Zurich University
of Applied Sciences

School of
Engineering
InES Institute of
Embedded Systems

Augmented Reality Platform using Sensor Fusion and embedded GPU Processing

Zurich University of Applied Sciences

Institute of Embedded Systems

Authors	Marcel Wegmann

Version	1.0
Last changes	December 23, 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

Augmented Reality is the concept of enhancing the real world with virtual objects or information with projections into a viewfinder or through specialized goggles. Simpler forms of Augmented Reality – like a heads-up display in a car – do not need to estimate the camera's motion, an object, or the user. However, more elaborate implementations of Augmented Reality need to track things and, more importantly, the camera's movement itself. The applications in which Augmented Reality could be leveraged range from social interaction over pedestrian navigation to various use cases in different professions. Multiple companies already have shown closed source or custom-tailored programming interfaces, either running on smartphones or shipped with industry-targeted goggles. The tracking of real-world objects surfaces or is possible with the provided interfaces, but the algorithms behind the different functions are corporate secrets.

This thesis describes an approach for an end-to-end pipeline in a prototype of an Augmented Reality platform without using commercial interfaces. A time-of-flight camera provides a depth image that allows reconstruction of the recorded scene as a cloud of SIFT features. Frame-by-frame analysis of the point cloud estimates the camera's motion by highly parallel processing and a three-dimensional extension of the RANSAC algorithm. An accelerometer and a gyroscope provide additional data, fused with a Kalman filter to improve the motion estimation. A regular color camera acts as a viewfinder, and Vulkan renders the result to a monitor.

Enhancing the matching quality of SIFT features between consecutive frames of a time-of-flight camera using a three-dimensional RANSAC algorithm led to over two times as many correct matches. Despite noisy camera data, the estimation of the camera rotation and translation of the time-of-flight camera based on these matches works as demonstrated in the thesis. The sensor fusion with the Kalman filter works as intended for rotations. Still, it fails for translations because of the system's low sampling rate and the accelerometer's hysteresis, failing to compensate appropriately for gravity.

Preface

I am thankful to my supervisor, Prof. Dr. Matthias Rosenthal, for allowing this deep dive into Augmented Reality and the support given during this thesis. Furthermore, I am grateful for the support and advice from the ZHAW InES HPMM team.
Special thanks, especially to Lukas Neuner, for his valuable inputs during this thesis and for proofreading this document. Additional thanks are given to Alexey Gromov for his support in setting up the Jetson Xavier and proofreading this document. I am thankful for the chance of gaining further experience in CUDA, Vulkan and for the time given to learn new topics in the math involved in three-dimensional rendering.
In addition, I thank all my friends and family members for giving support and motivation.

Contents

1 Introduction

Augmented Reality - or AR – is the concept of projecting virtual objects into the real world. Phone screens, tablet computers, and specialized goggles render virtual objects over the camera image or display them on translucent screens. A form of Augmented Reality is a heads-up display, for example, in cars to project the current speed and navigation information to the windshield or in airplanes for comprehensive avionic information.
In contrast, Virtual Reality – or VR – limits itself to entirely virtual worlds, into which the user dives. While Virtual Reality is already readily available through off-the-shelf goggles, which lets users meet other people and play games in virtual worlds, Augmented Reality is mainly limited to smartphone applications. Currently, it lacks readily available and specialized hardware, other than niche products specialized for specific industries.
Augmented Reality faces numerous technical challenges. Projecting a virtual object – for example, a flowerpot – into the real world requires the system to recognize a table and find an unoccupied location. Apart from placing decoration or furniture, a pair of Augmented Reality goggles could project helpful information into the air. A mechanic could have virtual schematics or instructions floating beside his work, while another virtual monitor displays a video phone call with the customer. Hand detection and gesture control would enable interaction with virtual objects. Another example could be pedestrian navigation, projecting arrows to the street.
AR goggles need to react in real-time to any motion of the user's head. Any latency would break immersion as virtual objects lose the connection with their anchor point in the real world. A flowerpot would jump on the table, and arrows on the street would start to float and collide with walls. Fast and reliable motion tracking of the system itself is vital for avoiding visual glitches.

1.1 Initial Situation

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 Augmented Reality and Virtual Reality, and hardware teams account for more than 6000 employees [3].
The most advanced hardware available today is the Microsoft HoloLens [4] featuring four cameras for motion tracking, two infrared cameras for eye-tracking, a time-of-flight sensor for depth measurement, a 9-Axis IMU (Accelerometer, Gyroscope, and Magnetometer), and an additional camera for photos and recording videos. Microsoft targets a professional environment with the Holo Lens, for example, to support Airbus technicians at maintenance [5].
The Implementation of how these sensors are fused for generating a smooth experience is closed source.
Other than the HoloLens, there exist numerous applications on Smartphones and Tablets running either Android, iOS, or iPadOS. Google ARCore implementation relies on Machine Vision and can benefit from – but does not require – a ToF camera. ARCore detects objects, estimates their size, tracks objects, and finds flat surfaces. The analysis of light sources allows a virtual object to cast a realistic shadow. Apple's counterpart is ARKit, which leverages machine learning hardware on their *Bionic* SoCs and the LiDaR sensors used in iPads and iPhones. [6]
Next to these two, numerous open-source projects focus on specific do-it-yourself projects and, for example, enable AR capabilities in a web browser through JavaScript [7] [8].

1.2 Motivation

While advanced solid-state LiDaR scanners used in consumer products were not available on the open market in finished modules, time-of-flight cameras get marketed directly. Multiple companies sell time-of-flight cameras for industrial processes or on evaluation boards for development. Building a prototype becomes viable with off-the-shelf time-of-flight cameras and powerful embedded devices, like the Nvidia Jetson series. With the availability of time-of-flight cameras, it becomes worthwhile investigating the possibilities they offer for motion detection and if the acquired data could be helpful in an Augmented Reality system.

1.3 Scope

This thesis focuses on close-to-hardware fundamentals and omits the implementation of end-user applications. Without solid motion tracking of the Augmented Reality system, all the other parts of Augmented Reality fail. Developing specialized goggles is pointless without reliable motion tracking; this also falls out of the scope.

1.4 Goals

The goals of the thesis are to set up an end-to-end pipeline from data recording over processing to rendering, involving a three-dimensional object which follows the motion of a camera. An off-the-shelf time-of-flight (ToF) camera delivers depth information from which the camera motion is estimated. A sensor fusion approach will combine the estimated camera motion with an IMU.

1.5 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 Concept

A working prototype of an AR system consists of two key components: Awareness regarding the environment and awareness regarding the motion of a recording 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 rectangle accordingly, as visualized in Figure 2.1.

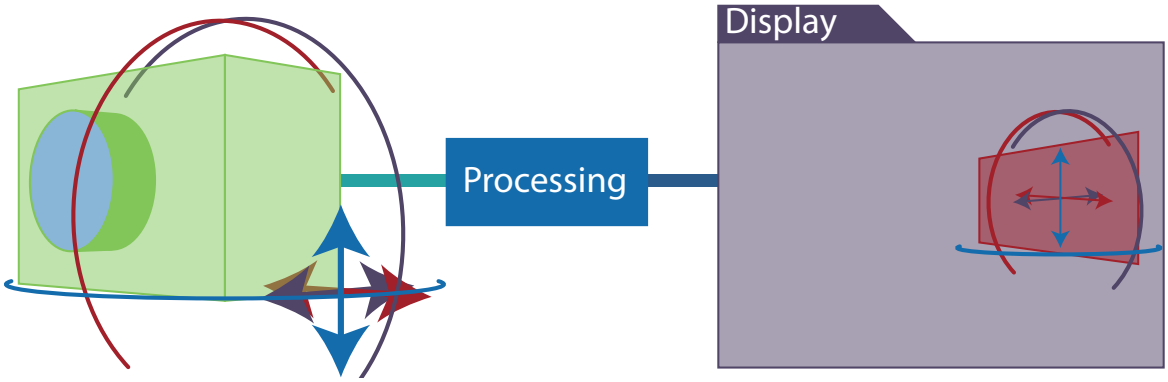


Figure 2.1: A free moving camera films the surroundings, that are displayed on the purple screen. Motion of the camera shall alter a virtual surface, projected into the display image, as if the virtual surface is stationary, despite of the motion of the camera.

