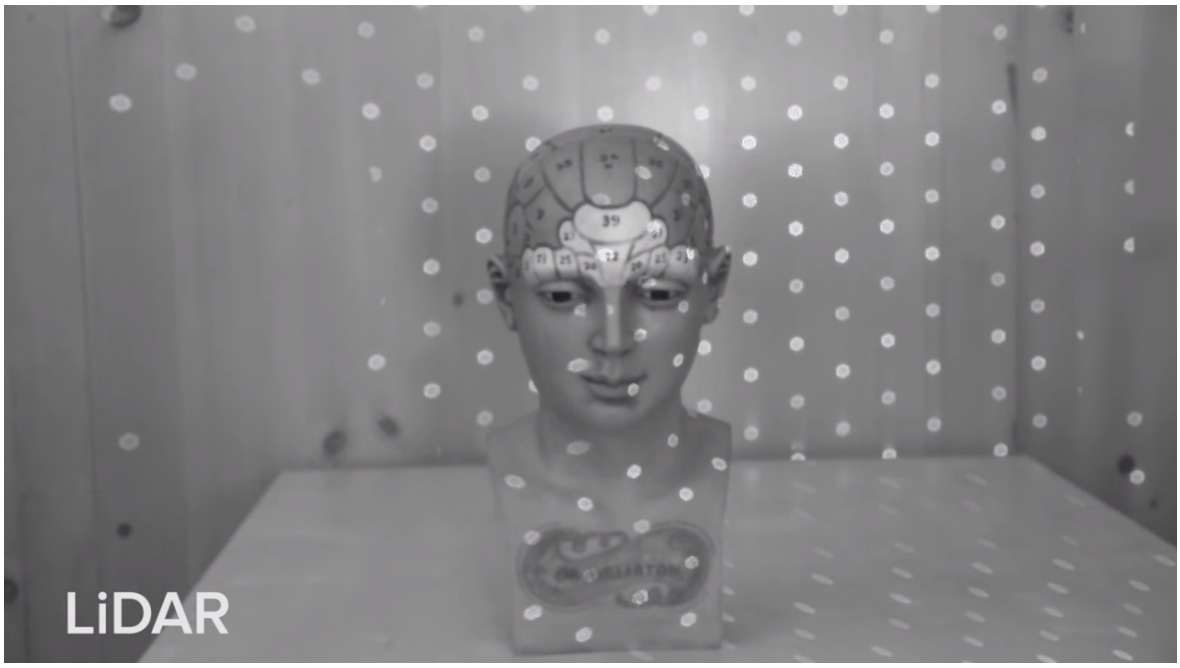

Figure 4.1: Projected dots from the LiDaR scanner of an Apple iPad Pro 2020, made visible with an infrared camera. Image source: iFixit.com

Another method for 3D mapping is, using one wide-area infrared flash and measuring the time of flight on each sensor pixel. This approach, in contrast, is often referred to be a ToF camera. Android-powered smartphones of various manufacturers use ToF cameras to improve autofocus capabilities or add artificial bokeh.
While the physical principle in both technologies is the same, the LiDaR scanner generates a point cloud, while the ToF camera outputs a depth map image. With mathematical transformations, both outputs are equivalent. Another difference is that a ToF camera can double as a grayscale infrared camera, providing an image by itself.
The sensor used for this thesis follows the principle of a ToF camera. The measured radial distance from the sensor for each pixel allows the three-dimensional reconstruction of the scene. As the distance measurement is radial, it needs to be corrected to obtain flat surfaces. A reference measurement helps solve this problem, section 5.3.1 explains the performed correction.

4.2 Mathematics for Rotation and Translation

Augmented reality relies on having accurate positional and angular information to estimate the required size and warp of a virtual object projected into the real world. A MEMS module containing a gyroscope and an accelerometer provides rotation and acceleration information to the system to assist the positional tracking.

4.2.1 Euler Rotations and Linear Algebra

A common way to calculate rotations and translations are matrix-vector multiplications. The standard matrices for rotating with the angle ϕ around X, Y and Z are shown in the following:

$$A_{rot,X} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\phi & -sin\phi \\ 0 & sin\phi & cos\phi \end{bmatrix} \quad A_{rot,Y} = \begin{bmatrix} cos\phi & 0 & sin\phi \\ 0 & 1 & 0 \\ -sin\phi & 0 & cos\phi \end{bmatrix} \quad A_{rot,Z} = \begin{bmatrix} cos\phi & sin\phi & 0 \\ -sin\phi & cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

A combination of the three matrices leads to a rotation matrix with a rotation axis that is not strictly bound to X, Y, or Z. Matrix multiplication is not commutative, so the order of the multiplications matters. In the following example, the vector gets rotated first around X, then Y, and around Z in the end. This chain of matrix operations is read from right to left in the equation.

$$A_{rot} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix} = A_{rot,Z} \cdot A_{rot,Y} \cdot A_{rot,X}$$

With this matrix, a three dimensional vector can be rotated at once around an arbitrary axis for the desired angle. Applying this transformation to each vertex of a virtual 3D object results in a rotation of the whole object around the origin $(0,0,0)$.

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

To avoid using an inhomogenous linear system for moving an object, a fourth dimension is needed. By extending the vectors with a 1 and using the fourth column in the matrix to alter X, Y and Z, these vector entries can be moved without applying any rotation.

$$\begin{pmatrix} x+\Delta X \\ y+\Delta Y \\ z+\Delta Z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & \Delta X \\ 0 & 1 & 0 & \Delta Y \\ 0 & 0 & 1 & \Delta Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

To combine the rotation matrix with the translation matrix, the 3x3 rotation matrix gets placed top-left into the 4x4 unit matrix. Now, the rotation matrix also being a 4x4 matrix, rotations and translations can be chained up following the common laws of linear algebra. Chaining up translations and rotations allows moving the rotation axis for an object.

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \Delta X \\ a_{1,0} & a_{1,1} & a_{1,2} & \Delta Y \\ a_{2,0} & a_{2,1} & a_{2,2} & \Delta Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

The dependency on the order of the rotations poses a problem visualized in image 4.2: The values returned by a gyroscope would need to be applied all at once and not one after another. Replacing rotation matrices by quaternions - described in section 4.2.2 - solves this problem.

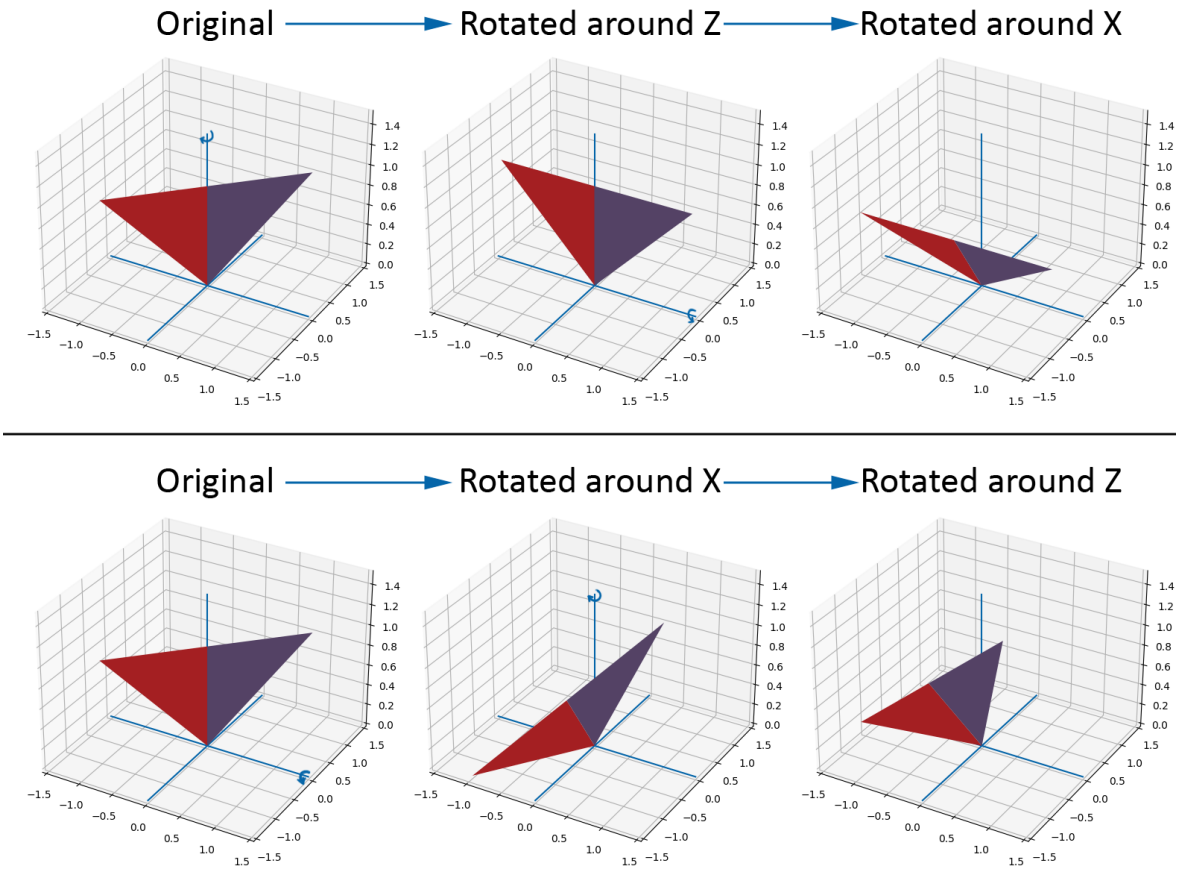

Figure 4.2: Euler rotations are dependent on the order of the individual rotations. Rotating around X and then Z results in a different outcome, than first rotating around Z and then X.

4.2.2 Rotation with Quaternions

Quaternions - also known as "Hamilton Numbers" - are the four dimensional equivalent to complex numbers. The theory of quaternions is vast, only the parts needed in this thesis are explained in this section. Analogue to complex numbers, quaternions also consist of a real part, but add three imaginary parts $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$. Most often, quaternions get represented in the form $a+b\mathbf{i}+c\mathbf{j}+d\mathbf{k}$ or a four dimensional vector (a,b,c,d).
The relations of the different imaginary parts in a quaternion are defined as following: $\mathbf{i}^2=\mathbf{j}^2=\mathbf{k}^2=\mathbf{ijk}=-1$, $\mathbf{ij}=\mathbf{k},\mathbf{jk}=\mathbf{j},\mathbf{ki}=\mathbf{j}$, and $\mathbf{ji}=-\mathbf{k},\mathbf{kj}=-\mathbf{i},\mathbf{ik}=-\mathbf{j}$. These definitions cause the quaternion multiplication (or Hamilton Product) to be non-commutative. The following equation shows how the Hamilton Product is calculated:

$$(a_1+b_1\mathbf{i}+c_1\mathbf{j}+d_1\mathbf{k})(a_2+b_2\mathbf{i}+c_2\mathbf{j}+d_2\mathbf{k}) = \begin{pmatrix} a_1a_2 & -b_1b_2 & -c_1c_2 & -d_1d_2 \\ (a_1b_2 & +b_1a_2 & +c_1d_2 & -d_1c_2)\mathbf{i} \\ (a_1c_2 & -b_1d_2 & +c_1a_2 & +d_1b_2)\mathbf{j} \\ (a_1d_2 & +b_1c_2 & -c_1b_2 & +d_1a_2)\mathbf{k} \end{pmatrix}$$

The neutral element of the quaternion multiplication is:

$$\begin{pmatrix} 1 \\ 0\mathbf{i} \\ 0\mathbf{j} \\ 0\mathbf{k} \end{pmatrix}$$

A plain three-dimensional vector is written in quaternions by only using the imaginary components.

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ x\mathbf{i} \\ y\mathbf{j} \\ z\mathbf{k} \end{pmatrix}$$

According to Euler's rotation theorem, providing a certain angle θ and a rotation axis $\vec{v_r}$, allows describing any rotation in three-dimensional space. These values - the angle and the axis - are embedded within the rotation quaternion. In addition, the rotation quaternion is required to have the norm being equal one. Such a rotation quaternion is sometimes named a unit quaternion or a versor.

$$\vec{v}_r = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad ; \quad \vec{v}_{r,norm} = \begin{pmatrix} \frac{x}{\|v_r\|} \\ \frac{y}{\|v_r\|} \\ \frac{z}{\|v_r\|} \end{pmatrix} = \begin{pmatrix} x_{norm} \\ y_{norm} \\ z_{norm} \end{pmatrix} \quad ; \quad Q_{rot} = \begin{pmatrix} \cos(\frac{\theta}{2}) \\ x_{norm}\sin(\frac{\theta}{2})\mathbf{i} \\ y_{norm}\sin(\frac{\theta}{2})\mathbf{j} \\ z_{norm}\sin(\frac{\theta}{2})\mathbf{k} \end{pmatrix} \quad ; \quad \|Q_{Rot}\| = 1$$

To apply this rotation quaternion Q_{rot} to a vector $\vec{v}$, it needs to be conjugated and calculated from the front and from the back as shown in the following. Conjugation of a unit quaternion is done by flipping the sign in each imaginary part. Vector $\vec{u}$ is the rotated vector $\vec{v}$.

$$\vec{v} = \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 0 \\ a\mathbf{i} \\ b\mathbf{j} \\ c\mathbf{k} \end{pmatrix} \quad ; \quad \vec{u} = Q_{rot}\vec{v}Q_{rot}^{-1} = \begin{pmatrix} \cos(\frac{\theta}{2}) \\ x_{norm}\sin(\frac{\theta}{2})\mathbf{i} \\ y_{norm}\sin(\frac{\theta}{2})\mathbf{j} \\ z_{norm}\sin(\frac{\theta}{2})\mathbf{k} \end{pmatrix} \begin{pmatrix} 0 \\ a\mathbf{i} \\ b\mathbf{j} \\ c\mathbf{k} \end{pmatrix} \begin{pmatrix} \cos(\frac{\theta}{2}) \\ -x_{norm}\sin(\frac{\theta}{2})\mathbf{i} \\ -y_{norm}\sin(\frac{\theta}{2})\mathbf{j} \\ -z_{norm}\sin(\frac{\theta}{2})\mathbf{k} \end{pmatrix}$$

By converting a rotation quaternion to a rotation matrix, the gap to Vulkan can be bridged. This formula only applies to true rotation quaternions, therefore the norm needs to be equal to one.

$$Q_{Rot} = \begin{pmatrix} a \\ b\mathbf{i} \\ c\mathbf{j} \\ d\mathbf{k} \end{pmatrix} \quad ; \quad \|Q_{Rot}\| = 1 \quad ; \quad A_{Rot} = \begin{bmatrix} 1-2(c^2+d^2) & 2(bc-ad) & 2(bd+ac) \\ 2(bc+ad) & 1-2(b^2+d^2) & 2(cd-ab) \\ 2(bd-ac) & 2(cd+ab) & 1-2(b^2+c^2) \end{bmatrix}$$

The basic quaternion operations - like the multiplication - are included in the linmath-library. The linmath-library represents a quaternion as a four-dimensional float-vector in the ordering (b, c, d, a) - the real part being the last element.

4.2.3 Singular Value Decomposition (SVD)

The singular value decomposition is a linear algebra method that allows finding the optimal rotation matrix between two matching point clouds. Adding additional algebraic steps makes it possible to

find the optimal translation vector in addition.[11] For demonstration purposes, the recipe is carried out in a 3D example in section 5.3.3 in the Implementation chapter.
The SVD decomposes any given matrix M of the dimension $n \times m$ into three matrices U of dimension $n \times n$, Σ of dimension $n \times m$, and V of dimension $m \times m$.[12] If M is real, U and V are guaranteed to be orthogonal, while Σ is a diagonal matrix. The matrices fulfill the following equation:

$$M = U\Sigma V^T$$

The individual matrices generated by the SVD act als two rotations and one distortion. As visible in image 4.3, the unit circle $\vec{x}$ first gets rotated by V^T that the distortion is in the directions of the coordinate system. After applying the distortion Σ to the rotated circle $\vec{y}_1$, the intermediate ellipse $\vec{y}_2$ gets rotated and mirrored to match $\vec{y}$.

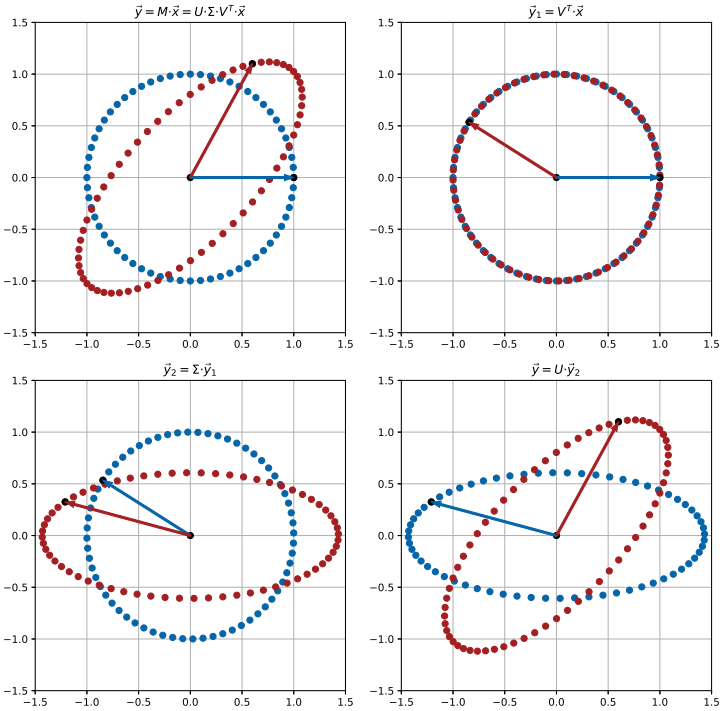

Figure 4.3: Singular Value Decomposition in 2D shown in the individual stages. Blue is before and red is after the applied transformation.

The following recipe[12] shows how to find the matrices of the singular value decomposition. The matrix M that is used in visualization 4.3 gets decomposed.

$$M = \begin{bmatrix} 0.6 & 0.9 \\ 1.1 & 0.2 \end{bmatrix}$$

In order to find U, the eigenvectors $\vec{v}_1$ and $\vec{v}_2$ of MM^T have to be calculated, that are directly filled in:

$$MM^T = \begin{bmatrix} 1.17 & 0.84 \\ 0.84 & 1.25 \end{bmatrix} \quad \vec{v}_1 = \begin{pmatrix} -0.6901 \\ -0.7237 \end{pmatrix} \quad \vec{v}_2 = \begin{pmatrix} -0.7237 \\ 0.6901 \end{pmatrix} \quad U = \begin{bmatrix} -0.6901 & -0.7237 \\ -0.7237 & 0.6901 \end{bmatrix}$$

Similarly to U, the eigenvectors of M^TM deliver V:

$$M^TM = \begin{bmatrix} 1.57 & 0.76 \\ 0.76 & 0.85 \end{bmatrix} \quad \vec{v}_1 = \begin{pmatrix} -0.8450 \\ 0.5347 \end{pmatrix} \quad \vec{v}_2 = \begin{pmatrix} -0.5347 \\ -0.8450 \end{pmatrix} \quad V = \begin{bmatrix} -0.8450 & -0.5347 \\ 0.5347 & -0.8450 \end{bmatrix}$$

Finally, the square-roots of the eigenvalues of either M^TM or MM^T are the diagonal values of Σ:

$$\Sigma = \begin{bmatrix} 1.4321 & 0 \\ 0 & 0.6075 \end{bmatrix}$$

4.2.4 Spatial coordinates and device coordinates

As the rotation of the camera head changes the coordinates of the measurement in respect to real-world coordinates, a convention helps carry out the calculations. For the sensors on the camera head, the coordinates are named a, b, and c, while the spatial coordinates are named x, y, and z. The coordinates can be transformed by the following formula, knowing the current orientation quaternion Q_{rot}.

$$Q_{rot} = \begin{pmatrix} r \\ u\mathbf{j} \\ v\mathbf{j} \\ w\mathbf{k} \end{pmatrix} \quad ; \quad \begin{pmatrix} 0 \\ x\mathbf{i} \\ y\mathbf{j} \\ z\mathbf{k} \end{pmatrix} = \begin{pmatrix} r \\ u\mathbf{i} \\ v\mathbf{j} \\ w\mathbf{k} \end{pmatrix} \begin{pmatrix} 0 \\ a\mathbf{i} \\ b\mathbf{j} \\ c\mathbf{k} \end{pmatrix} \begin{pmatrix} r \\ -u\mathbf{i} \\ -v\mathbf{j} \\ -w\mathbf{k} \end{pmatrix}$$

Section 4.2.2 describes how to carry out the mathematics of this quaternion multiplication.

4.3 Standard Vulkan Coordinate System

In Vulkan, every vertex coordinate of a 3D rendered object gets mapped to the nearest pixel in the viewport window. This vertex mapping is done in multiple steps from "local space" coordinates via "clip coordinates" towards "normalized device coordinates" to the "pixel coordinates". A 3D object is a group of vertex coordinates, described by a list three-dimensional vectors $\vec{v_v} = (x_v, y_v, z_v)$ (the subscript v denotes the "vertex coordinates"). These coordinates usually do not contain data regarding the whole object's scale, position, and rotation in 3D space - the coordinates are in the object's local space.

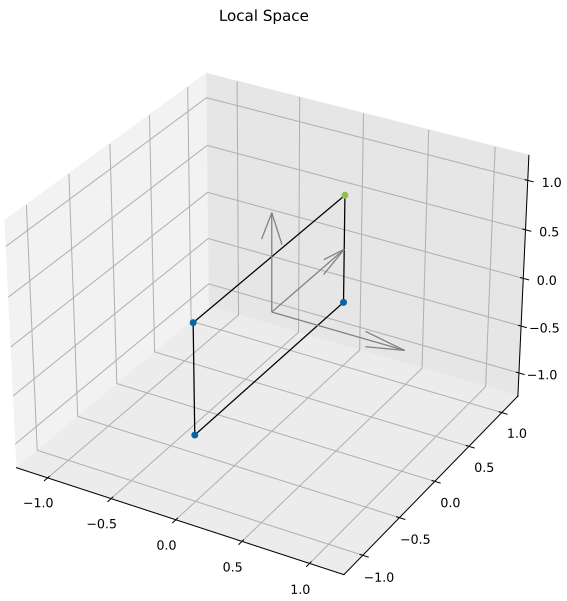

Figure 4.4: A Vulkan object consisting of four vertices in its local space. The green corner acts as an example.

Vulkan expects the output of the shader step to be in clip coordinates. The clip coordinates reside in the clip space. Clip coordinates are four-dimensional vectors $\overrightarrow{v_c} = (x_c, y_c, z_c, w_c)$ (the subscript c denotes the "clip coordinates") and the result of a matrix multiplication operation.

$$\vec{v_c} = A \cdot \vec{v_v}$$

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{M0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{M1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{M2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{M3,2} & a_{3,3} \end{bmatrix} \cdot \begin{pmatrix} x_v \\ y_v \\ z_v \\ 1 \end{pmatrix}$$

Generally, three combined matrix multiplications describe how a 3D object is rendered – the model matrix, the view matrix, and an added projection matrix. The model matrix (A_{Model}) defines the scale, rotation and position of the 3D object in the world space. The model matrix is a standard 4x4 rotation and translation matrix as explained in section 4.2. In the example, shown in the images 4.4 and 4.5, the rectangle is shifted on the x-axis.

$$A_{Model} = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The view matrix (A_{View}) is also a rotation and translation matrix, but describes the position and direction of the viewport camera inside the world space. Rotating the camera rotates the rendered virtual space, which indirectly moves and turns the models in the viewport. The linmath-library offers the "4x4_look_at" function that calculates the view matrix based on camera position, a viewing angle, and the "upwards" direction, as visualized in red in image 4.5. In the upper half of the image, the virtual camera is placed at the source, pointing towards the x-axis with the upwards direction facing alongside z. In the bottom part of the image, the camera position is shifted, and the camera is rotated slightly.

$$A_{View,top} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad ; \quad A_{View,bottom} = \begin{bmatrix} 0.266 & -0.963 & -0.036 & 0.036 \\ 0.266 & 0.037 & 0.963 & -0.963 \\ -0.927 & -0.266 & 0.266 & -0.266 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The projection matrix ($A_{Projection}$) is not a rotation and translation matrix besides the model and the view matrix. As its name suggests, the projection matrix reduces the vertex' 3d coordinates to the viewport plane by projecting them onto a virtual screen. The linmath-library offers the "4x4_perspective" function that calculates the projection matrix based on a given field of view angle. As the linmath-library was made for OpenGL, which uses a different alignment on the framebuffer, the element $A_{Projection}[1][1]$ needs to be multiplied by -1. In image 4.5, the effect of the projection matrix is shown with the black beams protruding out of the red dot. In the example, the following projection matrix is used.

$$A_{Projection} = \begin{bmatrix} 0.678 & 0 & 0 & 0 \\ 0 & -1.19175 & 0 & 0 \\ 0 & 0 & -1.0202 & -0.20202 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Within the vertex shader, these three matrices are chained to perform the desired transformation.

$$A = A_{Projection} \cdot A_{View} \cdot A_{Model}$$

Applying the example matrices to the green dot demonstrates the process of calculating the clip coordinates $\vec{v_c}$. The developer has to provide these coordinates to Vulkan as the output of the vertex shader:

$$\vec{v_v} = \begin{pmatrix} 0 \\ 1 \\ 0.5625 \\ 1 \end{pmatrix} \quad ; \quad \vec{v_{c,top}} = A_{top} \cdot \vec{v_v} = \begin{pmatrix} -0.678 \\ -0.670 \\ 1.838 \\ 2.0 \end{pmatrix} \quad ; \quad \vec{v_{c,bottom}} = A_{bottom} \cdot \vec{v_v} = \begin{pmatrix} -0.281 \\ -0.175 \\ 2.079 \\ 2.235 \end{pmatrix}$$

By division of the clip coordinate components x_c, y_c and z_c with w_c, Vulkan itself calculates the normalized device coordinates $\vec{v_{NDC}} = (x_{NDC}, y_{NDC}, z_{NDC})$.

$$\begin{pmatrix} x_{NDC} \\ y_{NDC} \\ z_{NDC} \end{pmatrix} = \begin{pmatrix} \frac{x_c}{w_c} \\ \frac{y_c}{w_c} \\ \frac{z_c}{w_c} \end{pmatrix}$$

In the example, the normalized device coordinates are:

$$\vec{v_{NDC,top}} = \begin{pmatrix} -0.339 \\ -0.335 \\ 0.919 \end{pmatrix} \quad ; \quad \vec{v_{NDC,bottom}} = \begin{pmatrix} -0.126 \\ -0.078 \\ 0.930 \end{pmatrix}$$

The transformation to the pixel coordinates are also done by Vulkan without requiring any action by the developer. Only the x and y parts of the normalized device coordinates define, where a pixel is rendered.

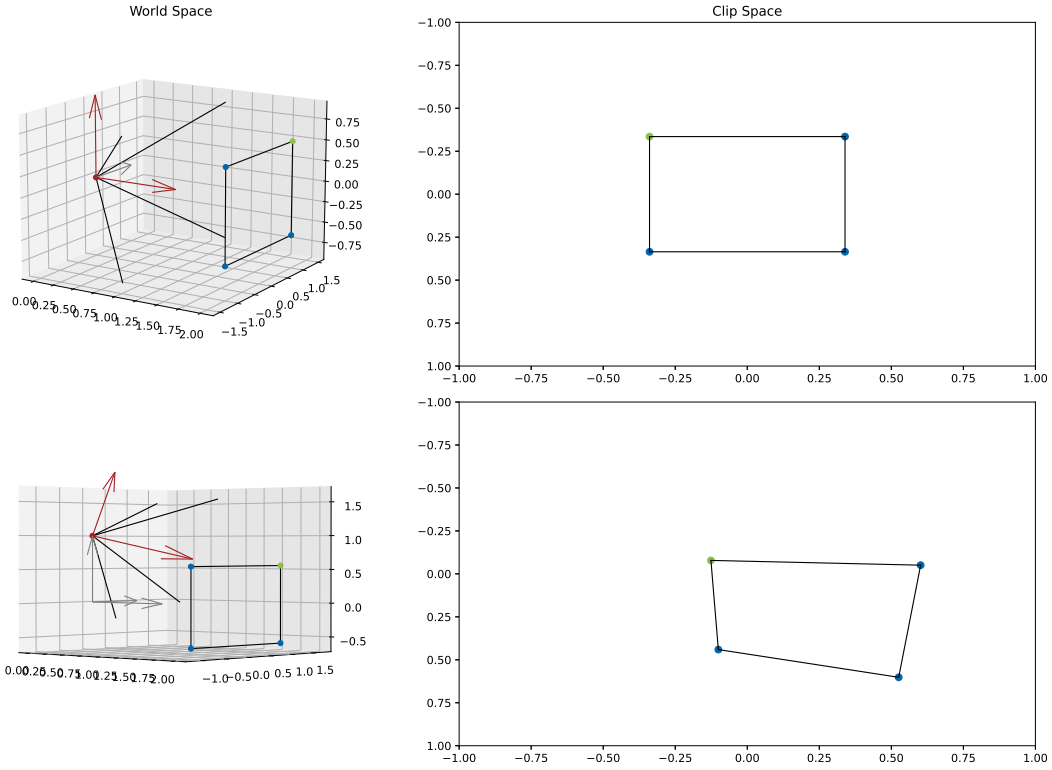

Figure 4.5: The rectangle is placed with the model-matrix, while the view- and projection-matrices determine the setup of the virtual camera. The green corner point acts as the example. Note the inverted y-axis in clip space.

The z part tells the depth, on how far the vertex is "inside the monitor". Video games use this information to not render objects that are too far away from the observer. On the viewport surface, the point (0/0) is located in the center. Top left is $(-1/-1)$, top right is $(1/-1)$, bottom left is $(-1/1)$ and bottom right is $(1/1)$.
If a vertex falls outside of the range ± 1 in the clip coordinates, it is not rendered or "clipped away", hence the name of the coordinate space.

4.4 Camera Calibration

An uncalibrated camera image often has lens distortion, warping a rectangle into a pillow or barrel shape and making areas appear closer in certain parts of the image. These distortions are named radial and tangential distortion and are induced by the camera lens.
According to OpenCV [13], radial distiortion can be modeled as:

$$x_{distorted} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \qquad y_{distorted} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Tangential distortion is modeled as [13]:

$$x_{distorted} = x + (2p_1 xy + p_2(r^2 + 2x^2)) \qquad y_{distorted} = y + (p_1(r^2 + 2y^2) + 2p_2 xy)$$

In these equations, r is the euclidian distance between the distorted image point and the distortion center. [13]

$$r = \sqrt{(x_{distorted} - x_{center})^2 + (y_{distorted} - y_{center})^2}$$

Therefore, for the lens distortions, the five coefficients k_1, k_2, k_3, p_1 and p_2 are needed. In addition, the effect of the focal length f and the optical center c get expressed as a 3x3 matrix. [13]

$$A_{Camera} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

OpenCV itself provides a script which estimates these values based on multiple photographs of chess boards. Applying these corrections leads to a smaller image as parts near the border get cut off.

Figure 4.6: Before correction

Figure 4.7: After correction

Figure 4.8: Camera correction demonstrated at the grayscale image of the ToF camera.

The implementation in CUDA is done by storing the pixel coordinates of the uncorrected image for each pixel in the corrected image. This data is loaded into a CUDA allocated memory area and allows mapping the coordinates for the correction without complex caluclations as visualized in image 4.9.

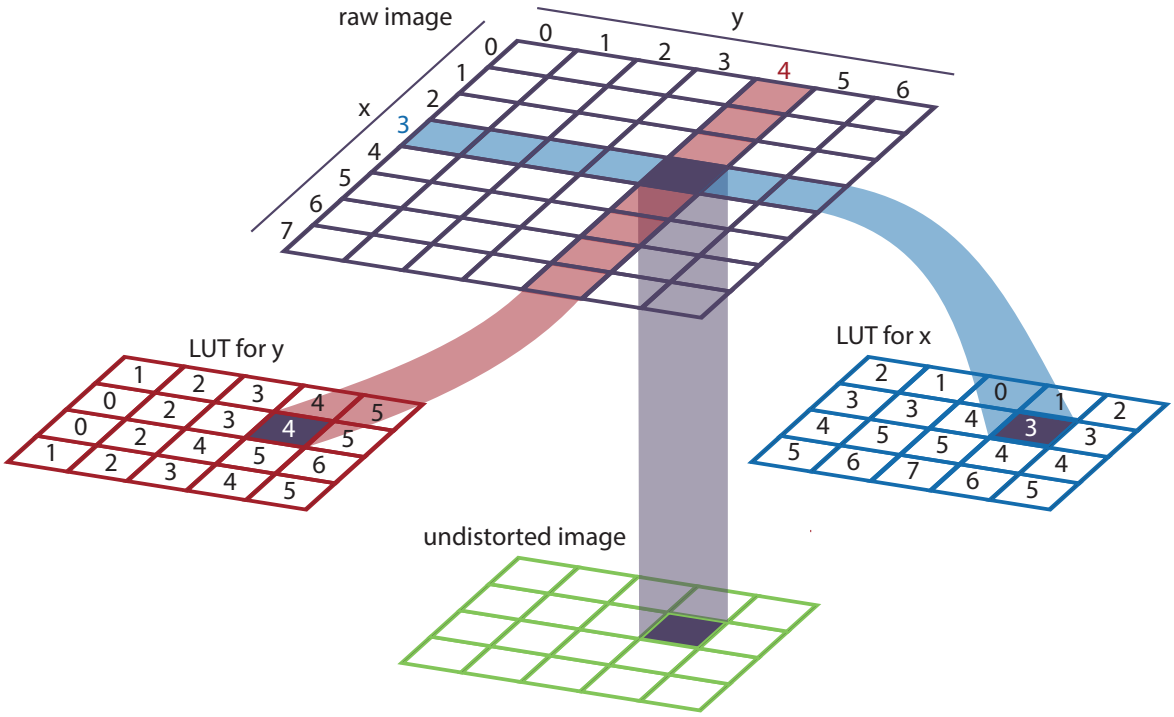

Figure 4.9: Storing the final calibration values in two lookup tables keeps the implementation in CUDA simple.

4.5 Kalman filter

The Kalman filter is an adaptive filter realized with a predictor-corrector algorithm that uses a model and known gaussian uncertainties of different sensors to estimate a value. Additionally to sensor data, the Kalman filter can adjust the estimate based on known forces – for example, the rotor speed of an RC drone. In an RC drone, a GPS position sensor, an accelerometer, and proximity sensors are fused with the rotor speeds to estimate the current motion with increased accuracy.
In this thesis, motion without a known input needs to be measured, having sensors for acceleration, velocity, and rotation. The following example demonstrates the position, velocity, and acceleration in one dimension. A fifth-order spline is used for the position data from which velocity and acceleration are derived to simulate the sensor data. Adding Gaussian noise to the simulated velocity and acceleration data reflects reality and allows demonstrating the Kalman filter.

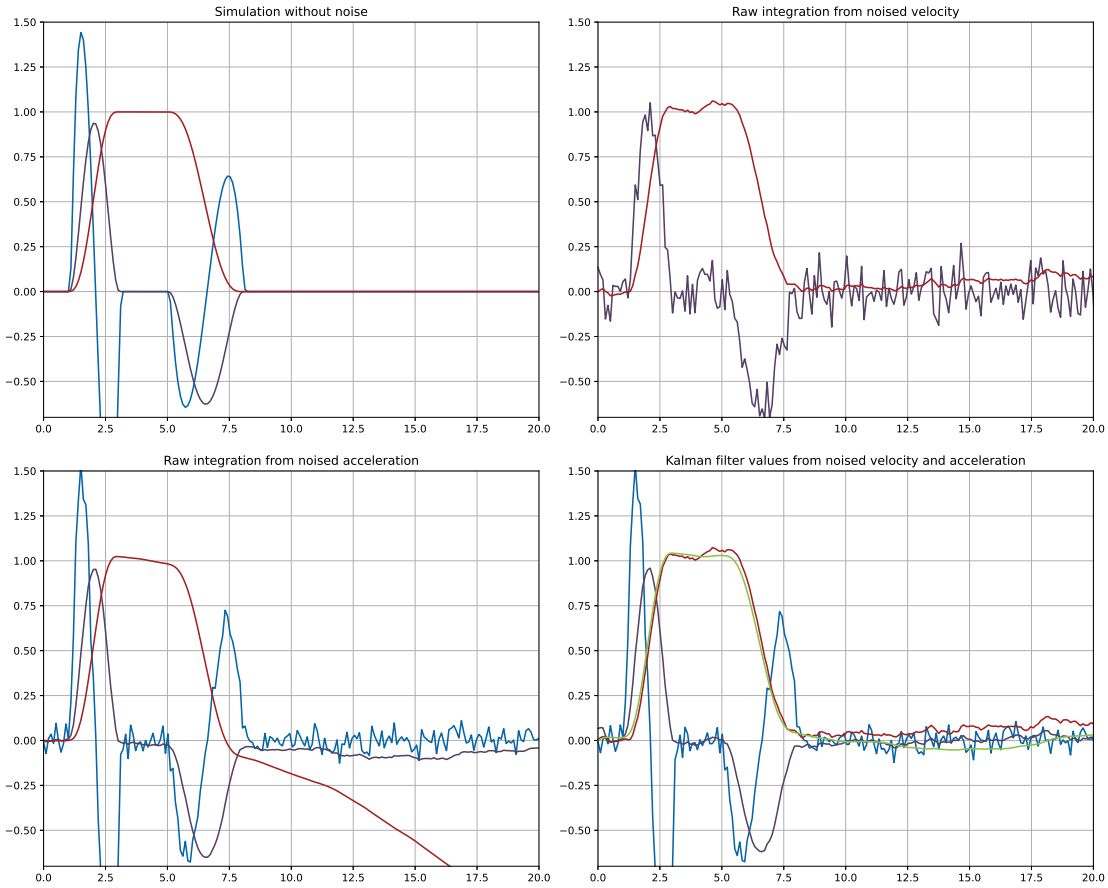

Figure 4.10: The Kalman filter demonstrated. Blue is the acceleration, purple the velocity and red the position. Raw integration of the velocity shows a jagged output of the position estimation. Raw double-integration of the acceleration leads to the position to drift away. The position estimate of the Kalman filter itself follows the jagged line of the raw velocity integration, but the estimated velocity is closer to the true value. In green, the integration of the estimated velocity shows a better approximation of the true value.

For each new data point, the Kalman filter predicts the next system state based on old data using the model matrix F. If a known input $\vec{u}$ would act on the system, the input matrix B would model that. Without a known input, B and $\vec{u}$ are omitted.The system state $\vec{x}$ transitions from the prior state $k-1$ to the prediction $k|k-1$.

$$\vec{x}_{k|k-1} = F \cdot \vec{x}_{k-1} + B \cdot \vec{u} \quad ; \quad \begin{pmatrix} p \\ v \\ a \end{pmatrix}_{k|k-1} = \begin{bmatrix} 1 & \Delta t & \frac{\Delta t^2}{2} \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} p \\ v \\ a \end{pmatrix}_{k-1}$$

In the example, the system matrix F reflects the derivations of the position p. The first time-derivative of p is the velocity v and the second time-derivative is the acceleration a.
In addition, the following formula predicts the covariance matrix of the errors P with the model matrix F and the gaussian process noise Q:

$$P_{k|k-1} = F \cdot P_{k-1} \cdot F^T + Q$$

Assuming that an external force $\vec{f}$ acts on the example system, constant during the sampling intervals Δt. The external force influences the acceleration, the velocity, and ultimately the position proportional to the input vector G.