The prototype is developed on a Nvidia Jetson Xavier AGX 8GB system with an attached accelerometer- and gyroscope-sensor, a time of flight (ToF) camera, and a Raspberry Pi Camera.
The three-dimensional information from the ToF camera allows extracting motion information on a frame-by-frame basis. The accelerometer complements the motion information given by the camera, especially at very fast motions. The Raspberry Pi camera acts as a viewfinder, rendered in full-size to the purple area in Figure 2.1.
The projected rectangle acts as a secondary screen, allowing to display any image, the output of an additional camera or debug-images.

3 Fundamentals

The following Chapter describes methods and technologies used within this thesis.

3.1 3D Cameras

In 3D mapping, two expressions often get mentioned: Light-Detection and Ranging (LiDaR) sensors and Time of Flight (ToF) cameras. As the basic principle in both technologies relies on measuring the Time of Flight 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. [9] 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 Figure 3.1. The time of flight for each dot is measured individually.

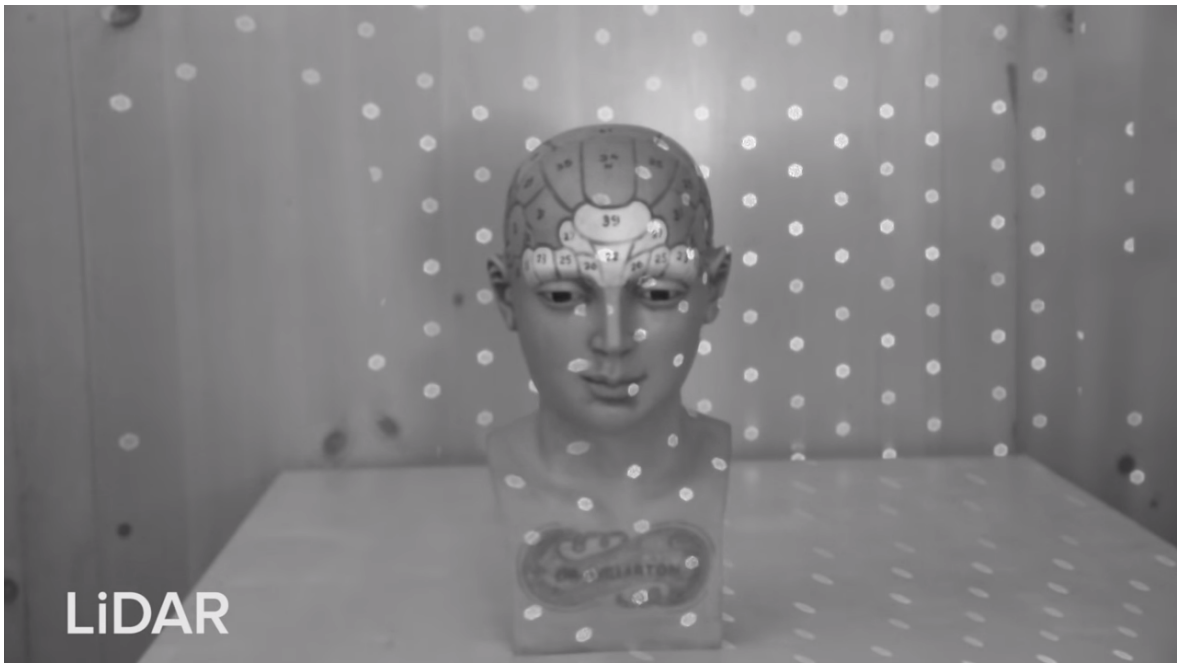


Figure 3.1: Projected dots from the LiDaR scanner of an Apple iPad Pro 2020, made visible with an infrared camera. Image source: iFixit.com

Another method for 3D mapping is using one wide-area infrared flash and measuring the time of flight on each pixel of a camera sensor. This approach, in contrast to the laser based LiDaR scanner, 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. A ToF camera also works as an infrared grayscale camera, providing an image by itself.
The sensor used for this thesis follows the principle of a ToF camera with a wide area infrared flash. 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 4.3.1 explains the performed correction.

3.2 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 microelectromechanical system (MEMS) containing a gyroscope and an accelerometer provides rotation and acceleration information to the system to assist the positional tracking.

3.2.1 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-axis are

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

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. A chain of rotations around X, Y, and Z results in a single rotation around an arbitrary axis for a specific angle using the matrix

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

Applying this transformation to each vertex of a virtual 3D object results in a rotation of the whole object around the origin $(0,0,0)$ with the equation

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

Moving an object in space without any rotation is generally done with a vector addition, creating an inhomogenous linear equation. Adding a fourth dimension allows packing the spatial translation into the matrix. 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 with the equation

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

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 Figure 3.2: The values returned by a gyroscope need to be applied all at once and not one after another. Replacing rotation matrices by quaternions, described in Section 3.2.2, solves this problem.

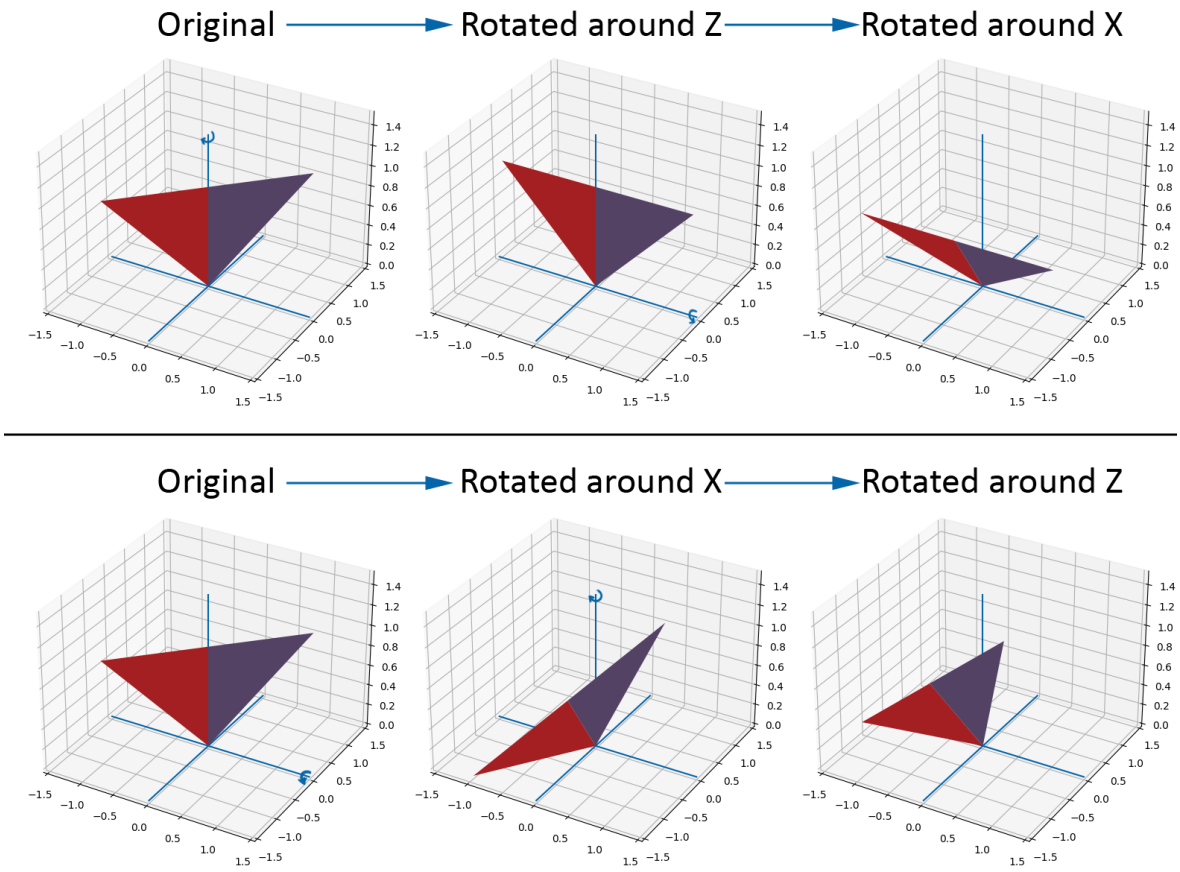


Figure 3.2: Euler rotations are dependent on the order of the individual rotations. Rotating around X and then Z results in a different outcome, than first rotating around Z and then around X.