The process noise matrix Q combines the input vector G and the standard deviation of the external force σ_f. The standard deviation of the external force σ_f is set to 1, as expected accelerations are much smaller than the gravitational acceleration. The process noise model is named the "piecewise white noise model."

$$G = \begin{pmatrix} \frac{\Delta t^2}{2} \\ \Delta t \\ 1 \end{pmatrix} \quad ; \quad Q = G \cdot G^T \cdot \sigma_f = \begin{bmatrix} \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} & \frac{\Delta t^2}{2} \\ \frac{\Delta t^3}{2} & \Delta t^2 & \Delta t \\ \frac{\Delta t^2}{2} & \Delta t & 1 \end{bmatrix} \cdot 1$$

The covariance matrix of the errors P shows that the position estimate will worsen over time. The error propagation causes this error to steadily increase without receiving a direct correction from any sensor. The system would need the possibility to determine the position to avoid long-term drift; otherwise, the error increases over time. On the other values - where sensor values are present - the covariance matrix of errors P converges towards a constant error.

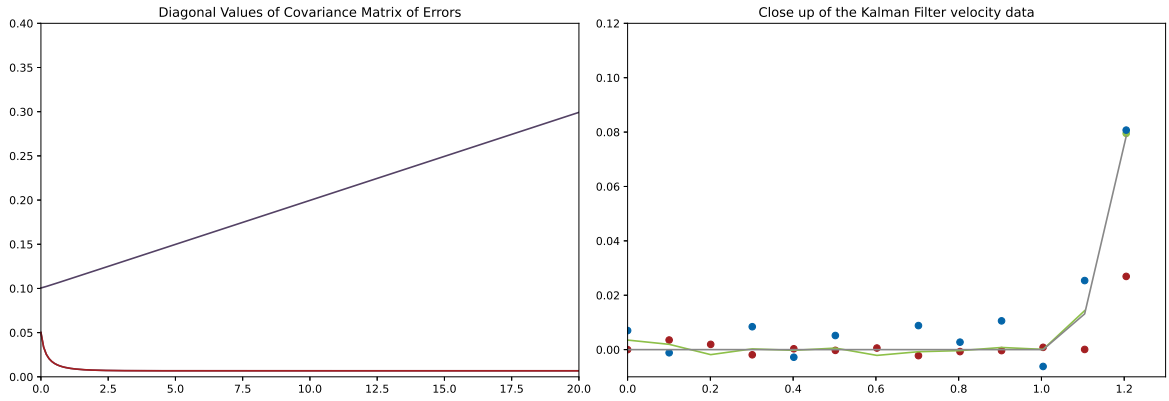

Figure 4.11: On the left, the diagonal values of the covariance matrix of the errors are visualized against each other. The errors for velocity and acceleration converge towards a low value, while the error for the position increases.
On the right, a close-up of the internal workings of the Kalman filter. The blue dots are the raw measurements, the red dots are the corresponding predictions and plotted in green is the output value of the Kalman filter. In contrast, the true velocity is plotted in grey.

With the system state $\vec{x}_{k|k-1}$ and the covariance matrix $P_{k|k-1}$ being predicted, the following steps describe the correction part of the Kalman iteration. The observation matrix H determines which sensor will influence which entry of the system state vector $\vec{x}$. For each correction step, a Kalman Gain matrix K_k determines the optimal correction based on the current covariance of errors P and the measurement noise R.

$$K_k = P_{k|k-1} \cdot H^T (H \cdot P_{k|k-1} \cdot H^T + R)^{-1} \quad ; \quad H = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

In the example, H is a 2×3 matrix, because there are only two measurements versus the three dimensional system state. Therfore, the matrix inversion for the bracket in the formula above is only of dimension 2×2.
With the Kalman gain present, the system is ready to correct both the system state vector $\vec{x}$ and the covariance matrix of errors P. The vector $\vec{z}_k$ contains the new measurement - in the Kalman filter often named, the new observation.

$$x_k = \vec{x}_{k|k-1} + K_k(\vec{z}_k - H \cdot \vec{x}_{k|k-1}) \quad ; \quad P_k = (I - K_k \cdot H) P_{k|k-1}$$

For multiple sensors influencing the same entry of the system state vector $\vec{x}$, multiple correction steps may be chained after each other.

5 Implementation

This chapter describes the implementation and the connections of the individual components of the augmented reality system.

5.1 Hardware

The following section describes the evaluation and setup of the hardware used for the camera head and the processing system.

5.1.1 Evaluation of ToF Camera

Having no prior experience with time-of-flight (ToF) cameras, a suitable device had to be evaluated first. Cost and availability have been the main factors in the evaluation, with it being directly CSI-2 connected being a big plus.
Internet research has shown a handful of different sensors powering multiple products of diverse manufacturers in varying price ranges.

Sony DepthSense IMX556PLR
The Sony IMX556PLR ToF Sensor offers a resolution of 640 x 480 pixels at 30 frames per second and is used by Basler, Lucid Vision Labs, and DephtEye, primarily for Gigabit Ethernet cameras. The IMX556PLR seems to be the most capable ToF sensor freely available in off-the-shelf products at the time.
The technically most compelling product for this thesis would have been the Helios Flex by Lucid Vision Labs, which directly connects via CSI-2 and is sold specifically for use on an Nvidia Jetson TX2 Developer Kit. It features a maximum range of 6 meters for depth measurement and accuracy of ±10mm. Although with 749 US Dollars, the Helios Flex is relatively expensive and unsuitable for the thesis because of an unknown lead time.
The other cameras using the IMX556PLR sensor are even more costly and also have uncertain lead times.

Infineon REAL3 IRS1125
The Infineon IRS1125 ToF Sensor is available in a USB-based development kit by pmdtec – an integrator for ToF technology into smartphones – and on the affordable PiEye Nimbus 3D camera, which is sold as a Raspberry Pi accessory.
The sensor features a resolution of 352x288 pixels at 30 frames per second in the variant IRS1125A. The variant IRS1125C – used by pmdtec – allows 60 frames per second. The pmdtec development kit is tuned for measuring up to 6 meters, while the PiEye camera is limited to 5 meters.
While both camera systems are readily available to be shipped, the pmdtec pico monstar costs about 1500 US Dollars; in contrast, the PiEye Nimbus costs only 230 Euros. The PiEye company advertises its camera with open source software to embed it into the Raspberry Pi ecosystem. However, further investigation has shown that the middleware – the library managing the camera's settings and connecting to the video4linux2 framework – is still under NDA with Infineon.
With a lightweight TCP/IP protocol, the PiEye module is suitable with a Raspberry Pi, acting as a Gigabit Ethernet camera. The PiEye Nimbus is the module of choice for this thesis – the availability, the price, and the specifications are all reasonable. The options to reverse engineer the middleware or sign an individual NDA with Infineon have been kept open initially but were not necessary as the Gigabit Ethernet implementation worked well enough.

Other image sensors

During the internet research, the Texas Instruments OPT8241 has shown up. TI declared the chip obsolete – the chip itself was still available, but the development kit was sold out. With 320 times 240 pixels and an advertised range of 4 meters, the OPT8241 is inferior to the chosen PiEye Nimbus. In addition, the Terabee 3Dcam features a custom sensor with a resolution of 80x60 pixels and 4 meters range. It is attached by USB 2.0 and costs 250 Euros. Due to the low resolution, the Terabee 3Dcam is also inferior to the PiEye Nimbus.

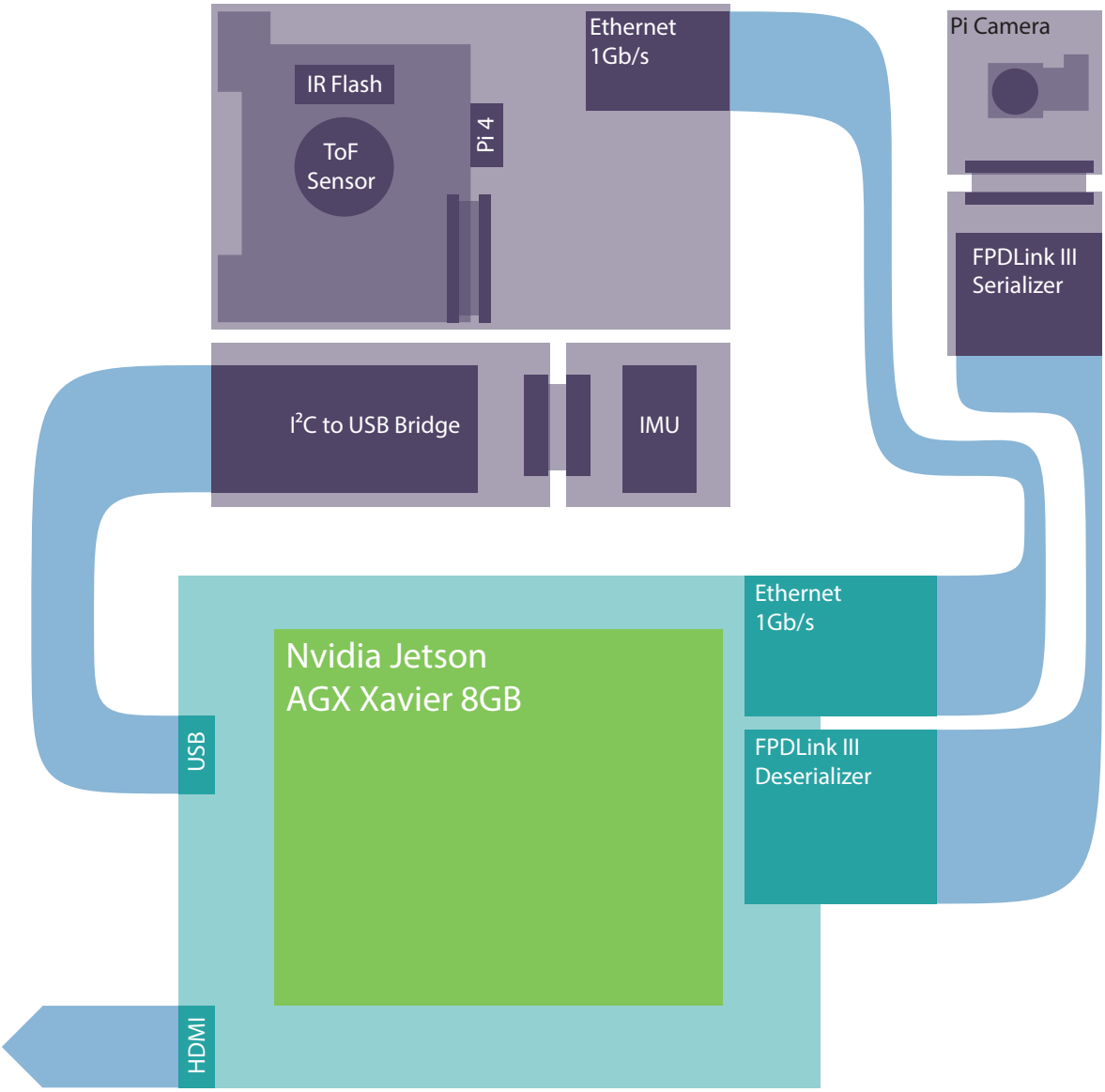

Figure 5.1: Hardware overview. The purple objects belong to the camera head, which is connected to the processing system in cyan.

5.1.2 Camera Head

The camera head, shown in image 5.1, contains the PiEye Nimbus ToF camera, mounted on a Raspberry Pi 4B, a Bosch BMI160 IMU, attached to a USB to I²C bridge, and a standard Raspberry Pi camera v2.1 which is connected to the processing system via an FPDLink module.
All is held together by a plywood structure glued onto a tripod-baseplate, shown in figure 5.4.

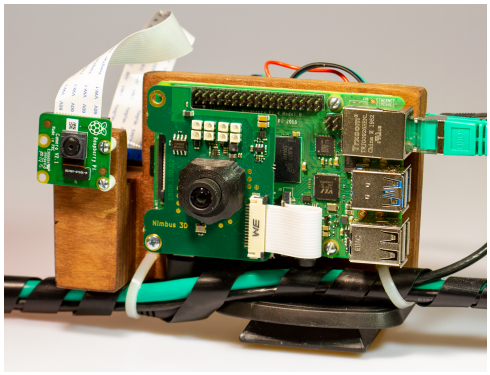

Figure 5.2: Frontside

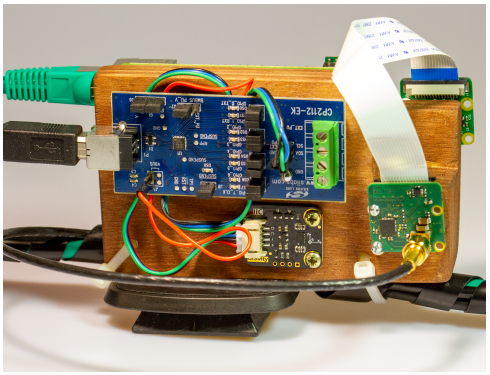

Figure 5.3: Backside

Figure 5.4: The camera head consisting of the ToF camera and the Raspberry Pi camera on the front side, and the IMU, its I²C to USB bridge and the FPDLink III Serializer on the backside.

The IMU could have been attached directly to the Raspberry Pi, but at the time of the implementation, the decision on how to connect the ToF camera was not taken yet. Therefore, it was not certain that a Raspberry Pi would be mounted at the camera head. Four individual cables connect to the different components on the camera head: A USB-C power cable and an RJ45 Ethernet cable for the Raspberry Pi, a USB 2.0 cable for the USB to I²C bridge, and an FPDLink coaxial cable for the Raspberry Pi Camera.

5.1.3 Processing System

An Nvidia Jetson Xavier AGX in the 8GB version carries out the processing, rendering, and data acquisition. An Anyvision baseboard - shown in image 5.5 - carries the Nvidia Jetson module and allows the direct attachment of the used data cables, thanks to its modular design. [14]
The Nvidia Jetson Xavier AGX 8GB offers a 6-core ARM v8.2 64bit CPU, a GPU with 384 Volta cores and 48 Tensor cores. A 256 bit wide link offers 85GB/s bandwith to the 8GB unified LPDDR4x RAM.
A TCP/IP server application on the Raspberry Pi powering the PiEye ToF camera, to which the processing system connects, serves the necessary ToF camera data in a frame-based protocol. The USB to I²C bridge gets loaded as a standard I²C device by the Linux on the processing system. The IMU is then directly configured and polled by the userspace software on the processing system without an additional driver. The Raspberry Pi camera is attached via FPDLink and integrated as a video4linux2 device. The capturing is done with FFMPEG and the debayering in CUDA. [15]

5.1.4 Video inputs

The Sony IMX 219 based Raspberry Pi Camera v2 features a color image with an 8 megapixel resolution for single images, 1080p with 30 frames per second, 720p with 60 frames per second or 480p with 90 frames per second. [16] In the 1080p mode, the Raspberry Pi Camera v2 crops the sensor, using only a subsection of the field of view. The camera is run in the 720p mode, as it uses pixel-binning and utilizes the whole sensor size and offers the full field of view.
The Raspberry Pi Camera v2 is a Mipi CSI2 attached camera module whichs cable length got extended by an FPDLink serializer and deserializer.

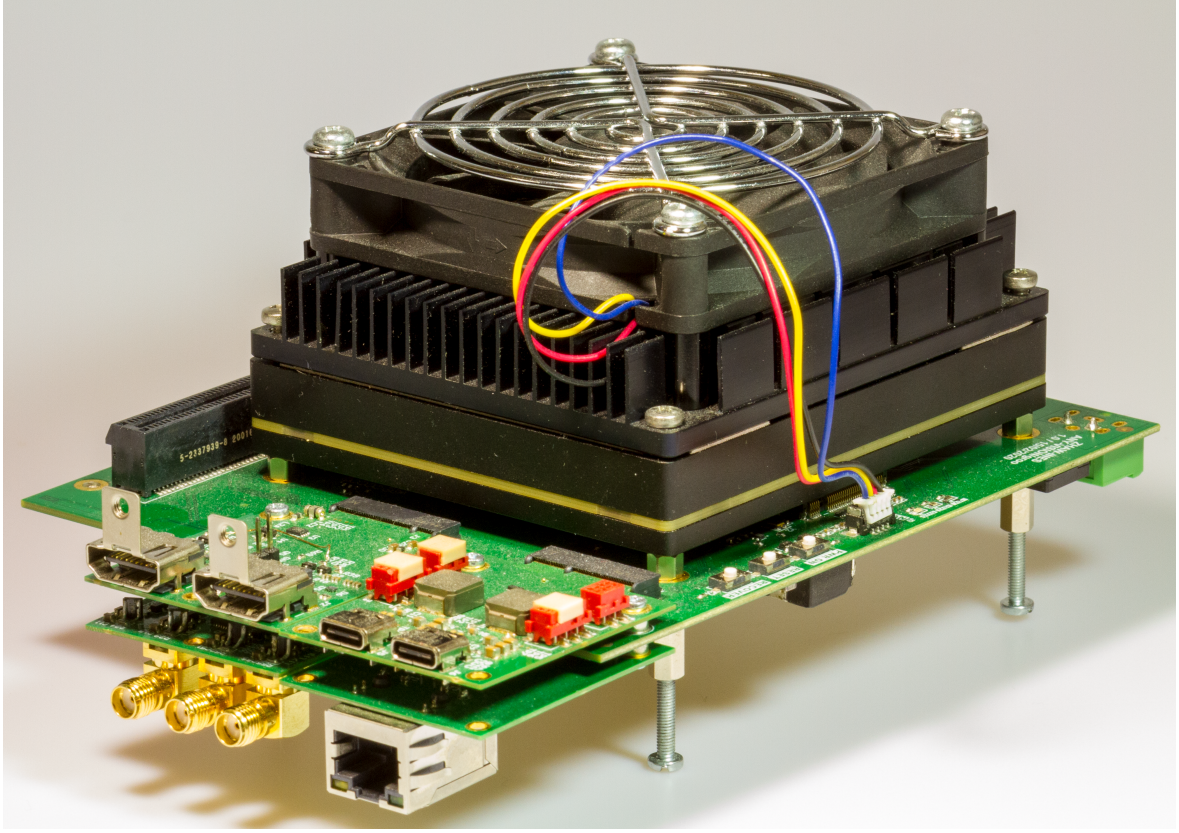

Figure 5.5: The Nvidia Jetson Xavier AGX on the Anyvision Baseboard.

The PiEye ToF camera is based on the Infineon REAL3 IRS1125A sensor which has a resolution of 352 x 288 pixels[17]. The ToF Camera is paired with infrared LED flashes to measure the time of flight of the light emitted and has an intended measurement range from 10 centimeters up to five meters. The entire ToF software stack runs on a Raspberry Pi that sends its video data via Ethernet to the augmented reality system.

5.1.5 Unified Memory

In a standard desktop computer, the GPU and the CPU have separate memory. It is possible to expose GPU memory to the host CPU, but this has numerous downsides. State-of-the-art desktop GPUs connect via PCIe 4.0 to the host system, which induces latency. The measured round-trip latency of PCIe 3.0 is around 900ns[18], which also varies with traffic on the system. In comparison, a DDR4 SDRAM access has a latency of around 10-15ns[19]. Disabling the caching on the memory - as required for concurrent memory access - worsens the latency problem. Therefore, the optimal way to share data between the CPU and the GPU in a standard desktop computer is to copy the memory from the host to the device and vice versa.
On a system with unified memory - like the used Nvidia Jetson AGX Xavier - both the CPU and the GPU have direct access to the same memory. Latency information for the LPDDR4 SDRAM on the Xavier is unavailable, but it is likely in the same range as DDR4 SDRAM. CUDA offers functions to allocate memory available for both CPU and GPU with individual memory pointers. This allocation method disables caching; memory only used by either the GPU or the CPU should be allocated traditionally.

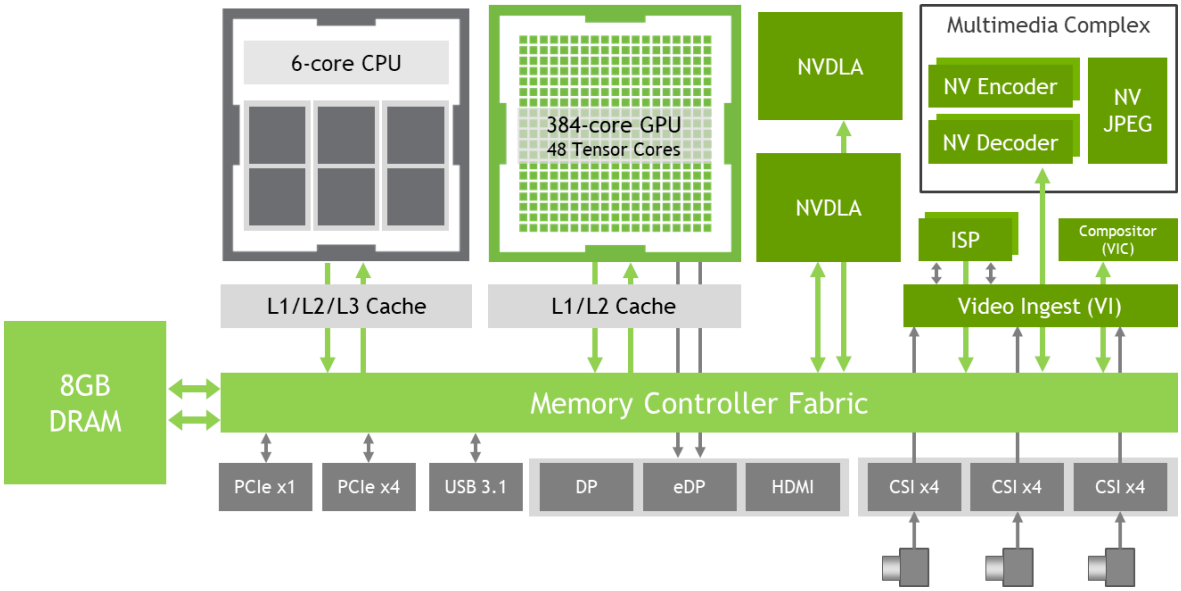

Figure 5.6: Block diagram of the technically equivalent Jetson Xavier NX. Note the shared DRAM and the separated caches for CPU and GPU. Copyright by Nvidia

5.2 Software Architecture

A single application needs to gather the data from the different sensors and cameras, apply the processing and render the result to an attached monitor. Within this application, various subsystems run in separate threads at different speeds. As visible in image 5.8, the Vulkan framework, developed for a former thesis, is the skeleton of this application, to which additional C++ and CUDA modules got attached. Vulkan itself renders the 3D object – a rectangle named "projected image" – into the 2D viewfinder image of the main camera, as described in section 5.7. The image processing and the calculation of rotation and translation are mainly performed on the GPU, while the Kalman filtering is done on the CPU.
Due to the multithreaded nature of the system, and as multiple threads access the GPU simultaneously, the GPU access got grouped into different CUDA streams.
A separate application runs on the Raspberry Pi, serving a TCP/IP connection for streaming the ToF data.

TODO

Figure 5.7: Data flow for one single frame.

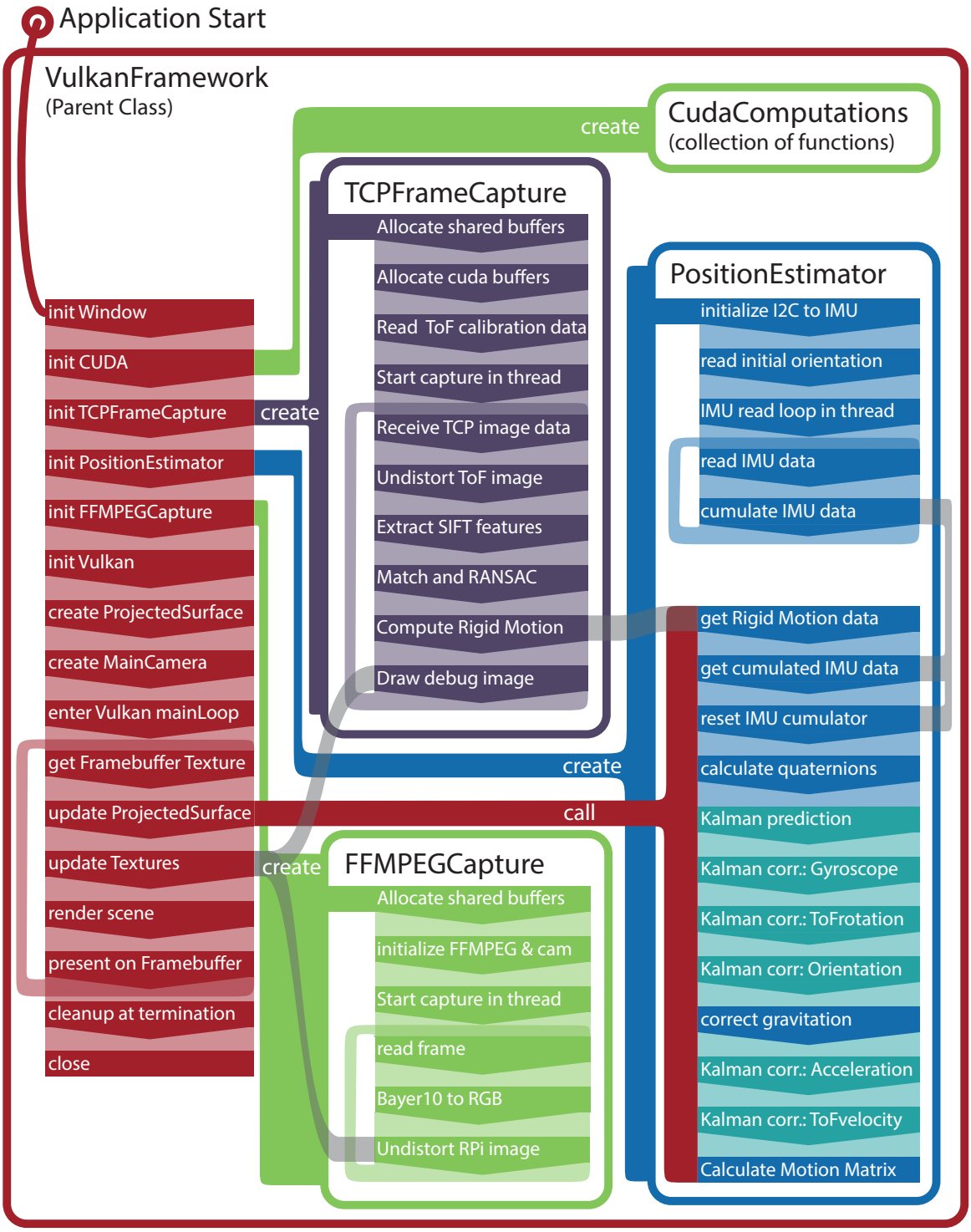

Figure 5.8: Software overview. The routine inside the Vulkan Framework creates four child classes.

5.3 Motion estimation from ToF camera

The following section describes the extraction of motion - rotational and translational velocity - from the ToF camera data. The purple section in image 5.8 shows the location of the algorithms within

the software.

5.3.1 ToF Camera calibration

On the ToF Camera, two parts need to be calibrated: The optics, as described in section 4.4, and the distance measurement. The ToF camera has a barrel type distortion that is corrected by the camera calibration algorithm implemented in OpenCV [13] and shown in image 4.8. As the process of lens correction cuts off parts of the image, the image size gets lowered to 265 x 205 pixels.
For the distance measurement, first the radial distances need to be flattened as described in the last part of section 4.1. As the angle α is not known for each pixel, a reference measurement provides the necessary information. To reduce noise, 19 images of the same flat wall has been taken, smoothed by a gaussian and averaged onto one reference image I_{Ref}. The wireframe image in figure 5.9 shows the curvature of the reference image. The rounding at the edges is an arfifact of the gaussian smoothing.

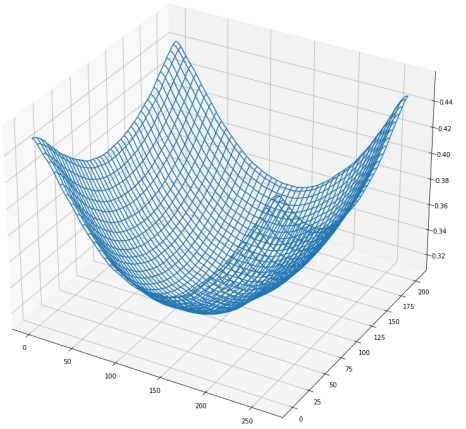

Figure 5.9: Wireframe rendering of the reference image I_{Ref} provided by the ToF camera

Dividing the minimum value of this reference image I_{Ref} with every pixel value generates a map of $\cos\alpha$ named I_{cos}.

$$I_{cos} = \frac{\min(I_{Ref})}{I_{Ref}}$$

Pixel by pixel multiplication of any other image I_{Any} with I_{cos} will correct the influence of the radial measurement as shown in image 5.10.

$$I_{Corr} = I_{cos} \cdot I_{Any}$$

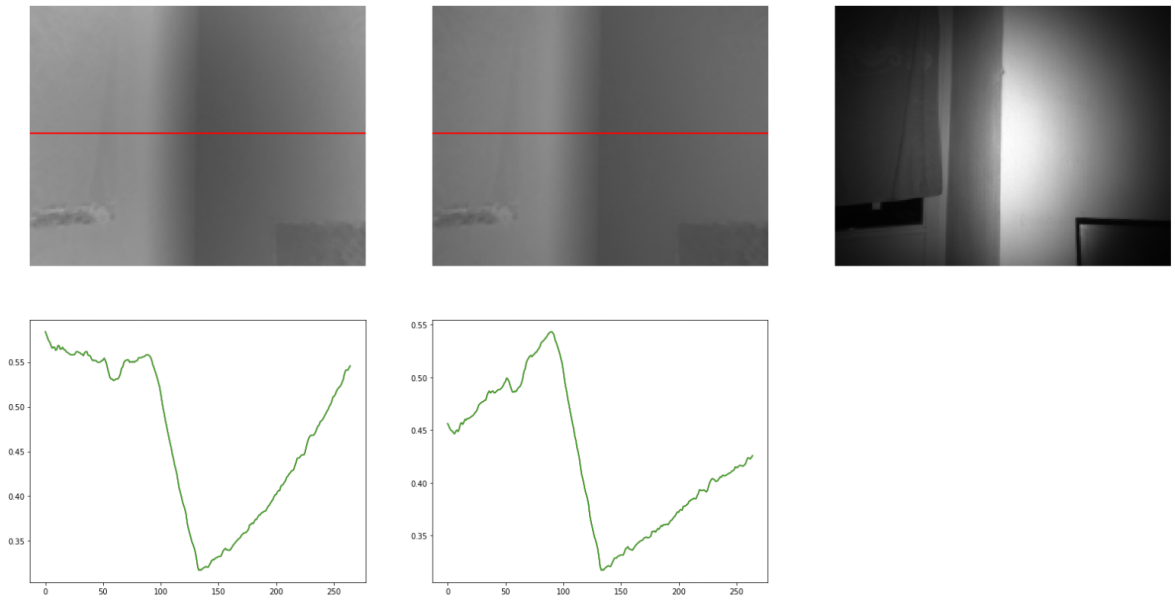

Figure 5.10: Left, the uncorrected ToF image I_{Any}, in the middle the corrected image I_{Corr} and on the right, the infrared grayscale image of the scene. To make the effect more apparent, the brightness accross the red lines have been plotted.

To apply this calibration in CUDA, a file has been generated which holds the I_{Cos} values for each pixel. The application reads the file at initialization and keeps it stored in a Cuda allocated memory area.

5.3.2 SIFT feature extraction

The PiEye Nimbus ToF camera generates three different image channels for each picture taken: The depth map, the confidence, and the greyscale infrared image. By correcting the lens distortion as described in chapter 4.4, straight lines in the real world get projected as straight lines on the images. By further correcting the characteristic of the ToF camera to measure radial distances, as described in section 4.1, flat surfaces in the real world are also flat on the depth map. The implementations of both corrections are explained in section 5.3.1.
The points of the point clouds need to be matched to estimate rotation and translation between two consecutive ToF images. The CudaSift library [20] [21] extracts SIFT [22] features on each greyscale infrared image the ToF camera sends k, which are then brute-force matched with the features of the prior image $k-1$, as visualized in image 5.11. SIFT features were chosen because of the author's prior experience, and it is a well-known feature extraction algorithm whose patent expired. [23]

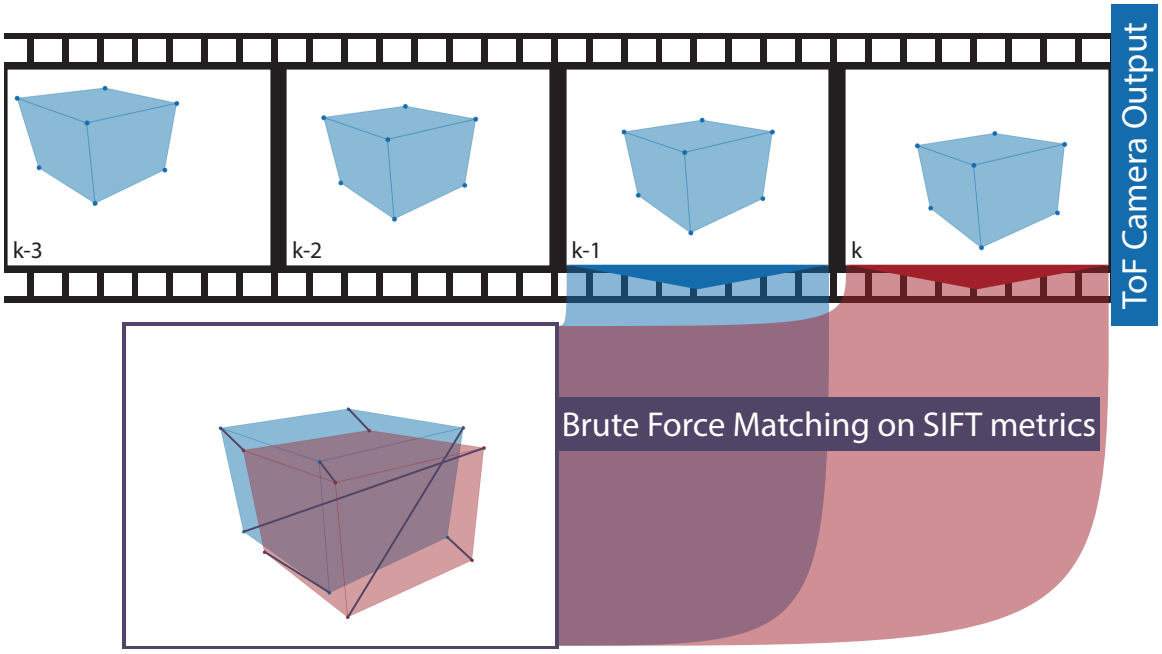

Figure 5.11: First step of ToF motion estimation: Extract SIFT features and brute-force match with prior image. Note that the brute-force matcher also generates false matches.

Each feature coordinate on the picture gets mapped to the 3D space to generate the point cloud. The lens projects objects within the space of a pyramid onto the sensor, which leads to the following coordinate mapping. The coordinate transformation is visualized in image 5.12. Please note the coordinate convention described in section 4.2.4. a, b and c being the 3D coordinates, u and v being the coordinates on the image and d being the value of the ToF depth image on position (u,v). f denotes a virtual focal length, merging the camera's field of view (viewing angle α) and the image resolution in one number.

$$a = d \qquad b = \frac{v}{f} \cdot x \qquad c = \frac{u}{f} \cdot x \qquad f = \frac{\frac{u_{max}}{2}}{tan(\frac{\alpha}{2})}$$

The PiEye Nimbus ToF camera has an advertised viewing angle of $1.152rad$ horizontally and $0.942rad$ vertically. Combined with an image resolution of $352x288px$, the formula for f outputs 271 for the horizontal and 282 for the vertical case. The chosen value is: $f = 280$.

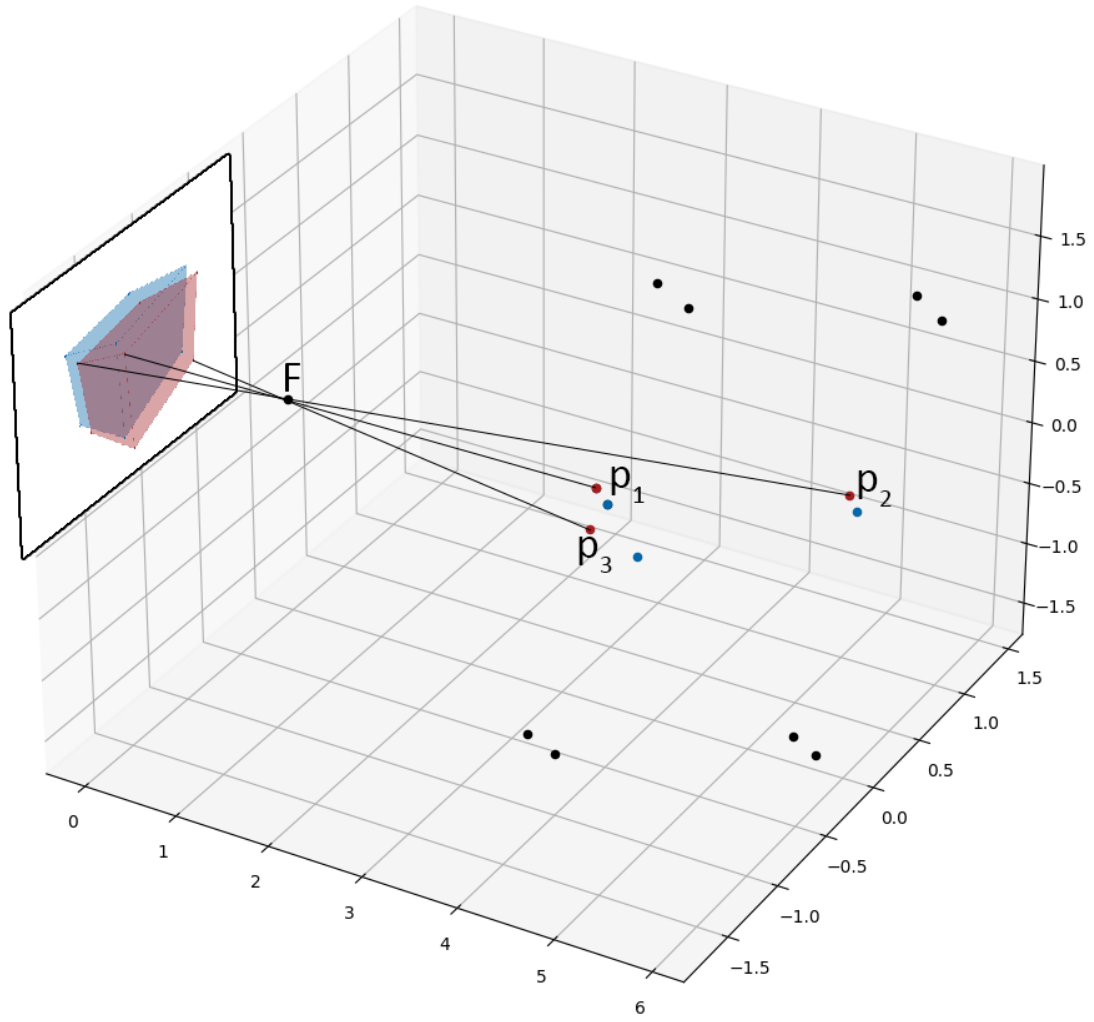

Figure 5.12: Second step of ToF motion estimation: Map features into 3D space

5.3.3 Find rotation and translation

Calculating the rotation and translation of one point cloud P_{k-1} to another point cloud P_k is possible with at least three correctly matched point pairs. Following the recipe from a note from the ETH Zurich, also containing the proof, the three points must be centered. The method described in the ETH note also adds the possibility to add weights to individual data points. Upper case P denotes a point cloud, lower case p denotes a single point in that cloud. The number of matched point pairs in the following formulae is n, in the example $n = 3$, i is the index of the single point in the point cloud and k is the image frame number from which the point cloud got extracted. The centroids for both point groups are:

$$\vec{c}_k = \frac{\sum_{i=1}^{n} \vec{p}_{i,k}}{n} \qquad \vec{c}_{k-1} = \frac{\sum_{i=1}^{n} \vec{p}_{i,k-1}}{n}$$