3.2.2 Rotation with Quaternions

Quaternions - also known as "Hamilton Numbers" - are the four dimensional equivalent to complex numbers. 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 $\mathbf{i}^2=\mathbf{j}^2=\mathbf{k}^2=\mathbf{ijk}=-1$, $\mathbf{ij}=\mathbf{k}, \mathbf{jk}=\mathbf{j}, \mathbf{ki}=\mathbf{j}$, and $\mathbf{ji}=-\mathbf{k}, \mathbf{kj}=-\mathbf{i}, \mathbf{ik}=-\mathbf{j}$. With these definitions, the quaternion multiplication,

also named Hamilton Product, is non-commutative. The Hamilton Product is

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

The neutral element, analogue to an identity matrix, of the quaternion multiplication is:

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

A 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, analogue to a rotation matrix' determinant. 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 multiplied from the left and conjugated and multiplied from the right. Conjugation of a unit quaternion is done by flipping the sign in each imaginary part. The result

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

is the rotated vector

$$\vec{v} = \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 0 \\ a\mathbf{i} \\ b\mathbf{j} \\ c\mathbf{k} \end{pmatrix}.$$

By converting a rotation quaternion to a rotation matrix, the gap to the standard linear algebra can be bridged. This formula only applies to true rotation quaternions, therefore the norm must 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.

3.2.3 Singular Value Decomposition (SVD)

Any matrix multiplication is a combination of rotations and stretching. The singular Value Decomposition allows splitting any matrix into three fundamental transformations, either pure rotations, reflections or pure stretchings. The SVD is a required step in finding the rigid motion of a point cloud [10].
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 $m \times m$. [11] 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 Figure 3.3 on the next page, the unit circle $\vec{x}$ first gets rotated by V^T that the stretching is in the directions of the coordinate system. After applying the stretching Σ to the rotated circle $\vec{y}_1$, the intermediate ellipse $\vec{y}_2$ gets rotated and reflected to match $\vec{y}$. The following recipe [11] shows how to find the matrices of the singular value decomposition. The matrix

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

that is used in Figure 3.3 gets decomposed. 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:

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

The determinant of U and V allows checking if they are rotations or reflections. A rotation has a determinant of 1, a reflection a determinant of -1.

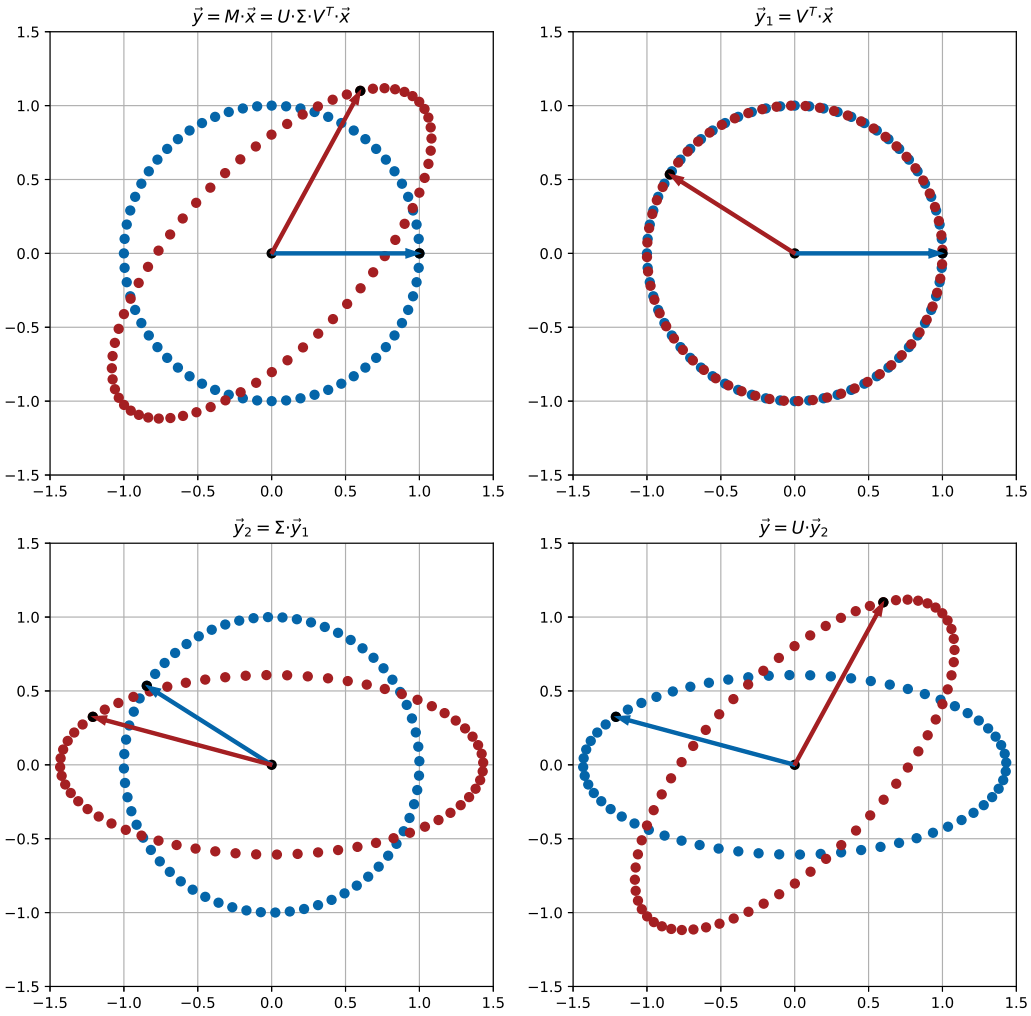


Figure 3.3: Singular Value Decomposition in 2D shown in the individual stages. Blue is before and red is after the applied transformation. On top left, the entire matrix operation is demonstrated. The figure on top right demonstrates the first rotation and the figure on bottom left shows the stretching. The final reflection of the stretched ellipse is shown on bottom right.

3.2.4 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} of the camera.

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

Section 3.2.2 describes how to carry out the mathematics of this quaternion multiplication.

3.3 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 of three-dimensional vectors $\vec{v_v} = (x_v, y_v, z_v)$. The subscript v denotes the vertex coordinates which reside in the object's local space. These coordinates usually do not contain data regarding the whole object's scale, position, and rotation in 3D space.