For example visualized in image 5.13 and with the following calculation:

$$\vec{c}_k = \begin{pmatrix} 4.421 \\ -0.154 \\ 0.223 \end{pmatrix} = \frac{1}{3} \cdot \left(\begin{pmatrix} 3.314 \\ 0.043 \\ -0.037 \end{pmatrix} + \begin{pmatrix} 4.852 \\ 1.198 \\ -0.586 \end{pmatrix} + \begin{pmatrix} 5.098 \\ -1.704 \\ 1.291 \end{pmatrix} \right)$$

$$\vec{c}_{k-1} = \begin{pmatrix} 4.433 \\ 0.059 \\ -0.090 \end{pmatrix} = \frac{1}{3} \cdot \left(\begin{pmatrix} 3.298 \\ 0.177 \\ -0.271 \end{pmatrix} + \begin{pmatrix} 4.709 \\ 1.436 \\ -0.924 \end{pmatrix} + \begin{pmatrix} 5.291 \\ -1.436 \\ 0.924 \end{pmatrix} \right)$$

Subtraction of the centroid vectors from the individual points in the respective point cloud P generates the centered point clouds Q. In an ideal case, a rotation matrix alone can transform one centered point cloud into the other.

$$\vec{q}_{i,k} = \vec{p}_{i,k} - \vec{c}_k \qquad \vec{q}_{i,k-1} = \vec{p}_{i,k-1} - \vec{c}_{k-1} \qquad i = 1,2,3,...,n$$

In the example, the results are:

$$\vec{q}_{1,k} = \begin{pmatrix} -1.108 \\ 0.197 \\ -0.260 \end{pmatrix} \qquad \vec{q}_{2,k} = \begin{pmatrix} 0.431 \\ 1.352 \\ -0.808 \end{pmatrix} \qquad \vec{q}_{3,k} = \begin{pmatrix} 0.677 \\ -1.549 \\ 1.068 \end{pmatrix}$$

$$\vec{q}_{1,k-1} = \begin{pmatrix} -1.134 \\ 0.118 \\ -0.181 \end{pmatrix} \qquad \vec{q}_{2,k-1} = \begin{pmatrix} 0.276 \\ 1.377 \\ -0.833 \end{pmatrix} \qquad \vec{q}_{3,k-1} = \begin{pmatrix} 0.858 \\ -1.495 \\ 1.014 \end{pmatrix}$$

A matrix-multiplication of the point groups Q_k and Q_{k-1} generates the 3×3 covariance matrix S as shown in the following formula. The point groups are packed in matrix form, in the three point example, the point group matrices are of dimension 3×3. When using n points, the point group matrices are of dimension $n \times 3$, whose covariance matrix remains of dimension 3×3.

$$S = Q_{k-1} Q_k^T$$

In the example:

$$Q_k = \begin{bmatrix} -1.108 & 0.431 & 0.677 \\ 0.197 & 1.352 & -1.549 \\ -0.260 & -0.808 & 1.068 \end{bmatrix} \quad Q_{k-1} = \begin{bmatrix} -1.134 & 0.276 & 0.858 \\ 0.118 & 1.377 & -1.495 \\ -0.181 & -0.833 & 1.014 \end{bmatrix}$$

$$S = \begin{bmatrix} 1.956 & -1.180 & 0.989 \\ -0.549 & 4.201 & -2.741 \\ 0.528 & -2.734 & 1.805 \end{bmatrix}$$

The singular value decomposition (SVD), briefly explained in section 4.2.3, splits the covariance matrix S into three separate 3×3 matrices U, Σ, and V. As the SVD in this use case is always applied to matrices of dimension 3×3, the optimized variant [24] from GitHub [25] can be utilized.

$$S = U \Sigma V^T$$

In the example:

$$U = \begin{bmatrix} -0.301 & -0.950 & -0.080 \\ 0.796 & -0.297 & 0.528 \\ -0.525 & 0.096 & 0.846 \end{bmatrix} \quad \Sigma = \begin{bmatrix} 6.309 & 0 & 0 \\ 0 & 1.690 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad V = \begin{bmatrix} -0.207 & -0.973 & 0.102 \\ 0.814 & -0.229 & -0.534 \\ -0.543 & 0.028 & -0.839 \end{bmatrix}$$

The wanted rotation matrix then follows by calculating:

$$R = V \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & det(VU^T) \end{bmatrix} U^T$$

Without the correction of the calculation, using the term $det(VU^T)$ in the intermediate matrix, the method could generate a reflection instead of a rotation. This is numerically sound, but would not reflect the real world scenario. The determinant $det(VU^T)$ equals -1 in the case of a reflection, which can be used to flip the signs of the 3rd column of the rotation matrix. If the SVD directly generates a rotation, $det(VU^T)$ equals to 1, which transforms the intermediate matrix into the identity.
In the example the resulting rotation matrix R equals:

$$R = \begin{bmatrix} 0.995 & 0.071 & -0.071 \\ -0.071 & 0.998 & 0.002 \\ 0.071 & 0.002 & 0.998 \end{bmatrix}$$

The wanted translation is computed by applying the rotation to the centroid vectors.

$$\vec{t} = \vec{c}_k - R \cdot \vec{c}_{k-1}$$

In the example:

$$\vec{t} = \begin{pmatrix} 0 \\ 0.1 \\ 0 \end{pmatrix}$$

The rotation matrix R and the translation vector $\vec{t}$ fulfill the following equation, in which p are individual points of the point clouds P. If the number of points is $n = 3$, the calculation is exact, for point clouds with $n > 3$ points, the error E is the least-square error [11].

$$\vec{p}_{i,k} = R \cdot \vec{p}_{i,k-1} + \vec{t} + E$$

The estimated translation is only the translation between two frames. To estimate the velocity, the translation is divided by the sampling time.

5.3.4 3D Random Sample Consensus (RANSAC)

The motion estimation of the ToF camera relies on having good matches, which is not the case with the brute-force matcher, as the authors of the CudaSift library claim to have less than 50% accuracy on its brute-force matcher. [20] Low matching accuracy is a known problem in other fields of image processing - like panorama stitching - and is there often solved with an algorithm named RANSAC (random sample consensus).
These applications of the RANSAC algorithm work on two-dimensional images and are not suitable for three-dimensional point clouds; therefore, the RANSAC algorithm got extended.
The first step of the three-dimensional RANSAC is finding a proper rotation and translation from the brute-force matches. To find these transformations, each matched feature pair gets two other feature pairs randomly assigned. Using the method described in section 5.3.3, the corresponding rotations and translations are calculated on these groups of three feature pairs as visualized in image 5.13.

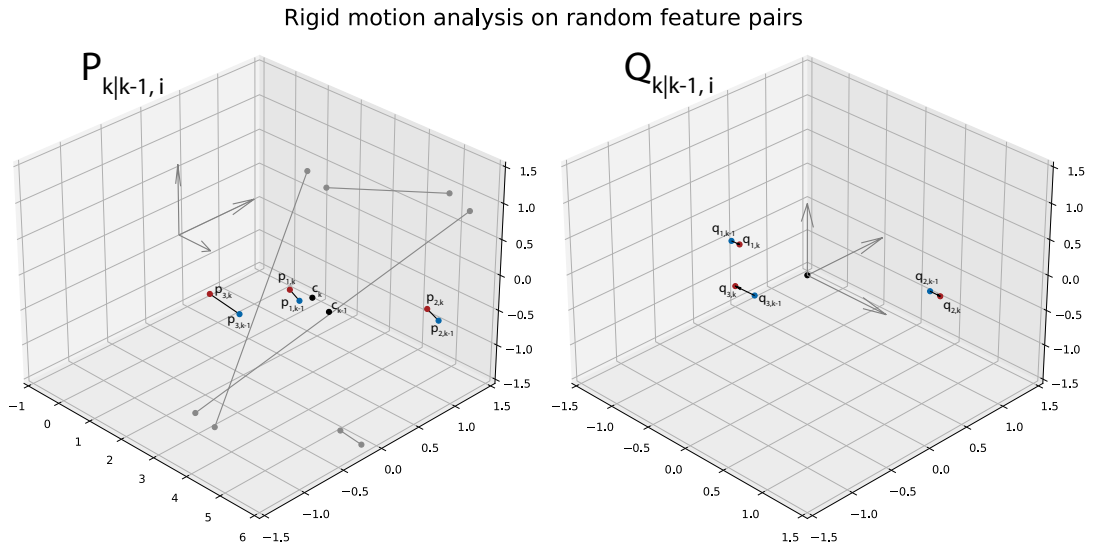

Figure 5.13: First step of the 3D RANSAC: The SVD method calculates the rotation and translation from three randomly assigned points. This step is performed in parallel on multiple groups of three. Each group P_i generates the rotation R_i and translation $\vec{t}_i$.

Each group's calculated rotation matrices and translation vectors get checked against all the other matched feature pairs. The data point from the previous image of the feature pair gets moved by the matrix-vector-pair and compared to the data point of the current picture. If the sum of square differences (SSD) of the two points is underneath a threshold, the feature pair is counted for the matrix-vector-pair. The threshold is explained in the chapter "Results" in section 6.1.4. On these feature pairs, the average distance from the calculated points to the matched points is measured. The matrix-vector-pair that generates the smallest average distance is likely suitable for further processing.

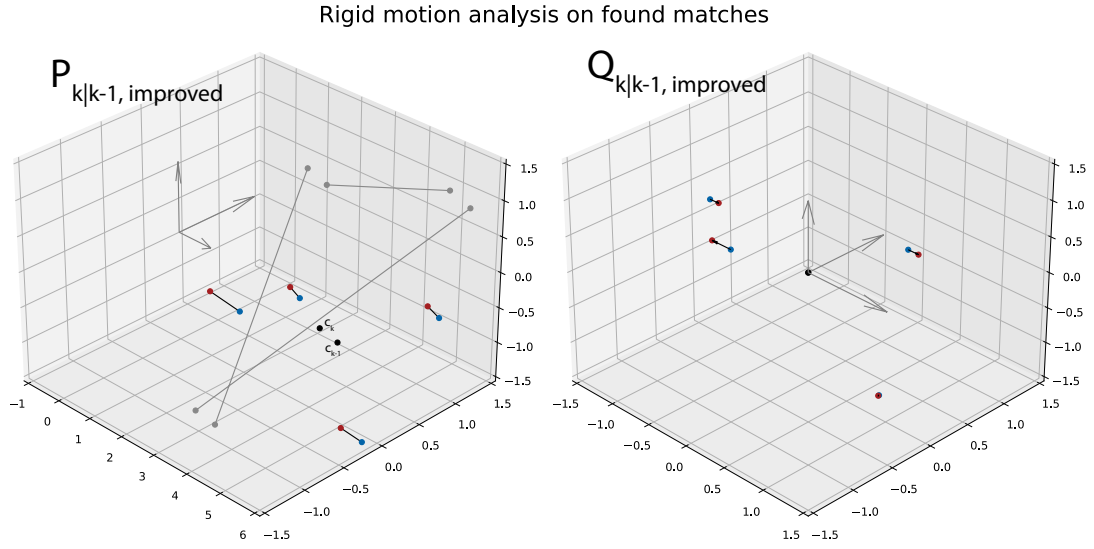

Figure 5.14: Second step of the 3D RANSAC: The calculated rotations and translations get evaluated against the whole dataset. In the example, the fourth matching point fulfills the criterion. As this group is the best possible in the example, an improved rotation $R_{improved}$ and translation $\vec{t}_{improved}$ is calculated from these.

In a third step, this matrix-vector-pair - R_i and $\vec{t_i}$ - is recalculated using all brute-force matches within a certain proximity to the calculated point, resulting in $R_{improved}$ and $\vec{t}_{improved}$.
Ignoring the SIFT metrics that lead to the brute force matching, all the features get matched anew based on the position alone, using the rotation and translation estimated before. The matrix-vector pair transforms every data point from the previous image. The distance between the transformed point and the closest data point of the current picture determines the new match. If the proximity is below a threshold, the match is accepted.

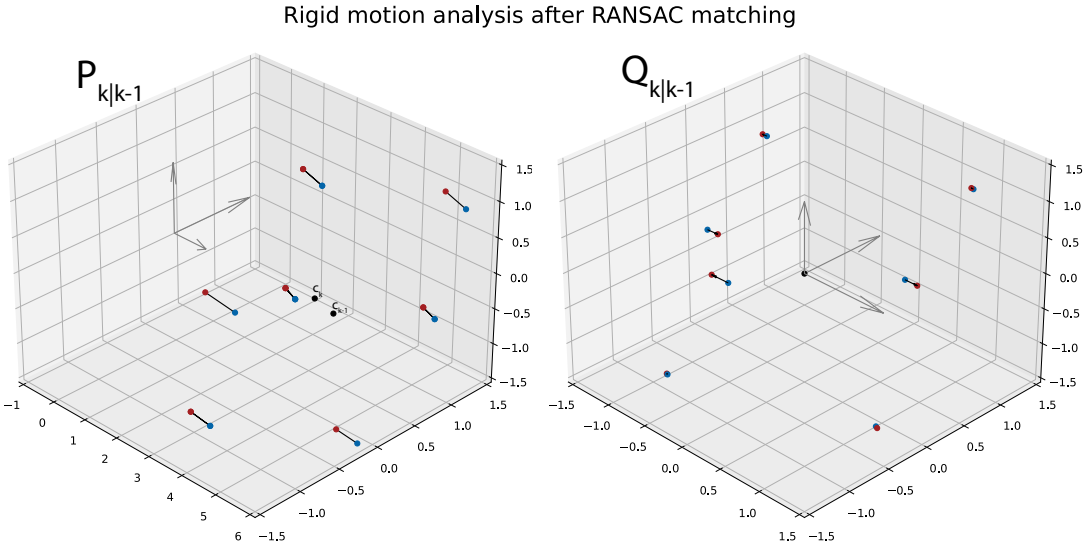

Figure 5.15: Third step of the 3D RANSAC: With $R_{improved}$ and $\vec{t}_{improved}$ being present, the brute-force matches get discarded. Applying the currently available rotation- and translation-information the the point clouds allows re-matching based alone on the position. Theoptimal rotation R_{opt} and translation ${}_{v}ect_{opt}$ result from these feature pairs.

Finally, the optimal rotation matrix and translation vector are calculated using the method described in section 5.3.3, this time not with only three points but all accepted matches. With more than three points, the calculated rotation and translation is not the exact result, but the result that yields the least square error.[11]
The whole methodology relies on finding the correct rotation and translation from randomly grouped matches.

5.4 Sensor Fusion

The following section describes the implementation of the accelerometer and gyroscope processing, marked blue in image 5.8, and the Kalman filter, marked in cyan in the same image, which fuses the data with the ToF camera motion data.

5.4.1 Gyroscope and Accelerometer

The 6-axis IMU Bosch BMI160 provides gyroscope and accelerometer measurements via an I²C connection, extended by a Silicon Labs CP2112 USB-to-I²C bridge. The BMI160 is soldered onto a module by DFRobot. The IMU generates measurements regarding acceleration in the directions a, b, and c, and measurements regarding rotation speed around these axes.
As the gyroscope and accelerometer output incremental movement, the values need to be integrated

over time. Accurate data is crucial for finding the direction of gravity or detecting movement - especially because of positional information being the result of derivating the acceleration twice.
The range of the acclerometer is variable and has been set to ±8 G, it has an output resolution of 16 bit and an output data rate of 200 Hz. The accelerometer has been calibrated in 24 orientations to even out angle errors inside the IMU, on the PCB and of the calibration table. For each orientation, 100 raw measurements have been averaged to even out noise. The largest error has been 38 mG, that lies within the sensor's specification of ±40 mG [26]. In addition, the gain for the accelerometer has been corrected based on gravity. A maximum error of 1.8% has been measured and corrected, which also lies in the sensor's specification of ±0.5% full scale [26].
For the gyroscope, the range is set to ±2000 degrees per second with an output data rate of 200 Hz. The gyroscope has been calibrated only for zero offset, whose maximum error was 0.2 degrees per second which lies well in the specified ±3 degrees per second [26]. A gain correction was not made, because of not having the required equipment to do so.
The IMU shows a hysteresis that lies within the specification of the datasheet; therefore, a simple offset correction does not yield the best results. A moving average – that gets updated whenever no motion of the camera head is detected – helps deal with the hysteresis by subtraction from the measurement value.

5.5 Sensor Fusion with Kalman Filter

The combination of data coming from different sensors or sensor types is named sensor fusion. In this case, the accelerometer and the calculated motion from the ToF camera add information to the system describing the same movement. Both sensors have noise and inaccuracies that need to be considered to calculate the system state $\vec{x}$, which includes the current position, velocity, acceleration, angular orientation, and angular velocity.
A method to estimate the system state is the Kalman filter that allows modeling the system by combining the sensors' uncertainties and the described model itself. The Kalman filter is a discrete predictor-corrector algorithm - as explained in section 4.5 - that uses a system model to predict the next system state using the previous system state $\vec{x}_{k-1}$. The sensor data corrects the system state prediction $\vec{x}_{k|k-1}$ which results in the corrected system state $\vec{x}$.
The system requires equal sampling rates for the individual sensors; therefore, the IMU data is downsampled to match the ToF camera's framerate. With the translational information being present in all three directions and the rotation being inserted in quaternion form, the system state vector $\vec{x}$ has 17 entries. x, y and z describe the position in space, and r_a, r_b, r_c, and r_d being the components of the orientation quaternion $R = (a, b\mathbf{i}, c\mathbf{j}, d\mathbf{k})$. The time-derivatives of these values are denoted with dots.

$$\vec{x}^T = (x, \dot{x}, \ddot{x}, y, \dot{y}, \ddot{y}, z, \dot{z}, \ddot{z}, r_a, r_b, r_c, r_d, \dot{r}_a, \dot{r}_b, \dot{r}_c, \dot{r}_d)$$

5.5.1 Prediction

The model F that predicts the intermediate state $\vec{x}_{k|k-1}$ brings the individual entries of the system state into a relationship. The velocities and accelerations both alter the position, while the accelerations alter the velocities. For the rotation, the quaternion containing the angular rotation already contains the sampling rate and gets added by a quaternion multiplication.

$$\vec{x}_{k|k-1} = F \cdot \vec{x}_{k-1}$$

The $\dot{r}$-values in the matrix are from $\vec{x}_{k-1}$ and Δt is the sampling rate.

$$\begin{pmatrix} x_{k|k-1} \\ \dot{x}_{k|k-1} \\ \ddot{x}_{k|k-1} \\ y_{k|k-1} \\ \dot{y}_{k|k-1} \\ \ddot{y}_{k|k-1} \\ z_{k|k-1} \\ \dot{z}_{k|k-1} \\ \ddot{z}_{k|k-1} \\ r_{a,k|k-1} \\ r_{b,k|k-1} \\ r_{c,k|k-1} \\ r_{d,k|k-1} \\ \dot{r}_{a,k|k-1} \\ \dot{r}_{b,k|k-1} \\ \dot{r}_{c,k|k-1} \\ \dot{r}_{d,k|k-1} \end{pmatrix} = \begin{bmatrix} 1 & \Delta t & \frac{\Delta t^2}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & \Delta t & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & \Delta t & \frac{\Delta t^2}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \Delta t & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & \Delta t & \frac{\Delta t^2}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \Delta t & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dot{r}_a & -\dot{r}_b & -\dot{r}_c & -\dot{r}_d & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dot{r}_b & \dot{r}_a & \dot{r}_d & -\dot{r}_c & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dot{r}_c & -\dot{r}_d & \dot{r}_a & \dot{r}_b & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dot{r}_d & \dot{r}_c & -\dot{r}_b & \dot{r}_a & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x_{k-1} \\ \dot{x}_{k-1} \\ \ddot{x}_{k-1} \\ y_{k-1} \\ \dot{y}_{k-1} \\ \ddot{y}_{k-1} \\ z_{k-1} \\ \dot{z}_{k-1} \\ \ddot{z}_{k-1} \\ r_{a,k-1} \\ r_{b,k-1} \\ r_{c,k-1} \\ r_{d,k-1} \\ \dot{r}_{a,k-1} \\ \dot{r}_{b,k-1} \\ \dot{r}_{c,k-1} \\ \dot{r}_{d,k-1} \end{pmatrix}$$

The $\dot{r}$ values inside the model matrix F are the same as in the vector $\vec{x}_{k-1}$. Technically, the operation is not a linear transformation anymore, as the quaternion multiplication itself is not linear.
If there were a known active influence on the system, for example, an RC car to which the camera head could be mounted, the motor input and the steering could be modeled in a different matrix B with the input vector $\vec{u}$.

$$\vec{x}_{k|k-1} = F \cdot \vec{x}_{k-1} + B \cdot \vec{u}$$

As there are no known active inputs to the system, both B, and $\vec{u}$ are omitted.
The Kalman filter relies on knowing the uncertainties of the model. If a system has rigid constraints by design, the sensor data will naturally be interpreted to support the model. The model of the camera head has loose restrictions, as the camera head is free to move in any direction and rotation; therefore, the output heavily relies on the sensor data.
The second step of the prediction is the prediction of the covariance matrix of the errors P.

$$P_{k|k-1} = F \cdot P_{k-1} \cdot F^T + Q$$

While P_0 starts as identity matrix, Q is the gaussian process noise, which got explained in section 4.5 for the translational part. For the rotation, the diagonal elements got set to 50, making the system noise vastly bigger than the noise of the input data.
Generally, a large process noise leads to a lower emphasis to the current system state and larger weight to the sensor data.

$$Q = \begin{bmatrix}
\frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} & \frac{\Delta t^2}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{\Delta t^3}{2} & \Delta t^2 & \Delta t & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{\Delta t^2}{2} & \Delta t & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} & \frac{\Delta t^2}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{\Delta t^3}{2} & \Delta t^2 & \Delta t & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{\Delta t^2}{2} & \Delta t & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} & \frac{\Delta t^2}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \frac{\Delta t^3}{2} & \Delta t^2 & \Delta t & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \frac{\Delta t^2}{2} & \Delta t & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 50 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 50 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 50 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 50
\end{bmatrix}$$

5.5.2 Correction: Rotation

The Kalman filter allows chaining multiple correction steps after another, which helps add data of different sensors to the same input and helps simplify the calculation. The observation matrix H maps the four quaternion values of an input – gyroscope or ToF camera – to the 17 values of the system state vector x. Choosing a 4×17 matrix changes the bracket of the following equation to a 4×4 matrix, simplifying the matrix inversion for calculating the Kalman gain. The iteration n in the equations denote the number of the chained correction steps, as the system state vector $\vec{x}$ and the covariance matrix of errors P are updated in each step.

$$K_{k|k-1|n} = P_{k|k-1|n} \cdot H^T (H \cdot P_{k|k-1|n} \cdot H^T + R)^{-1}$$

The order of the subsequent correction steps is irrelevant. The following H maps the rotation speed to the system state and is used for both sensors.

$$H = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}$$

For both sensors – the gyroscope and the ToF camera – R is a diagonal matrix with the standard deviation of the respective sensor as its values.

$$R = \sigma \cdot I$$

With the Kalman gain at hand, the correction for the system state vector $\vec{x}$ and the covariance matrix of errors P are the last equations of the Kalman correction step with the rotation quaternion Q_{Rot}.

$$x_k = \vec{x}_{k|k-1|n} + K_k(\vec{z}_k - H \cdot \vec{x}_{k|k-1|n}) \quad ; \quad P_{k|k-1|n+1} = (I - K_k \cdot H) P_{k|k-1|n}$$

$$Q_{Rot} = \begin{pmatrix} r_c \\ a_c\mathbf{i} \\ b_c\mathbf{j} \\ c_c\mathbf{k} \end{pmatrix} \quad ; \quad \vec{z}_k = \begin{pmatrix} r_c \\ a_c \\ b_c \\ c_c \end{pmatrix}$$

The quaternion multiplication inside the model matrix F adds up the subsequent rotation speed in each iteration to the cumulated rotation.

5.5.3 Rotation Drift compensation with Accelerometer

When the camera head is stationary, the accelerometer measures the direction of gravity, allowing the correction of rotation on the x- and y-axis. Using the earth's magnetic field, a magnetometer could compensate for the drift on the z-axis, but the used IMU does not contain one. Knowing the system's orientation at the start Q_{init} of the application and the cumulated rotation R, calculating the system's current orientation Q_c is a quaternion multiplication, as described in section 4.2.2. If the orientation drifted, it would be different from the measured orientation Q_m from the accelerometer. The complex parts of the two orientation quaternions get normalized and stored in individual vectors $\vec{s}$ and $\vec{d}$.

$$Q_c = \begin{pmatrix} r_c \\ a_c\mathbf{i} \\ b_c\mathbf{j} \\ c_c\mathbf{k} \end{pmatrix} \quad ; \quad \vec{v}_c = \begin{pmatrix} a_c \\ b_c \\ c_c \end{pmatrix} \quad ; \quad \vec{s} = \frac{\vec{v}_c}{|\vec{v}_c|} \quad ; \quad Q_m = \begin{pmatrix} r_m \\ a_m\mathbf{i} \\ b_m\mathbf{j} \\ c_m\mathbf{k} \end{pmatrix} \quad ; \quad \vec{v}_m = \begin{pmatrix} a_m \\ b_m \\ c_m \end{pmatrix} \quad ; \quad \vec{d} = \frac{\vec{v}_m}{|\vec{v}_m|}$$

The dot product of the two normalized vectors calculates the cosine of the angle between the vectors θ, and the cross product calculates the rotation axis $\vec{r}$.

$$cos(\theta) = \vec{s} \cdot \vec{d} \quad ; \quad \vec{r} = \vec{s} \times \vec{d} \quad ; \quad \vec{r} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \end{pmatrix}$$

Estimating the value x from the angle helps scale the values for generating a correcting rotation quaternion Q_{corr}.

$$x = \sqrt{(1 + cos(\theta) * 2)} \quad ; \quad Q_{corr} = \begin{pmatrix} 0.5 \cdot x \\ r_1 \cdot \frac{1}{x}\mathbf{i} \\ r_2 \cdot \frac{1}{x}\mathbf{j} \\ r_3 \cdot \frac{1}{x}\mathbf{k} \end{pmatrix}$$

An additional Kalman correction step adds the correcting rotation quaternion Q_{corr} to the system. The calculation is the same as for the rotation quaternions from the gyroscope and the ToF camera, which is described in section 5.5.2.

5.5.4 Correction: Translation

Data from the accelerometer and the translational velocity from the ToF camera get sampled in the ABC-space, as the sensors rotate with the whole camera head. The speeds in the ABC-space require transformation to the XYZ-space, as covered in section 2.2.2.. After processing all the rotation sensors, the system rotation in the system state vector x transforms the data into the XYZ-space. Even though six individual values from two different sensors correct the system state vector, the 4-dimensional

equations of the rotation correction are kept. By choosing H with a lower rank, three-dimensional corrections are possible with the same formulae. Same as for the rotation, the system state vector and the covariance matrix of errors get refreshed at each subsequent correction. Again, the following three equations are calculated:

$$K_k = P_{k|k-1|n} \cdot H^T (H \cdot P_{k|k-1|n} \cdot H^T + R)^{-1}$$

$$x_{k|k-1|n+1} = \vec{x}_{k|k-1|n} + K_k(\vec{z}_k - H \cdot \vec{x}_{k|k-1|n})$$

$$P_{k|k-1|n+1} = (I - K_k \cdot H) P_{k|k-1|n}$$

The observation matrices for acceleration data H_a and for velocity data H_v are different.

$$H_a = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$H_v = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The process noise matrices R are diagonal matrices with the standard deviation being the values.

$$R = \sigma \cdot I$$

The input vectors $\vec{z}$ contain the XYZ-corrected sensor values, either for acceleration or for velocity.

5.6 Raspberry Pi Camera calibration

The Raspberry Pi camera only provides images for enhancing the system's video output cosmetically; the lens calibration serves no algorithmic purpose. The lens was calibrated solely to map the image into the undistorted Vulkan 3D space, so the virtual rectangle fits the real world's picture.
The camera streams in 720p to use the entire sensor with pixel binning. In 1080p, the sensor uses only a portion of the image sensor, which narrows down the field of view. Like with the ToF camera, a lookup table calibrates the Raspberry Pi camera, which reduces the image size to a resolution of 1273 times 709 pixels. Section 4.4 describes the process of the lens calibration.

5.7 Video display

TODO: Vulkan Output, briefly explained

6 Testing and Results

The following chapter describes the calibration, testing and the results of the system components.

6.1 ToF Camera

The time-of-flight camera is the crucial part of this thesis, allowing a three-dimensional scene reconstruction. This section describes the measurements and the results of the motion estimation of the ToF Camera at every involved step.

6.1.1 Setup

As described in section 5.1.2, the ToF camera sends its data by ethernet, using a lightweight TCP protocol. The software controlling the setup of the ToF camera runs on the Raspberry Pi and is the proprietary part of the ToF software stack. The software allows a basic configuration in two main modes: automatic and manual control. The software supports HDR functionality in both automatic and manual control, which significantly enhances the dynamic range on the infrared black-and-white image. The implementation of the HDR function is not documented by the supplier of the Nimbus 3D ToF camera.
The ToF camera uses an infrared flash, which is not brightness-controlled; setting the camera to a fixed exposure time would lead to overexposure on close objects. The automatic mode lets the user select a maximum amplitude of the image, to which the exposure time is set. As the CudaSift library, which extracts the SIFT features from the ToF camera, only supports the 8-bit resolution, the maximum amplitude got selected at 255. Higher values lead to a longer exposure time and let the frame rate drop. Not needing any amplitude scaling while keeping a high frame rate is favorable.

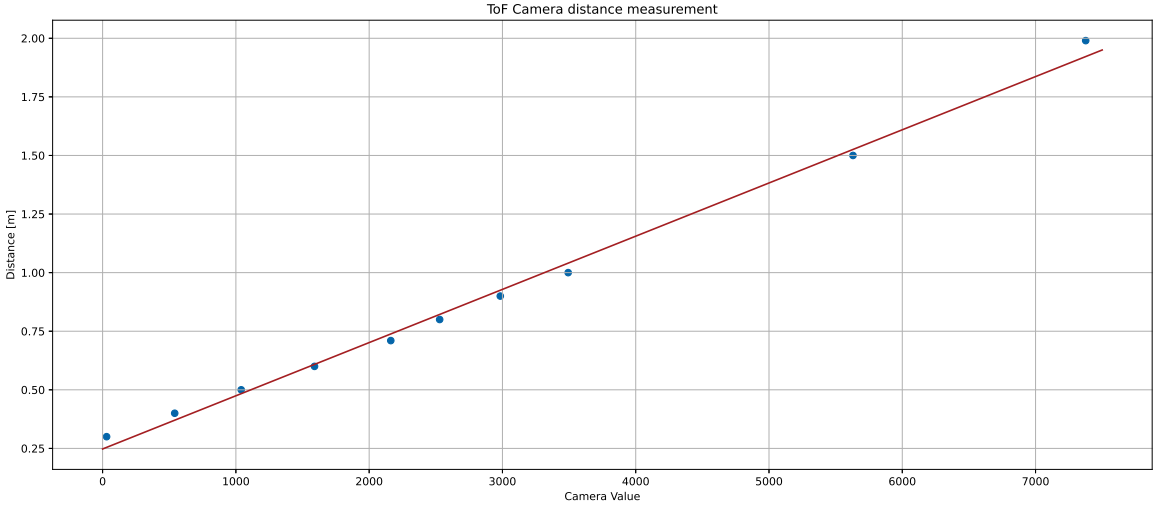

Figure 6.1: Sensor values measured against a meterstick. In red, the linear fit generated in Microsoft Excel.

The coordinate system is not corrected for x, y and z yet, as the ToF camera itself does not know its rotation in the field of gravity. Therefore, the axis for the ToF camera calculations are kept in the a, b and c system. Details are explained in section 4.2.4.

6.1.2 Distance measurement

Every pixel of a ToF camera sensor combined with the wide-area infrared flash acts like a laser rangefinder. A single pixel of the sensor targeting a brown cardboard surface positioned at various distances allows measuring this distance with a meterstick and a comparison with the sensor pixel value. A linear fit in Microsoft Excel leads to the following function, which is also visualized in image 6.1:

$$d[m] = 0.000227 * val + 0.247532$$

The camera noise does affect not only the black-and-white image but also the distance measurement. At the chosen camera settings, the camera noise lets the distance measurement have a standard deviation of around 7mm in the a-axis.
The formulae for the 3D scene reconstruction propagate the standard deviation to about 4mm in the b-axis and 3mm in the c-axis.

6.1.3 3D Scene Reconstruction

After calibrating the lens and correcting the radial nature of the distance measurement as described in section 5.3.1, the generated point cloud should directly reconstruct the three-dimensional scene recorded by the ToF camera. Section 5.3.2 describes the process of this reconstruction. Straight lines in the real world should appear straight in the point cloud, which can be checked by analyzing an image. The two images in figure 6.4 demonstrate the point cloud generation, by mapping SIFT features into the 3D space.

Figure 6.2: Image

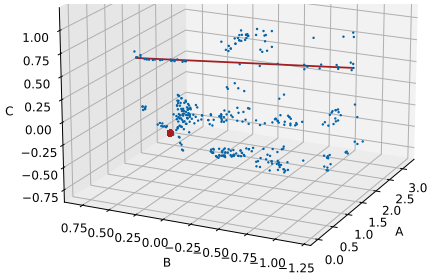

Figure 6.3: Cloud

Figure 6.4: The scene reconstruction from the left image to the full point cloud. The red line is equivalent in both figures. Each green dot in the image on the left is a SIFT feature. The red dot in the point cloud is the position of the camera.

6.1.4 RANSAC feature matching

The RANSAC algorithm improves the quality of matched features over the flawed brute-force matcher. Section 5.3.4 explains the implementation of the three-dimensional RANSAC and mentions a threshold. The noisy data the ToF camera delivers gives single SIFT features a positional uncertainty. As discussed in section 6.1.2, the standard deviation in the a-axis is roughly 7mm, which also affects the mapping to the$b$- and c-axis from the 3d reconstruction. Therefore, the RANSAC matcher is required to have flexibility against positional noise.
Prior analysis of the dataset tells a possible rotation and translation. This rigid motion moves each data point in the old cloud to an estimated position, where a data point of the new cloud is expected. The lowest sum of square differences (SSD) of all the possible matches to the estimated position determines a match. It is accepted if this matched point lies inside the chosen SSD threshold. The SSD creates a sphere around the estimated position. The equation of a sphere of radius r follows the

equation $r^2 = x^2 + y^2 + z^2$. Setting the radius r equal to the standard deviation of 7 mm on the a-axis results in a threshold of around 0.00005. Statistically, around 68% of the matches should reside inside this sphere of 14 mm diameter. The error on the other axis is smaller as discussed in section 6.1.2, this does not explain the poor matching performance at this threshold, as seen in image 6.5. Because of the image noise on the black-and-white image, the extracted SIFT features jitter. Even a jitter in the range of 1 pixel easily leads - depending on its distance - to an error of more than a centimeter. On the other hand, to detect a lateral speed of 0.1 m/s at a frame rate of approximately 10 frames per second, a shift of 1 cm needs to be detected.

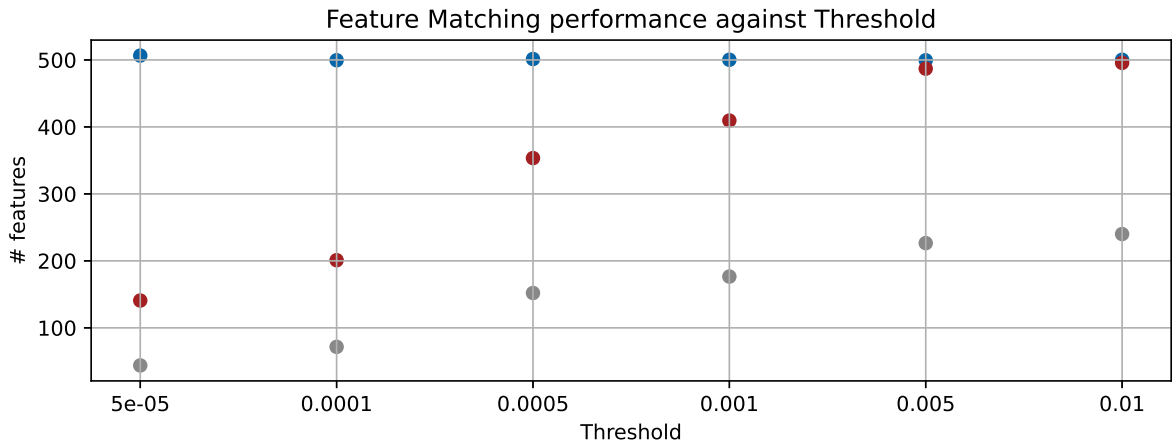

Figure 6.5: Plotting the feature matching performance against different thresholds shows the quality of the RANSAC feature matcher. In blue, the raw number of unmatched features in each test, in gray the brute force matches and in red the RANSAC matches.

Higher tresholds in image 6.5 show the expected performance of the brute-force matcher in grey, which lies below 50% according to the developers of the CudaSift library.[20] The RANSAC feature detector vastly improves the matching performance, seemingly up to over 97% at the threshold of 0.005. At the threshold of 0.005, the sphere around the expected position measures over 14 cm in diameter. The RANSAC matcher might find the same match for different feature points; its matches are not exclusive, combined with the large threshold, it is likely that most features find a match.
The balance between the uncertainty and the requirements for detecting slow speeds lead to a chosen threshold of 0.0005. This threshold leads to a sphere of about 4.4 cm in diameter, which filters harsh outliers.
Figure 6.6 on the next page demonstrates the matching with the chosen threshold at the example of a rotation of the camera head. Two consecutive frames generate a point cloud each; one is marked blue, the other in red. The estimated rotation and translation move each blue data point to a hypothetical position. The closest red data point around this hypothetical position is accepted if it lies within a sphere of 4.4cm diameter. After another calculation, these matched data points lead to the optimal rigid motion, discussed in the following sections.

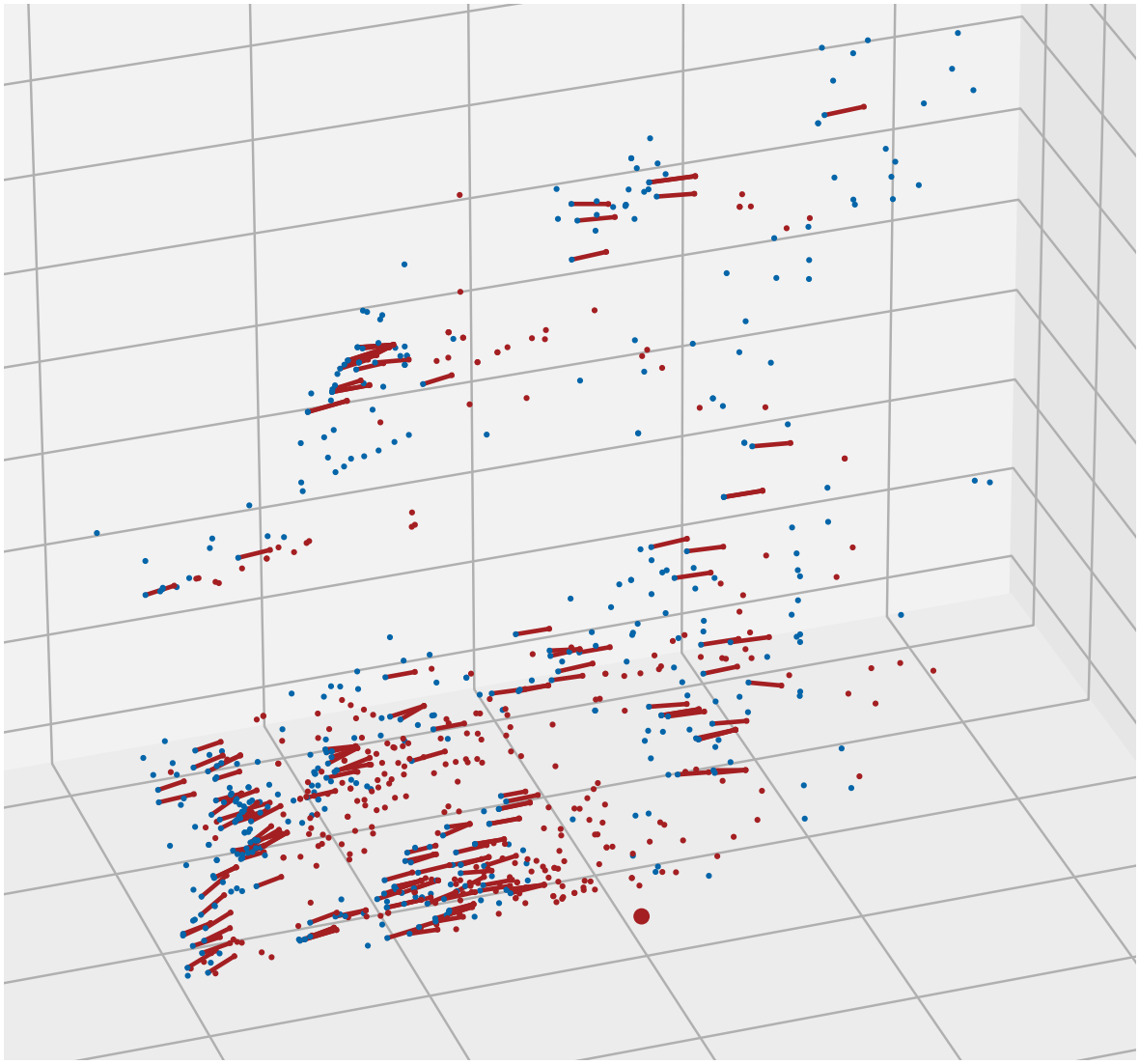

Figure 6.6: Two raw point clouds with the found matches connecting them. The camera is rotated in between the frames, which leads to the displacement of the point cloud. The connecting lines are roughly 10cm long, depending on the position. The large red dot in the foreground is the position of the camera. The scene is the same as in image 6.4, but zoomed for better visibility.