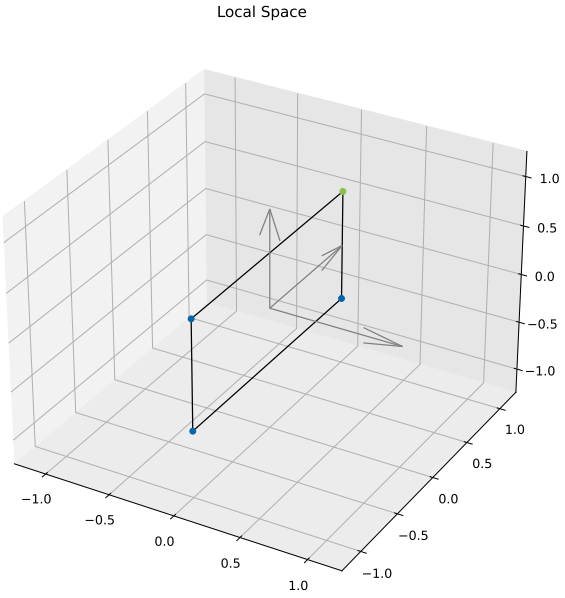


Figure 3.4: A Vulkan object consisting of four vertices in its local space. The green corner acts as an example in the ongoing section.

Vulkan expects the output of the shader step to be in clip coordinates. The clip coordinates, denoted with the subscript c, reside in the clip space. Clip coordinates are four-dimensional vectors $\vec{v_c} = (x_c, y_c, z_c, w_c)$ 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 world space is the coordinate system in which multiple objects and virtual cameras get placed, to assemble the scene. The model matrix is a standard 4x4 rotation and translation matrix as explained in Section 3.2. In the example, shown in the Figures 3.4 and 3.5, the rectangle is shifted on the x-axis.

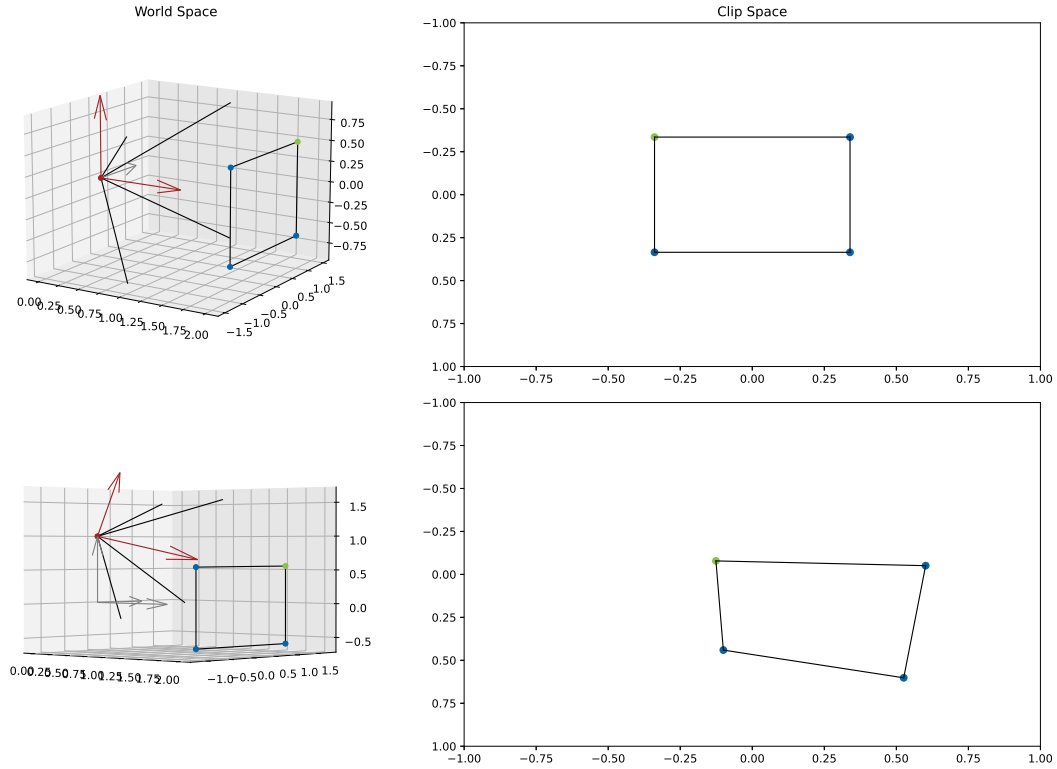


Figure 3.5: The rectangle is placed with the model-matrix, while the view- and projection-matrices determine the perspective. The green corner point acts as the example. Note the inverted y-axis in clip space.

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

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 [12] offers the "4x4_look_at" function to calculate the view matrix based on a virtual camera at a specific position position, a viewing-, and an upwards-direction, as visualized in red in Figure 3.5. In the upper half of the figure, 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 figure, 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 $A_{Projection}$ 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 Figure 3.5, 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 by 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. 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, therefore clipped away, hence the name of the coordinate space.

3.4 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 is modeled as

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

Similarly, tangential distortion is modeled as

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

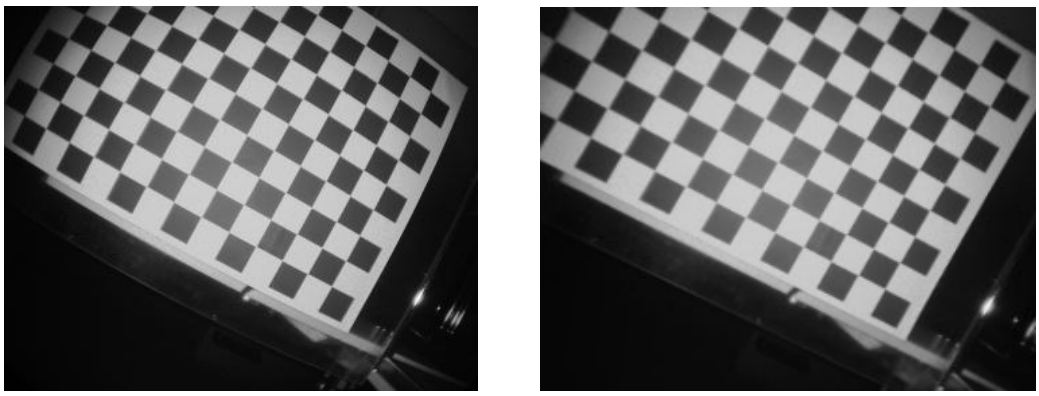
In these equations, r is the euclidian distance between the distorted image point and the distortion center.

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

Therefore, to compensate for the lens distortion, the five coefficients k_1, k_2, k_3, p_1 and p_2 need to be estimated by calibration. In addition, the effect of the focal length f and the optical center c get expressed as a 3x3 matrix.

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

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.



(a) Before correction (b) After correction

Figure 3.6: Camera correction demonstrated at the grayscale image of the ToF camera.

The implementation in CUDA stores 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 provides a direct coordinate mapping with lookup tables as shown in Figure 3.7.

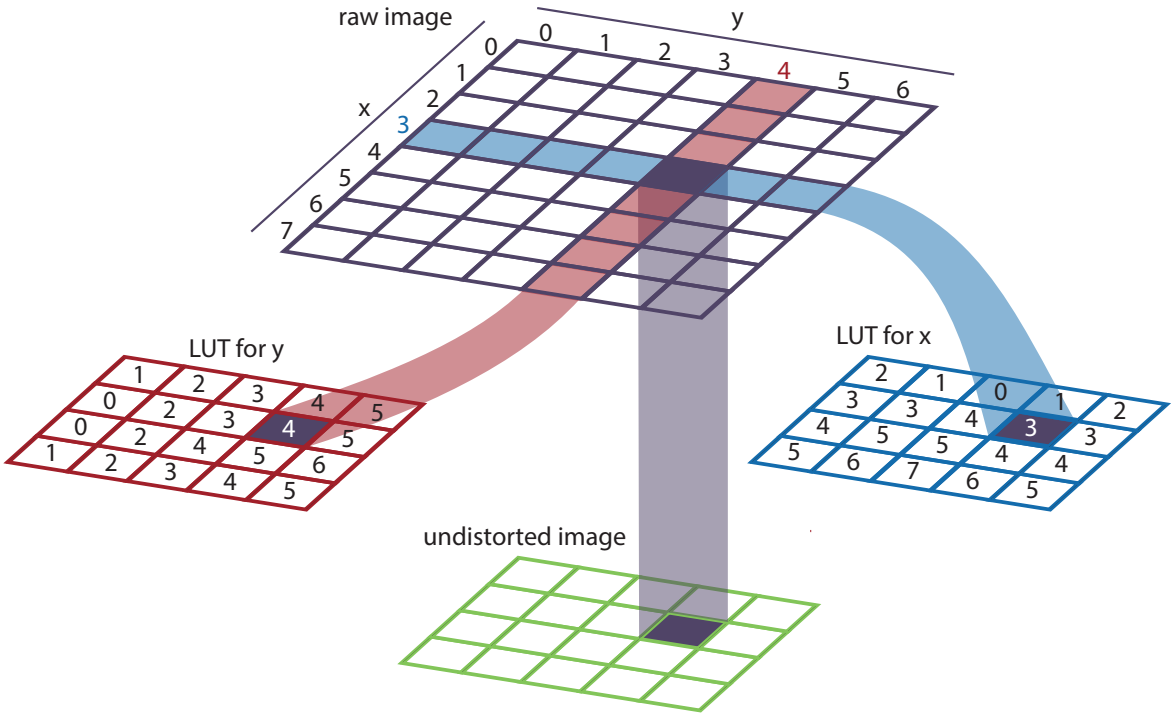


Figure 3.7: Applying the lens correction with the use of lookup tables (LUT) in CUDA.

3.5 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 an attached motor.
In this thesis, as the camera head will be moving freely, motion without a known input needs to be measured, having sensors for acceleration, velocity, and rotation. The following example demonstrates the position p, velocity v, and acceleration a in one dimension. In Figure 3.8 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.
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 models that. Without a known input, B and $\vec{u}$ are omitted and the system is entirely model based. The system state $\vec{x}$ transitions from the prior state $k-1$ to the prediction $k|k-1$.

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

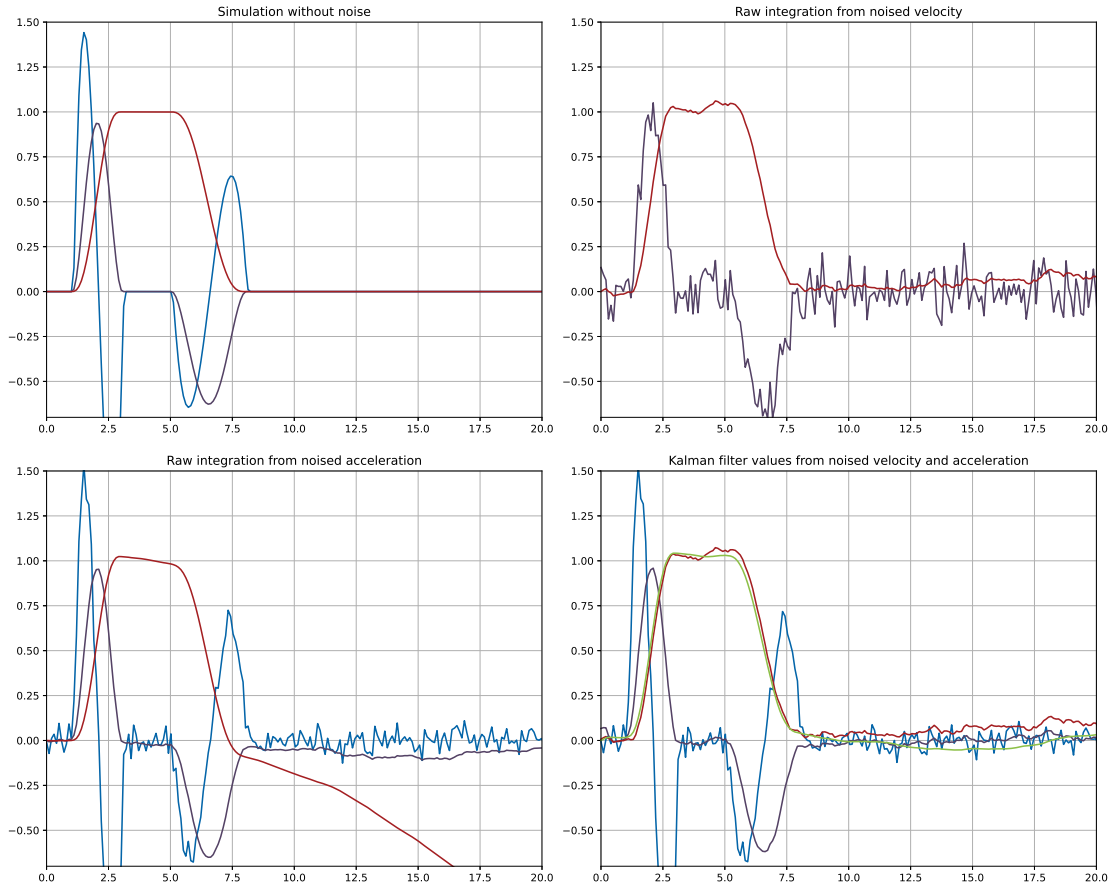


Figure 3.8: The Kalman filter demonstrated. Blue is the acceleration, purple the velocity and red the position. Raw integration of the velocity, on top right, shows a jagged output of the position estimation. Raw double-integration of the acceleration, on bottom left, leads to the position to drift away. The position estimate of the Kalman filter, on bottom right, 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.

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.
The evolution of the covariance matrix of the errors is :

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

with the gaussian process noise N. Assuming that an external force $\vec{f}$ acts on the system, which is constant during the sampling intervals Δt. The external force influences the acceleration, the velocity, and ultimately the position proportional to the input vector

$$G = \begin{pmatrix} \frac{\Delta t^2}{2} \\ \Delta t \\ 1 \end{pmatrix}.$$

The process noise matrix N 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 used process noise model is the piecewise white

noise model.

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

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 needs 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.

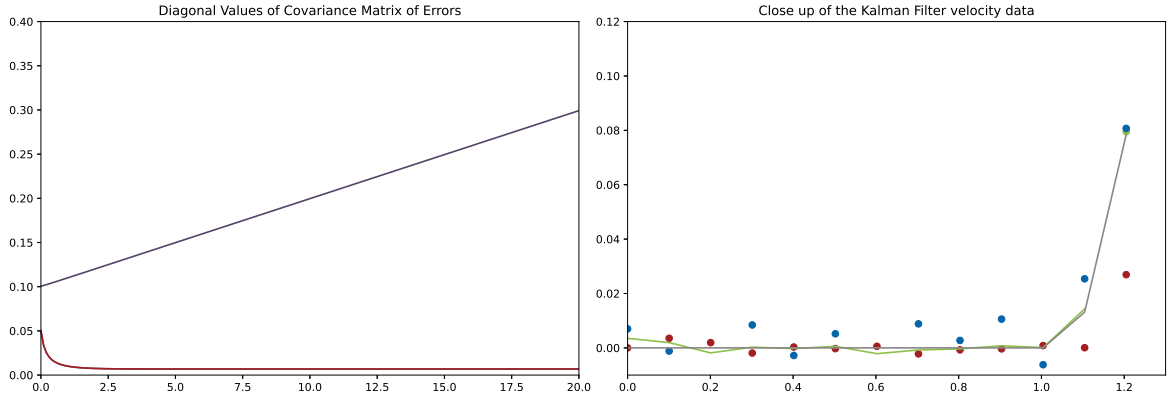


Figure 3.9: On the left, the diagonal values of the covariance matrix of the errors are visualized against each other. The errors for velocity and acceleration converge towards a low value, while the error for the position increases.
On the right, a close-up of the internal workings of the Kalman filter. 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 $\vec{x}_{k|k-1} = F \cdot \vec{x}_{k-1}$ and the covariance matrix $P_{k|k-1} = F \cdot P_{k-1} \cdot F^T + N$ being predicted, the following steps describe the correction part of the Kalman iteration. The observation matrix H determines the sensor influence on 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.