6.1.5 Rotation from ToF camera

The optimal rotation between the point clouds is ready for comparison against the gyroscope of the IMU. Both the gyroscope and the ToF rotation generate rotation speeds and are both converted into rotation quaternions. Transforming the rotation speed to a quaternion allows direct comparison of the different imaginary parts. In addition, the quaternions can be fed directly to the Kalman Filter. The real part of a rotation quaternion is always entirely dependent on the imaginary factors; therefore, analyzing the imaginary parts is sufficient, as explained in section 4.2.2. For demonstration, the camera head gets rotated in each direction and directly compared to the gyroscope data. The author does not own a device that allows reference rotations; the camera got mounted on a standard camera tripod and rotated by hand. Figure 6.7 shows the results. First the camera head got rotated sideways around the c-axis, then up and down along the b-axis and lastly turned along the a-axis. The plotted values resemble imaginary parts of a quaternion and are therefore without units.

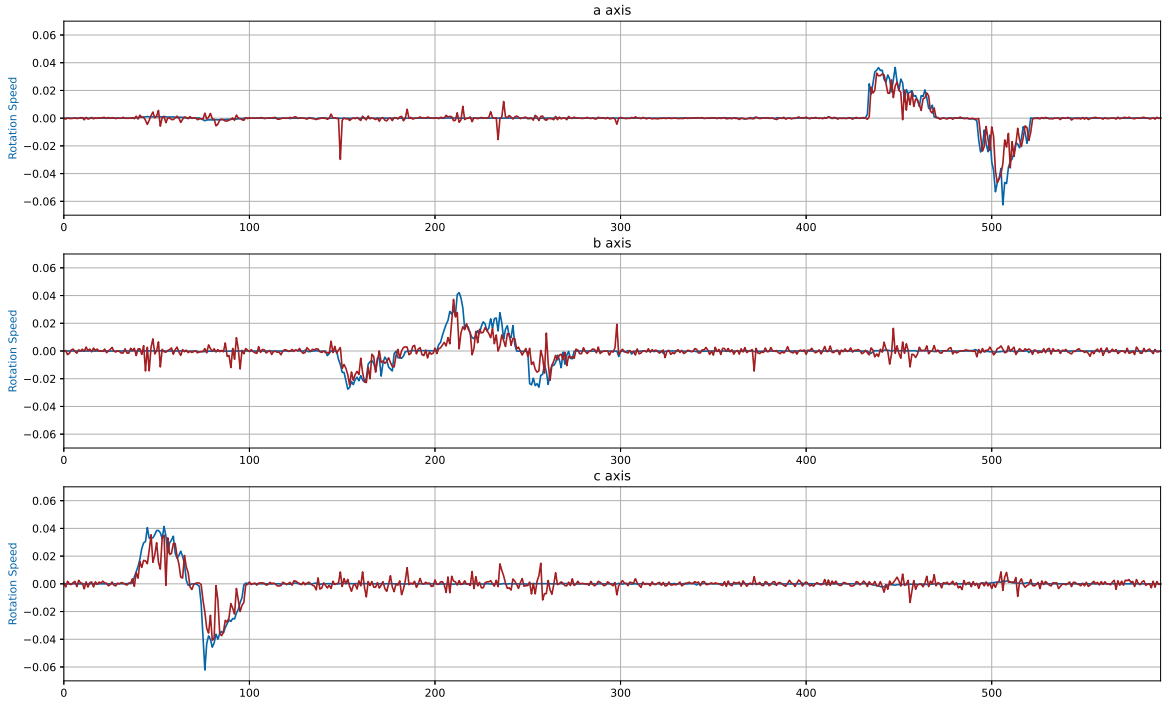

Figure 6.7: The individual axis of the ToF rotation quaternion plotted against the gyroscope quaternion. The camera head got rotated in each direction after another.

The measurement shows significant noise, especially on the rotation alongside the b- and c-axis, and does not entirely follow the gyroscope curve in blue. Nevertheless, the concept works as this measurement demonstrates.

6.1.6 Translation from ToF camera

Like for the rotation measurement, the translation is measured against the IMU. In the case of translation, the IMU only contains an accelerometer, which stands in contrast to the velocity estimated by the ToF camera. As for the rotation, the author has no device that allows standardized motion; the camera head is moved by hand.
A toy railway system helps guide the camera alongside the x- and y-axis. Vertical motion - along the z-axis - technically works the same as motion alongside the y-axis; it is not measured, as there is no vertical rail system at hand. The translation cannot be measured in one take as the Railway needs to be moved for the different axis.
Alongside the x-axis, the camera moves closer to the objects in the viewfinder, alongside the y-axis, the camera moves perpendicular. Note that these measurements are already in xyz-notation, as the orientation is corrected. As the camera does not rotate during these measurements, the values are equivalent to the abc-notation. The camera is slid from its origin to the front, respectively to the side, twice, while going back to the origin twice. The first motion is kept fast, the second motion is kept slow. The length of the track is about 0.5 m in both directions.
As visible in the figures 6.8 and 6.9 on the next page, the velocity data from the ToF camera shows significant noise, but the motion is detected correctly. The coloring of the curves is kept the same as in figure 4.10.

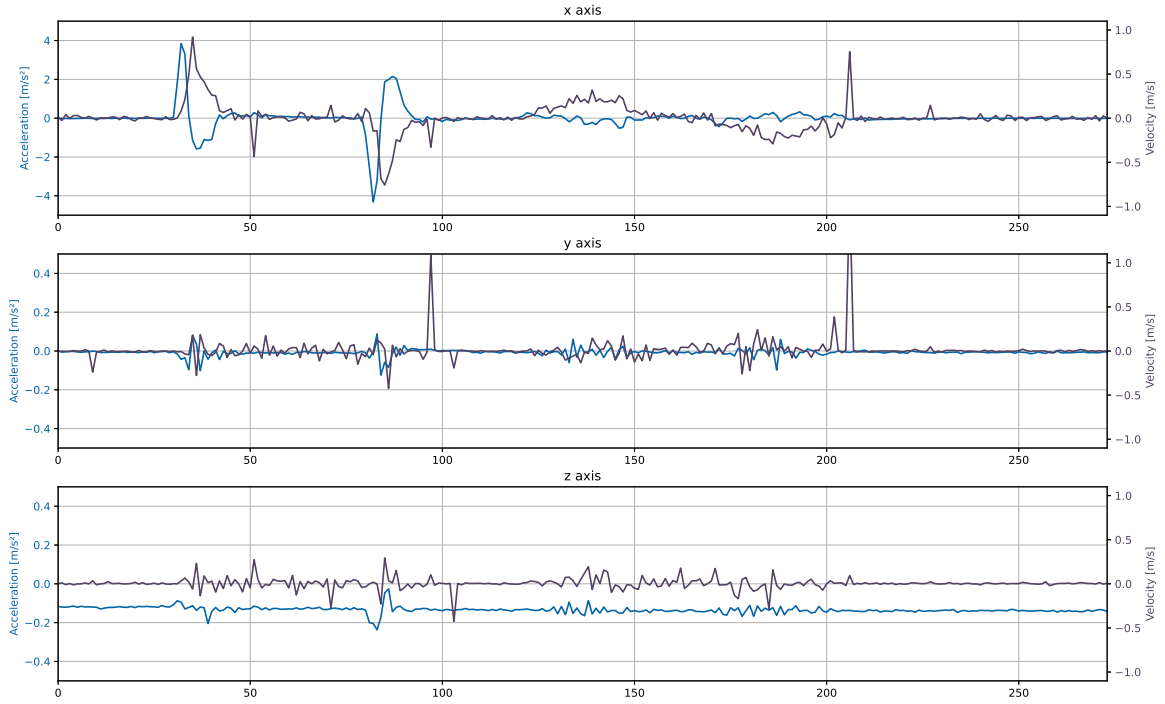

Figure 6.8: Raw measurement of the translation alongside the x-axis. Before and around 50, the camera is slid fast forward, kept in pause just to slide back before reaching frame number 100. After 100 and before 220, the same motion is repeated but slower. Blue is the acceleration and purple the velocity.

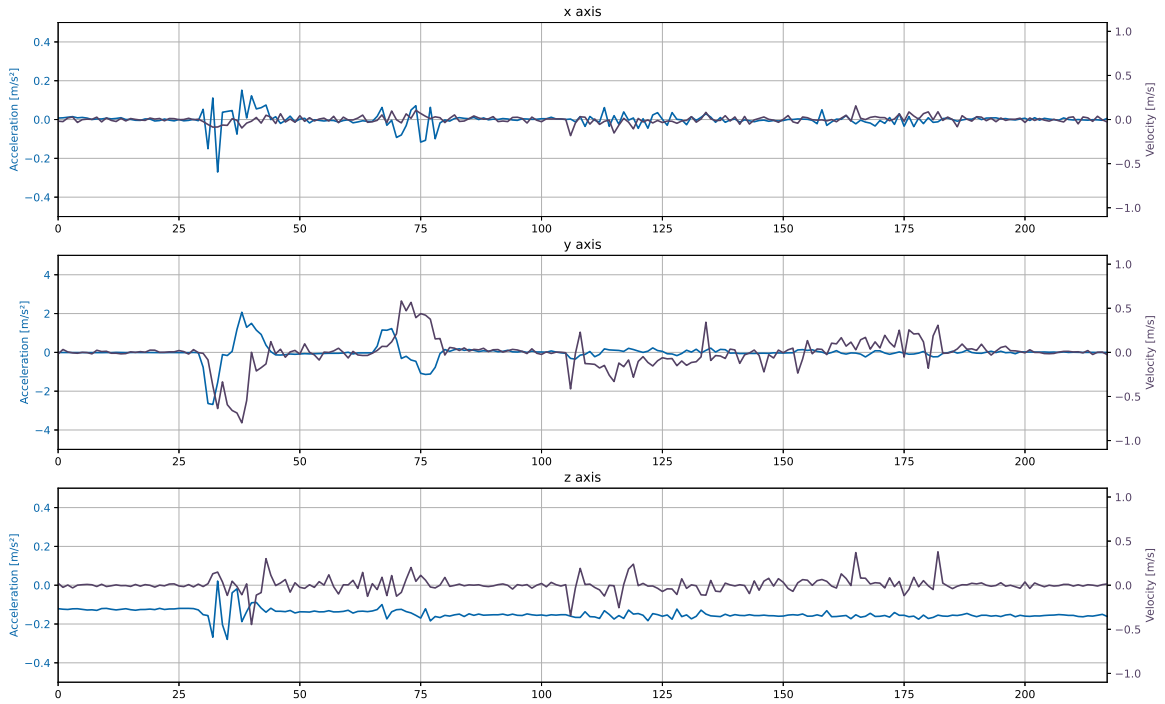

Figure 6.9: Raw measurement of the translation alongside the y-axis. Between the frames 25 and 50, the camera is slid to the right, kept there and slid back to the origin around frame 75. The motion is repeated slower after frame 100 and before 200. Blue is the acceleration and purple the velocity.

Raw integration of the ToF velocity values gives insight into the measurement's quality. As visible in image 6.10, the data reconstructs the linear motion in both directions, even though it gets jagged by noise as expected from theory. Noteable outliers, like at the end of the second motion on the x-axis, induce larger errors.

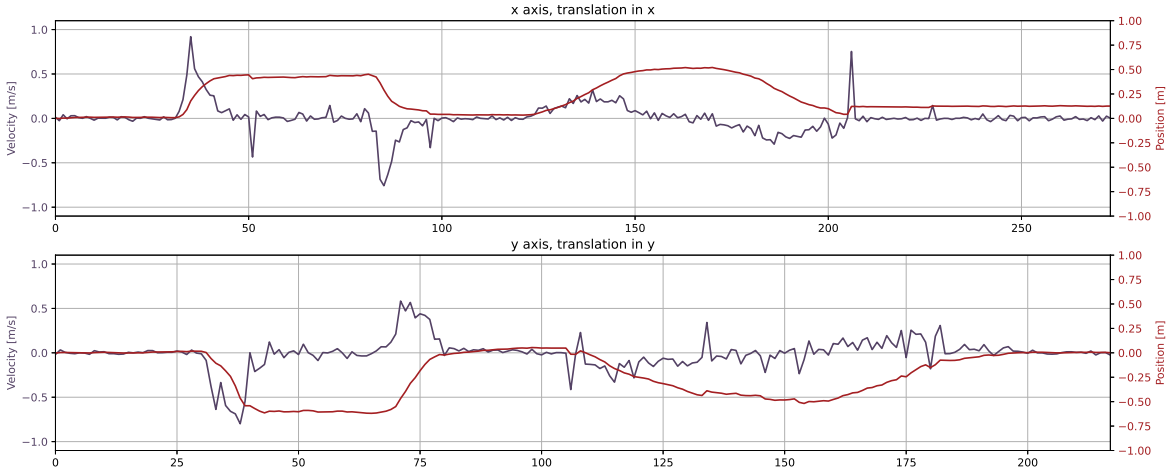

Figure 6.10: Raw integration of the relevant axis in both translations of the ToF velocity data. Red: ToF velocity, Blue: Its integration.

6.2 Inertial Measurement Unit

The Inertial Motion Unit (IMU) contains a gyroscope and an accelerometer, whose data inputs the Kalman filter. While the comparison to the rotation estimation from the ToF camera already covered the gyroscope in section 6.1.5, the accelerometer data requires more profound analysis.

6.2.1 Accelerometer

An accelerometer within the gravitational field of the earth will always measure the earth's gravitational pull in addition to other accelerations. The accelerometer needs calibration on each axis to compensate for the gravitational force, so the subtraction of $\vec{g}$ works in any orientation. Experimentation with the accelerometer has shown that a hysteresis prevents the accelerometer from reaching zero or $\vec{g}$ when standing still, depending on prior rotation. The hysteresis leads to a tiny offset on the acceleration, which bleeds into an integrated velocity and devastatingly affects the further integrated position, as visible in image 6.11 on the next page. The offset is easily visible in images 6.8 and 6.9 on the z-axis, thanks to the narrowed scale.
The added offset error worsens the drift compared to the simulation in section 4.5 dramatically. The integration to the velocity draws a smoother curve, than compared to the raw data of the ToF camera, but a drift is present. The third plot - of the z-axis - shows the devastating drift, if the gravitational pull is not compensated entirely. The data for the third plot got recorded during the measurement of the motion alongside the x-axis.

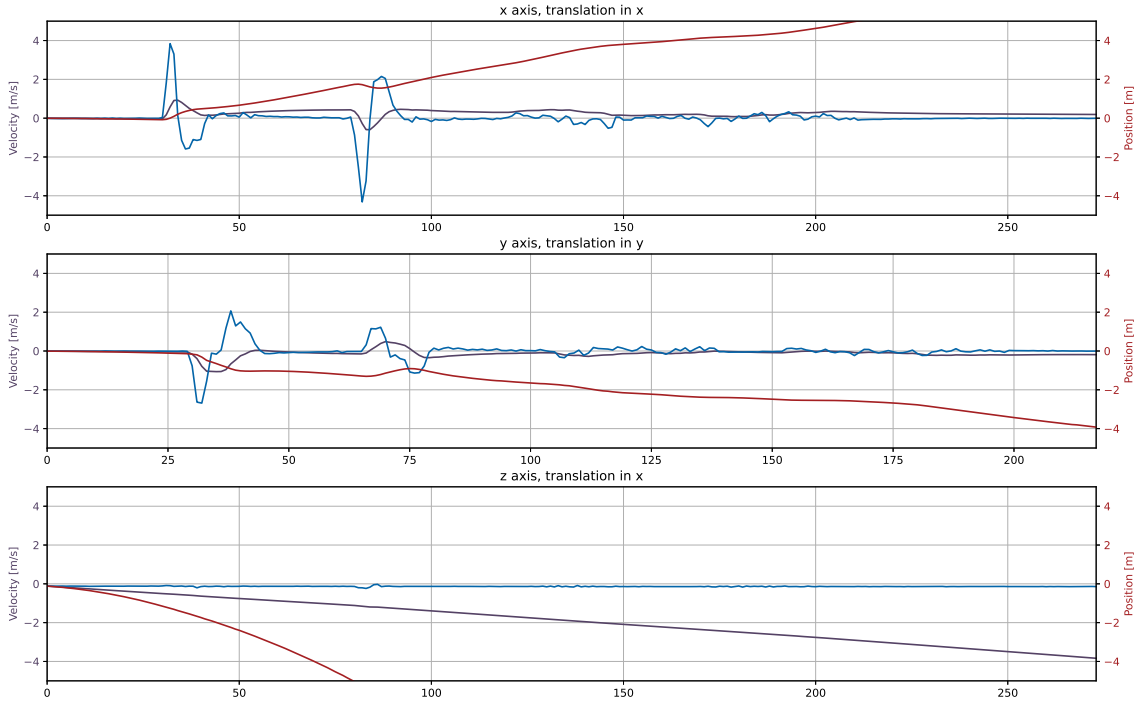

Figure 6.11: Raw integration of the relevant axis in both translations of the accelerometer. Blue is the acceleration, purple the velocity (single integration) and red the position (double integration).

6.3 Kalman Filter

The Kalman filter combines the known motion model of the camera head with the measurement uncertainties and the raw input data from the ToF camera and the IMU. Both types of motion – rotation and translation – are calculated in one large matrix but mostly decoupled. The only coupling between the rotation and the translation is the transformation from ABC-coordinates to the XYZ-space. The rotation is required for the translation data; it gets analyzed first.

6.3.1 Rotation

As mentioned in section 6.1.5, the gyroscope and the ToF camera generate a rotation speed of comparable magnitude and shape. The gyroscope data is less distorted by noise, which gives this sensor more weight in the Kalman filter. The Kalman filter smoothes data for the rotation speed through its model function and calculates the rotation through an integration step. Additional to the gyroscope and the ToF camera, the drift compensation with the accelerometer stabilizes the values when the camera head is stationary.
As explained in section 5.5, the integration step is a quaternion multiplication of the prior value with the new rotation speed.
When plotting the output of the Kalman filter against the raw data of the gyroscope and the ToF camera, the strong weight towards the gyroscope becomes apparent, as shown in figure 6.12 on the next page.
Without a reference measurement, it isn't easy to classify these results. The demonstration in figure 6.15 of the whole pipeline using only rotation data helps estimate the result's quality. The three rotation axes of the camera tripod do not intersect with the recording Raspberry Pi camera; therefore, the 3D object is expected to move within the camera image. While the gyroscope's rotation axes intersect in the housing of the IMU, the axes of the ToF camera intersect at the optical center of the lens.