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

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 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.

4 Implementation

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

4.1 Hardware

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

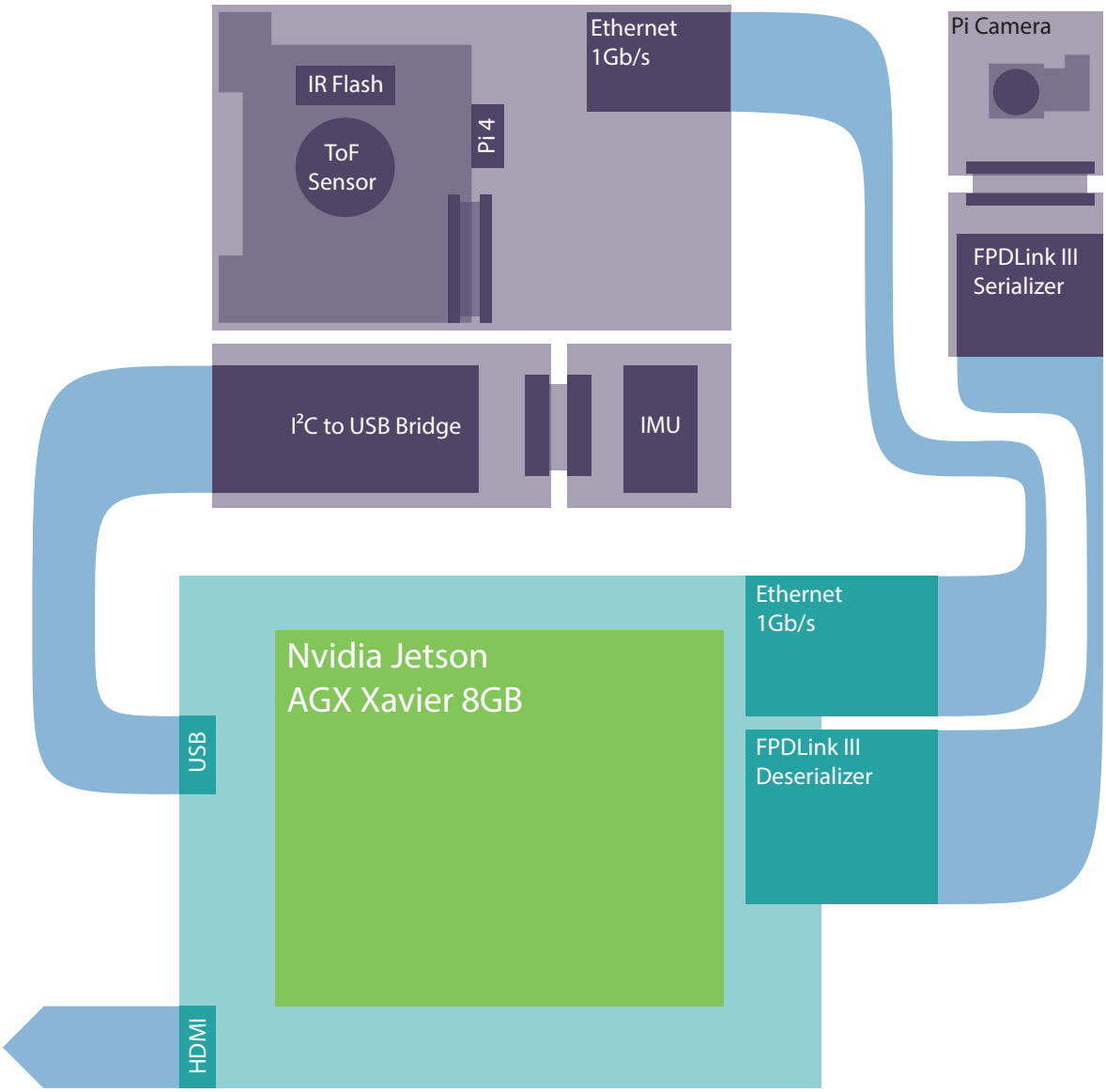


Figure 4.1: Hardware overview. The purple objects belong to the camera head, which is connected to the processing system in cyan.

4.1.1 Evaluation of ToF Camera

For the evaluation of the ToF camera, cost and availability have been the main factors in the evaluation, with it being directly CSI-2 connected being an important feature.
Internet research has shown a handful of different sensors powering multiple products of diverse manufacturers in varying price ranges.

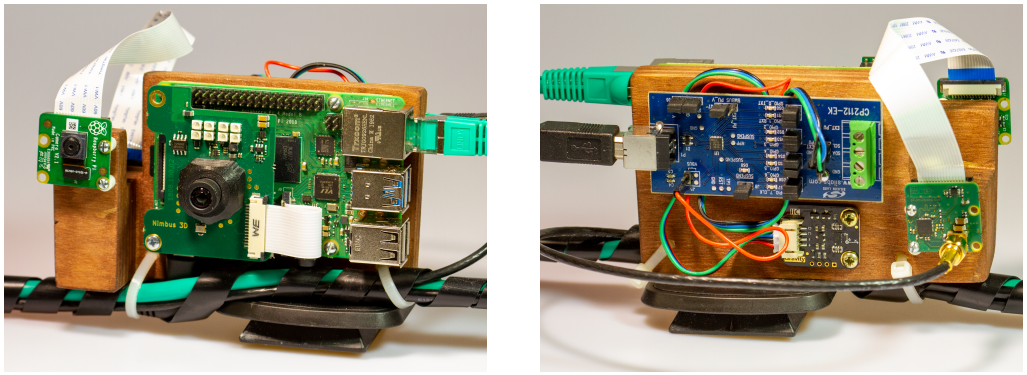
Infineon REAL3 IRS1125 (Chosen)
The Infineon IRS1125 ToF Sensor is available in a USB-based development kit by pmdtec – an integrator for ToF technology into smartphones – and on the affordable PiEye Nimbus 3D camera, which is sold as a Raspberry Pi accessory and used in this thesis.
The sensor features 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 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.

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.

Other image sensors
During the internet research, the Texas Instruments OPT8241 was discovered. 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.

4.1.2 Camera Head

The camera head, shown in Figure 4.1, 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 mounted to a plywood structure glued onto a tripod-baseplate, shown in Figure 4.2.



(a) Frontside (b) Backside

Figure 4.2: 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.

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. Zip ties through holes in the plywood structure help managing the cables and relieve stress on the connectors.

4.1.3 Processing System

An Nvidia Jetson Xavier AGX in the 8GB version carries out the processing, rendering, and data acquisition. An Anyvision baseboard, shown in Figure 4.3, 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]

4.1.4 Video inputs

The Sony IMX 219 based Raspberry Pi Camera v2.1 features a color image with a 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; therefore, it utilizes the whole sensor size and offers the full field of view.
The Raspberry Pi Camera v2 is a Mipi CSI2 attached camera module with an FPDLink III serializer and deserializer, to extend the cable length.

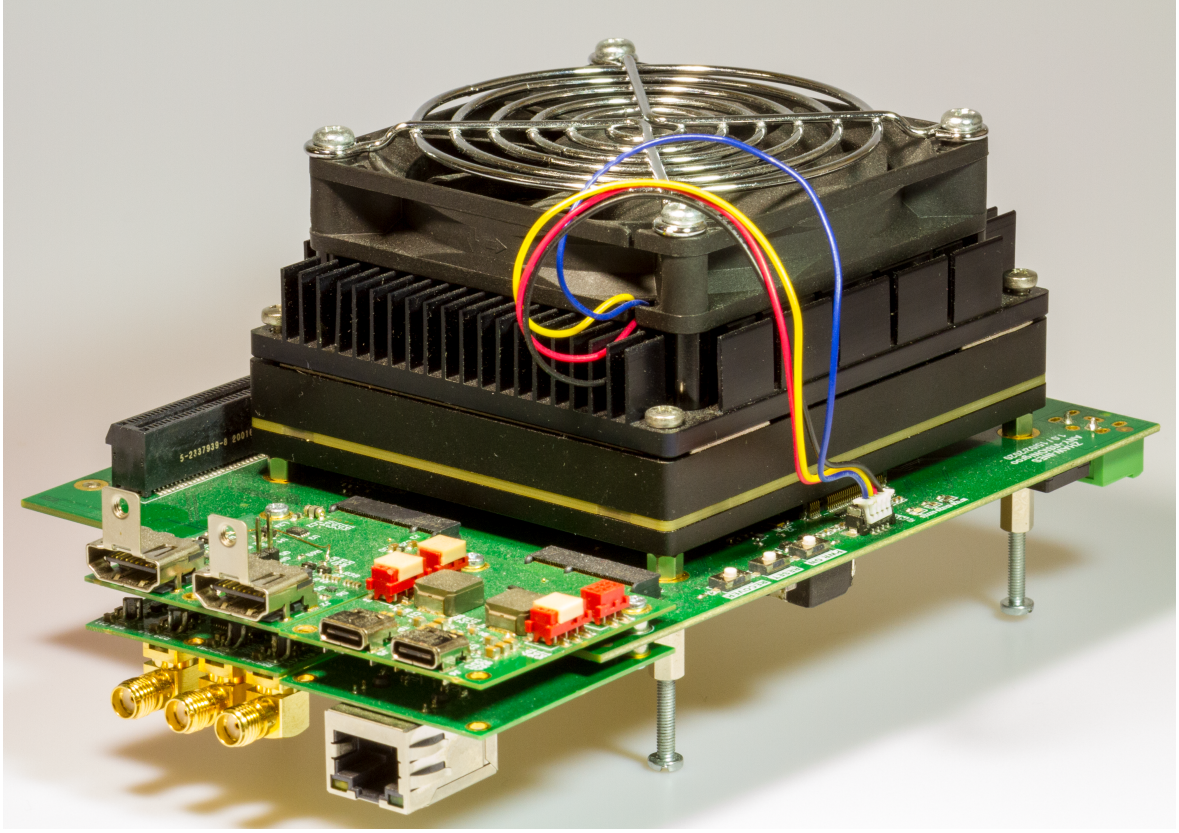


Figure 4.3: The Nvidia Jetson Xavier AGX on the Anyvision Baseboard.

The entire ToF software stack runs on a Raspberry Pi that sends its video data via Ethernet to the augmented reality system.

4.1.5 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 technical 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 [17], which also varies with traffic on the system. In comparison, a DDR4 SDRAM access has a latency of around 10-15ns [18]. Disabling the caching on the memory - as required for concurrent memory access - worsens the latency. Therefore, the optimal way to share data between the CPU and the GPU in a standard desktop computer is to copy the data 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.

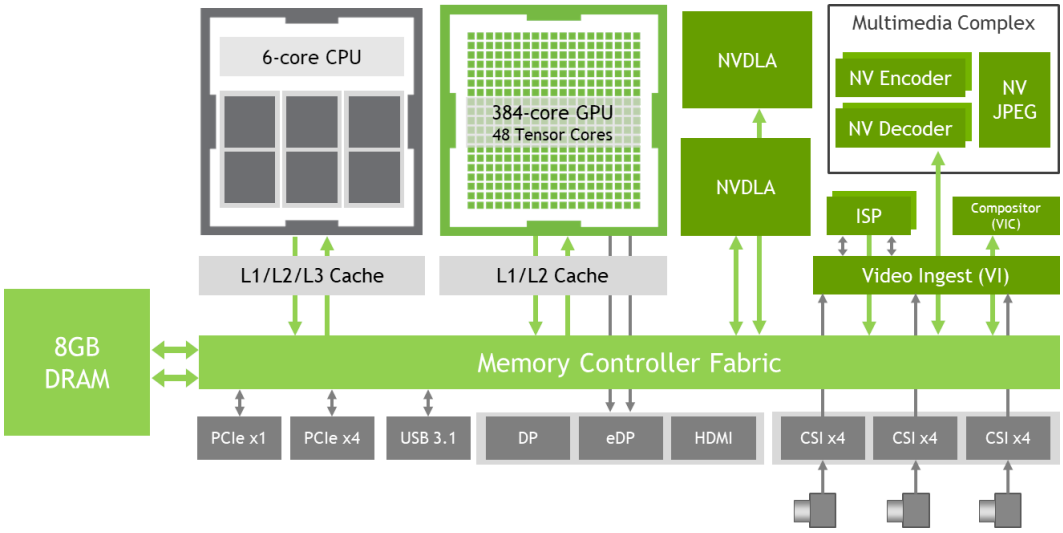


Figure 4.4: Block diagram of the technically equivalent Jetson Xavier NX. Note the shared DRAM and the separated caches for CPU and GPU. Copyright by Nvidia.

4.2 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 Figure 4.6, 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 4.7. 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.

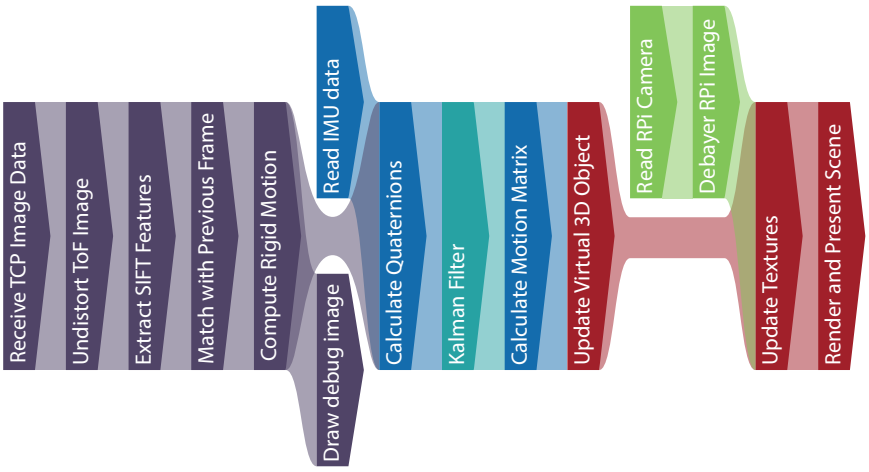


Figure 4.5: Data flow for one single frame. The colors are the same as in the software architecture overview in Figure 4.6