For better accuracy, these sensor- and camera displacements on the camera head require additional compensation.

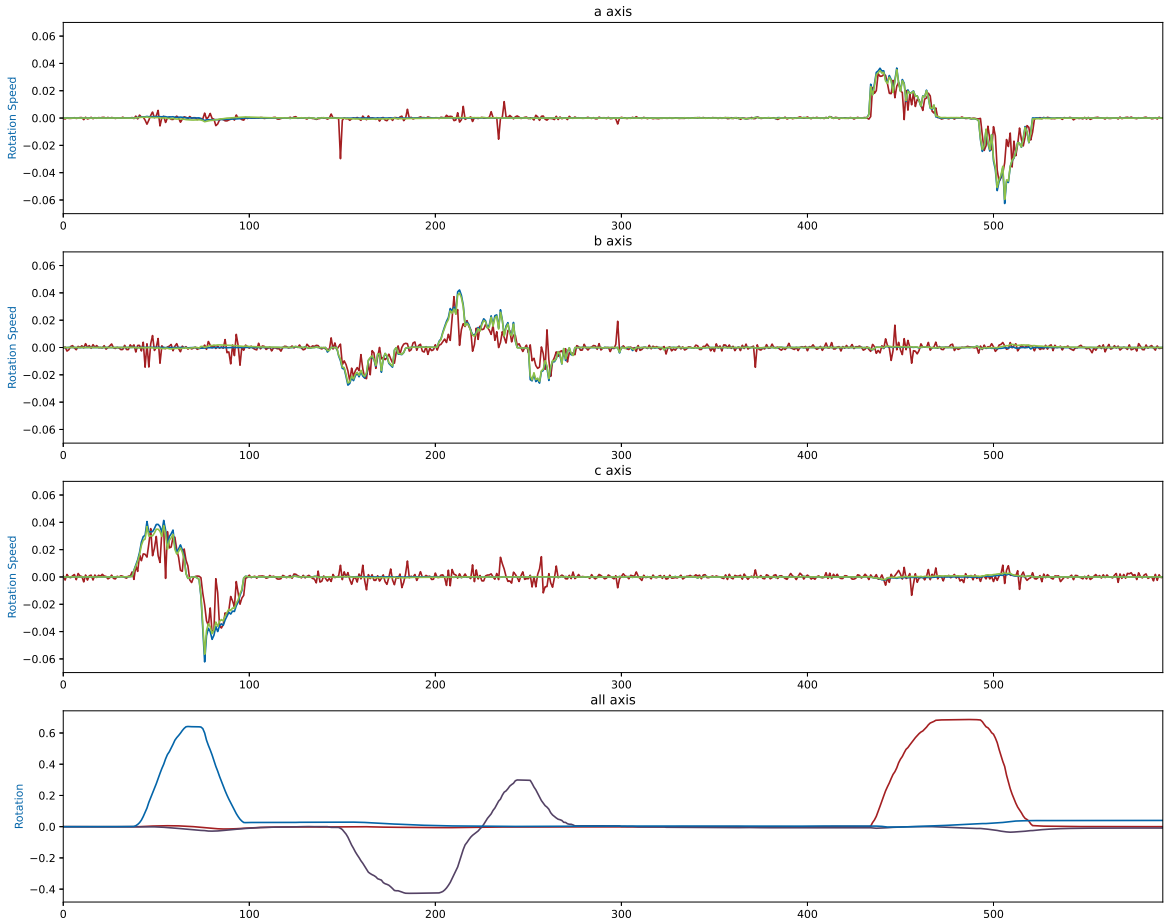

Figure 6.12: The output of the Kalman filter is plotted in green, against the gyroscope data in blue and the ToF camera data in red in the upper three graphs. The fourth graph shows the calculated rotation. The a-axis in red, the b-axis in purple and the c-axis in blue.

Figure 6.13: Initial situation

Figure 6.14: After rotation

Figure 6.15: Demonstration of the rotation against the camera image with the moving projection. The displacement of the rectangle inside the picture frame is a result of the Raspberry Pi camera facing translational motion when rotated on the camera tripod.

6.3.2 Translation

The fusion of the accelerometer data and the velocity data from the ToF camera has been simulated in section 222. The simulation did not consider that an offset taints the accelerometer data, leading to an even faster drift on its integrations. Figure 333 shows the output of the Kalman filter in green against the data from the accelerometer in blue and the ToF camera in red. Lacking any other input to compare against, the Kalman filter follows the accelerometer directly; the blue line is hidden behind the green Kalman output. The accelerometer's integration curve compares directly against the raw ToF camera data and the Kalman filter output on the velocity. Further integration allows the comparison with the positional data.
Arguably, the accelerometer taints the Kalman filter data, rendering it worse than the raw integration of the ToF camera. The Kalman velocity output is significantly worse than in the simulation, the proposed integration on this output, to get even better positional data fails.

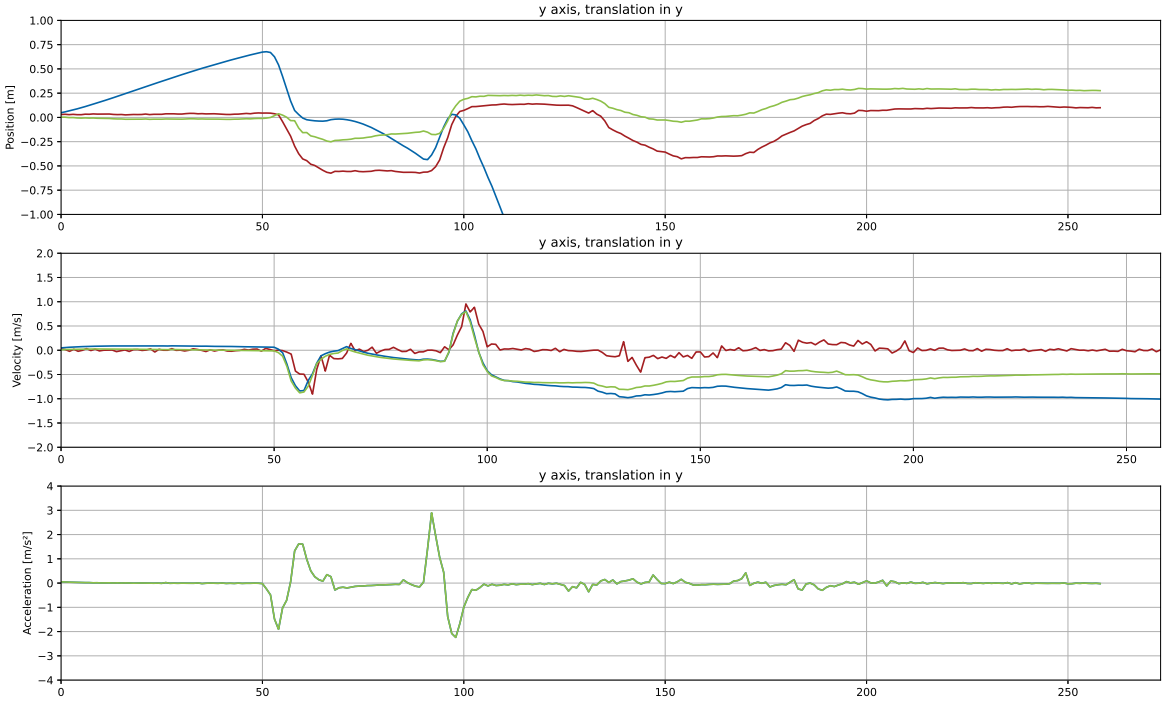

Figure 6.16: Motion in y direction. Blue lines are data from the accelerometer and its integrations, red are the velocity from the ToF camera and its integration and in green are the different Kalman outputs.

6.4 Cross sensitivity

Todo: Discuss translation of rotation tests and a free hand motion.

6.5 Performance

Todo: Cuda-Speed Performance measurement

7 Conclusion

7.1 Possible improvement

7.1.1 Proposal: Position Estimation by Camera

7.2 Outlook

Bibliography

[1] Kpmg - the future of virtual and augmented reality: Digital disruption or disaster in the making? https://home.kpmg/xx/en/home/insights/2016/03/the-future-of-virtual-and-augmented-reality.html, March 2016. Accessed: 28.01.2021.

[2] Deloitte - augmented/virtual reality, nixt bigh thing of digital environment. https://www2.deloitte.com/content/dam/Deloitte/in/Documents/technology-media-telecommunications/in-tmt-augmented-reality-single%20page-noexp.pdf. Accessed: 28.01.2021.

[3] Bloomberg - this time, augmented reality really could be the next big thing. https://www.bloomberg.com/news/newsletters/2021-06-09/this-time-augmented-reality-really-could-be-the-next-big-thing, June 2021. Accessed: 28.01.2021.

[4] Time - nintendo stock slumps as investors realize it doesn't make pokémon go. https://time.com/4421450/pokemon-go-nintendo-shares-tokyo/, July 2016. Accessed: 28.01.2021.

[5] Google glass. https://www.google.com/glass/start/. Accessed: 28.01.2021.

[6] The guardian - google glass advice: how to avoid being a glasshole. https://www.theguardian.com/technology/2014/feb/19/google-glass-advice-smartglasses-glasshole, February 2014. Accessed: 28.01.2021.

[7] Microsoft hololens. https://www.microsoft.com/de-ch/hololens. Accessed: 28.01.2021.

[8] Microsoft - airbus reaches new heights with the help of microsoft mixed reality technology. https://news.microsoft.com/europe/features/airbus-reaches-new-heights-with-the-help-of-microsoft-mixed-reality-technology/, June 2019. Accessed: 28.01.2021.

[9] Apple unveils new ipad pro with breakthrough lidar scanner and brings trackpad support to ipados. https://www.apple.com/newsroom/2020/03/apple-unveils-new-ipad-pro-with-lidar-scanner-and-trackpad-support-in-ipados/. Accessed: 28.01.2021.

[10] What is a lidar scanner, the iphone 12 pro's camera upgrade, anyway? https://www.techradar.com/news/what-is-a-lidar-scanner-the-iphone-12-pros-rumored-camera-upgrade-anyway. Accessed: 23.07.2021.

[11] Olga Sorkine-Hornung and Michael Rabinovich. Ethz note: Least-squares rigid motion using svd, January 2017.

[12] Mit: Singular value decomposition (svd) tutorial. http://web.mit.edu/be.400/www/SVD/Singular_Value_Decomposition.htm. Accessed: 12.11.2021.

[13] Camera calibration with opencv. https://docs.opencv.org/master/dc/dbb/tutorial_py_calibration.html. Accessed: 22.07.2021.

[14] Alexey Gromov. Optimal Platform for Embedded Supercomputers. Master's thesis, ZHAW, Zurich University of Applied Sciences, Switzerland, 2020.

[15] Dávid Isztl. Development of custom FFmpeg plug-in, 2021.

[16] Raspberry pi camera specification. https://www.raspberrypi.org/documentation/hardware/camera/. Accessed: 26.07.2021.

[17] Pieye shop - nimbus 3d camera. `https://shop.pieye.org/en/Cameras/Nimbus-3D-camera.html`. Accessed: 26.07.2021.

[18] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding pcie performance for end host networking, August 2018.

[19] Wikipedia: Cas latency. `https://en.wikipedia.org/wiki/CAS_latency`. Accessed: 25.11.2021.

[20] Cudasift repository. `https://github.com/Celebrandil/CudaSift`. Accessed: 08.11.2021.

[21] N. Bergström M. Björkman and D. Kragic. Detecting, segmenting and tracking unknown objects using multi-label mrf inference. *CVIU*, nA(118), 2014.

[22] Object recognition from local scale-invariant features. `https://www.cs.ubc.ca/~lowe/papers/iccv99.pdf`. Accessed: 02.08.2020.

[23] Sift patent information. `https://patents.google.com/patent/US6711293`. Accessed: 02.08.2020.

[24] Ming Gao*, Xinlei Wang*, Kui Wu*, Andre Pradhana, Eftychios Sifakis, Cem Yuksel, and Chenfanfu Jiang. Gpu optimization of material point methods. *ACM Transactions on Graphics (Proceedings of SIGGRAPH ASIA 2018)*, 37(6), 2018. (*Joint First Authors).

[25] Svd cuda github repository. `https://github.com/kuiwuchn/3x3_SVD_CUDA`. Accessed: 11.11.2021.

[26] Bosch. Bmi160. `https://www.bosch-sensortec.com/products/motion-sensors/imus/bmi160`, 2015.

List of Figures

3.1 Concept 4

4.1 Projected dots from the LiDaR scanner of an Apple iPad Pro 2020, made visible with an infrared camera. Image source: iFixit.com 5
4.2 Euler rotations are dependent on the order of the individual rotations. Rotating around X and then Z results in a different outcome, than first rotating around Z and then X. 7
4.3 Singular Value Decomposition in 2D shown in the individual stages. Blue is before and red is after the applied transformation. 9
4.4 A Vulkan object consisting of four vertices in its local space. The green corner acts as an example. 10
4.5 The rectangle is placed with the model-matrix, while the view- and projection-matrices determine the setup of the virtual camera. The green corner point acts as the example. Note the inverted y-axis in clip space. 12
4.6 Before correction 13
4.7 After correction 13
4.8 Camera correction demonstrated at the grayscale image of the ToF camera. 13
4.9 Storing the final calibration values in two lookup tables keeps the implementation in CUDA simple. 14
4.10 The Kalman filter demonstrated. Blue is the acceleration, purple the velocity and red the position. Raw integration of the velocity shows a jagged output of the position estimation. Raw double-integration of the acceleration leads to the position to drift away. The position estimate of the Kalman filter itself follows the jagged line of the raw velocity integration, but the estimated velocity is closer to the true value. In green, the integration of the estimated velocity shows a better approximation of the true value. 15
4.11 On the left, the diagonal values of the covariance matrix of the errors are visualized against each other. The errors for velocity and acceleration converge towards a low value, while the error for the position increases. On the right, a close-up of the internal workings of the Kalman filter. The blue dots are the raw measurements, the red dots are the corresponding predictions and plotted in green is the output value of the Kalman filter. In contrast, the true velocity is plotted in grey. 16

5.1 Hardware overview. The purple objects belong to the camera head, which is connected to the processing system in cyan. 18
5.2 Frontside 19
5.3 Backside 19
5.4 The camera head consisting of the ToF camera and the Raspberry Pi camera on the front side, and the IMU, its I²C to USB bridge and the FPDLink III Serializer on the backside. 19
5.5 The Nvidia Jetson Xavier AGX on the Anyvision Baseboard. 20
5.6 Block diagram of the technically equivalent Jetson Xavier NX. Note the shared DRAM and the separated caches for CPU and GPU. Copyright by Nvidia 21
5.7 Data flow for one single frame. 21
5.8 Software overview. The routine inside the Vulkan Framework creates four child classes. 22
5.9 Wireframe rendering of the reference image I_{Ref} provided by the ToF camera 23
5.10 Left, the uncorrected ToF image I_{Any}, in the middle the corrected image I_{Corr} and on the right, the infrared grayscale image of the scene. To make the effect more apparent, the brightness accross the red lines have been plotted. 24
5.11 First step of ToF motion estimation: Extract SIFT features and brute-force match with prior image. Note that the brute-force matcher also generates false matches. 25
5.12 Second step of ToF motion estimation: Map features into 3D space 26

5.13 First step of the 3D RANSAC: The SVD method calculates the rotation and translation from three randomly assigned points. This step is performed in parallel on multiple groups of three. Each group P_i generates the rotation R_i and translation $\vec{t_i}$. 29
5.14 Second step of the 3D RANSAC: The calculated rotations and translations get evaluated against the whole dataset. In the example, the fourth matching point fulfills the criterion. As this group is the best possible in the example, an improved rotation $R_{improved}$ and translation $\vec{t}_{improved}$ is calculated from these. 29
5.15 Third step of the 3D RANSAC: With $R_{improved}$ and $\vec{t}_{improved}$ being present, the brute-force matches get discarded. Applying the currently available rotation- and translation-information the the point clouds allows re-matching based alone on the position. Theoptimal rotation R_{opt} and translation ${}_{v}ect_{opt}$ result from these feature pairs. 30

6.1 Sensor values measured against a meterstick. In red, the linear fit generated in Microsoft Excel. 36
6.2 Image 37
6.3 Cloud 37
6.4 The scene reconstruction from the left image to the full point cloud. The red line is equivalent in both figures. Each green dot in the image on the left is a SIFT feature. The red dot in the point cloud is the position of the camera. 37
6.5 Plotting the feature matching performance against different thresholds shows the quality of the RANSAC feature matcher. In blue, the raw number of unmatched features in each test, in gray the brute force matches and in red the RANSAC matches. 38
6.6 Two raw point clouds with the found matches connecting them. The camera is rotated in between the frames, which leads to the displacement of the point cloud. The connecting lines are roughly 10cm long, depending on the position. The large red dot in the foreground is the position of the camera. The scene is the same as in image 6.4, but zoomed for better visibility. 39
6.7 The individual axis of the ToF rotation quaternion plotted against the gyroscope quaternion. The camera head got rotated in each direction after another. 40
6.8 Raw measurement of the translation alongside the x-axis. Before and around 50, the camera is slid fast forward, kept in pause just to slide back before reaching frame number 100. After 100 and before 220, the same motion is repeated but slower. Blue is the acceleration and purple the velocity. 41
6.9 Raw measurement of the translation alongside the y-axis. Between the frames 25 and 50, the camera is slid to the right, kept there and slid back to the origin around frame 75. The motion is repeated slower after frame 100 and before 200. Blue is the acceleration and purple the velocity. 41
6.10 Raw integration of the relevant axis in both translations of the ToF velocity data. Red: ToF velocity, Blue: Its integration. 42
6.11 Raw integration of the relevant axis in both translations of the accelerometer. Blue is the acceleration, purple the velocity (single integration) and red the position (double integration). 43
6.12 The output of the Kalman filter is plotted in green, against the gyroscope data in blue and the ToF camera data in red in the upper three graphs. The fourth graph shows the calculated rotation. The a-axis in red, the b-axis in purple and the c-axis in blue. 44
6.13 Initial situation 44
6.14 After rotation 44
6.15 Demonstration of the rotation against the camera image with the moving projection. The displacement of the rectangle inside the picture frame is a result of the Raspberry Pi camera facing translational motion when rotated on the camera tripod. 44
6.16 Motion in y direction. Blue lines are data from the accelerometer and its integrations, red are the velocity from the ToF camera and its integration and in green are the different Kalman outputs. 45

List of Tables

7.1 List of Abbreviations 52

List of Abbreviations

Abk.	Abbreviation
ZHAW	Zurich University of Applied Sciences
InES	Institute of Embedded Systems
HPMM	High Performance Multimedia - a research group ant ZHAW-InES
GLFW	Graphics Library Framework
GLM	Graphics Library Mathematics
OpenGL	Open Graphics Library - Predecessor of Vulkan
OpenGL	Open Computing Language - General-purpose computation on GPU
CUDA	Compute Unified Device Architecture - NVIDIAs General-purpose computation on GPU
CSI2	Camera Serial Interface 2 - Video input interface
V4L2	Video for Linux 2 - Videostreaming framework on Linux

Table 7.1: List of Abbreviations

8 Appendix

8.1 Structure of the Git-Repository

8.1.1 Documentation

Contains this documentation and all the images needed to build it.

8.1.2 Source

Contains all the source code, subdivided in Folders for device drivers, middleware and userspace-software

8.1.3 Literature

Contains 3rd party documents for technologies used in this thesis

8.2 Used software

The following software has been used for this thesis:

- Visual Studio Code: Software development on Linux, Jupyter-Notebooks for simulation and LaTeX for this documentation.
- Python 3.8 - with the Python-Plugin for VS Code.
- MikTex - with the LaTeX-Workshop-Plugin for VS Code.
- Adobe Illustrator 2021: Creating visuals for this documentation
- Adobe Photoshop 2021: Creating visuals for this documentation
- Grammarly Premium: For spellchecking this thesis
- Microsoft OneNote: As a notepad
- Microsoft Excel: For evaluation of the benchmarks
- NVIDIA JetPack 4.4: For flashing the TX2 and profiling the application