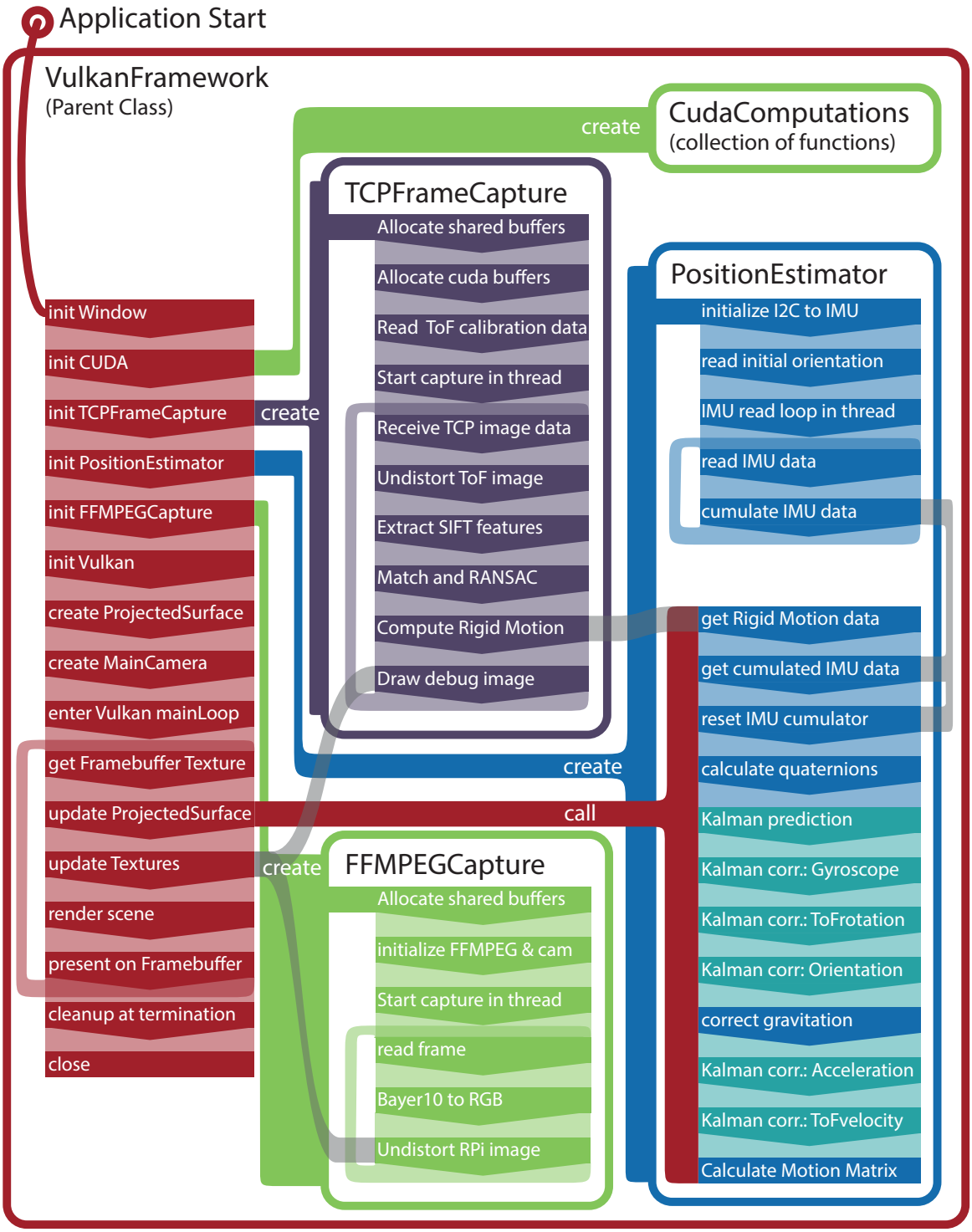


Figure 4.6: Software overview. The routine inside the Vulkan Framework creates four child classes.

Figure 4.5 shows the order in which a single time of flight frame, and a single Kalman iteration, is ordered. The IMU data cumulator, the FFMPEG capture for the Raspberry Camera and the Vulkan mainloop run at faster rates.

4.3 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 Figure 4.6 shows the location of the algorithms within the software.

4.3.1 ToF Camera calibration

On the ToF Camera, two parts need to be calibrated: The optics, as described in Section 3.4, 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 Figure 3.6. As the process of lens correction cuts off parts of the image, the image size gets reduced to 265 x 205 pixels.
For the distance measurement, first the radial distances need to be flattened as described in Section 3.1. As the angle α is not known for each pixel, a reference measurement is required. To reduce noise, 19 images of the same flat wall has been taken, smoothed by a two-dimensional gaussian filter and averaged onto one reference image I_{Ref}. The wireframe image in Figure 4.7 shows the curvature of this reference image. The rounding at the edges is an artifact of the gaussian smoothing.

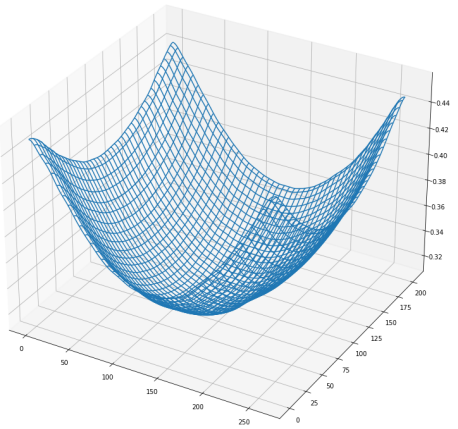


Figure 4.7: Wireframe rendering of the reference image I_{Ref} provided by the ToF camera

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

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

Pixel by pixel multiplication of any other image I_{Any} with I_{cos} will correct the influence of the radial measurement as shown in Figure 4.8.

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

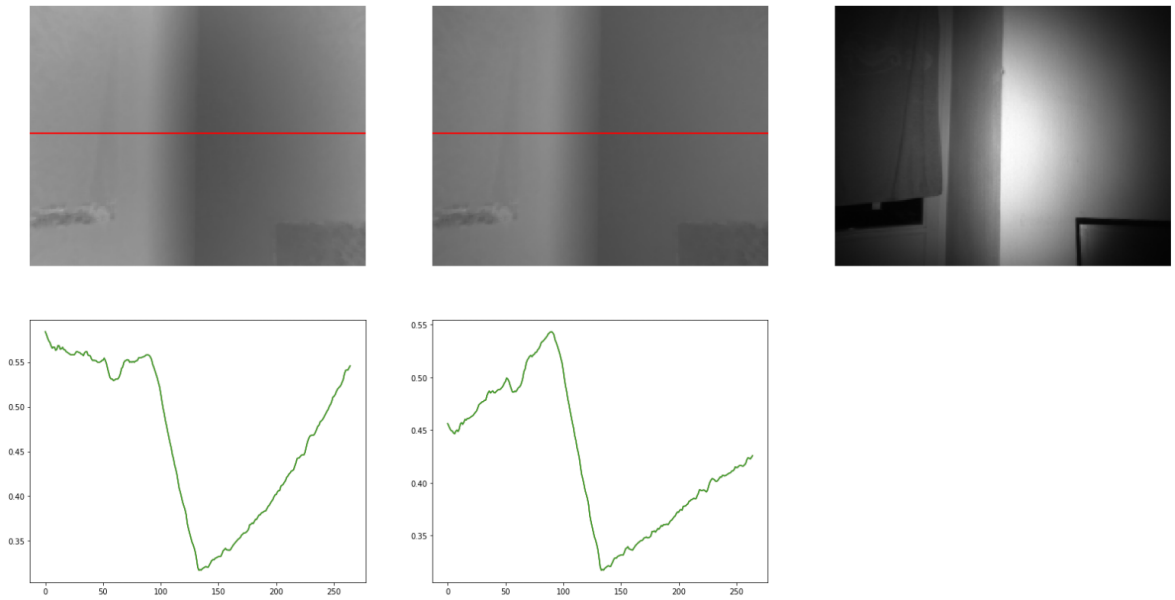


Figure 4.8: Left: uncorrected ToF image I_{Any} Middle: the corrected image I_{Corr} Right: the infrared grayscale image of the scene. To make the effect more apparent, the brightness accross the red lines have been plotted.

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

4.3.2 SIFT feature extraction

The PiEye Nimbus ToF camera generates three different image channels for each picture: The depth map, the confidence, and the greyscale infrared image. By correcting the lens distortion as described in Section 3.4, 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 3.1, flat surfaces in the real world are also flat on the depth map. The implementations of both corrections are explained in Section 4.3.1.
The points of the point clouds need to be matched to estimate rotation and translation between two consecutive ToF images. The CudaSift library [19] [20] extracts SIFT [21] features on each greyscale infrared image of the ToF camera. Extraced features of the frame k, get brute-force matched with the features of the prior frame $k-1$, as visualized in Figure 4.9. SIFT features were chosen because of the author's prior experience, and it is a well-known feature extraction algorithm whose patent expired. [22]

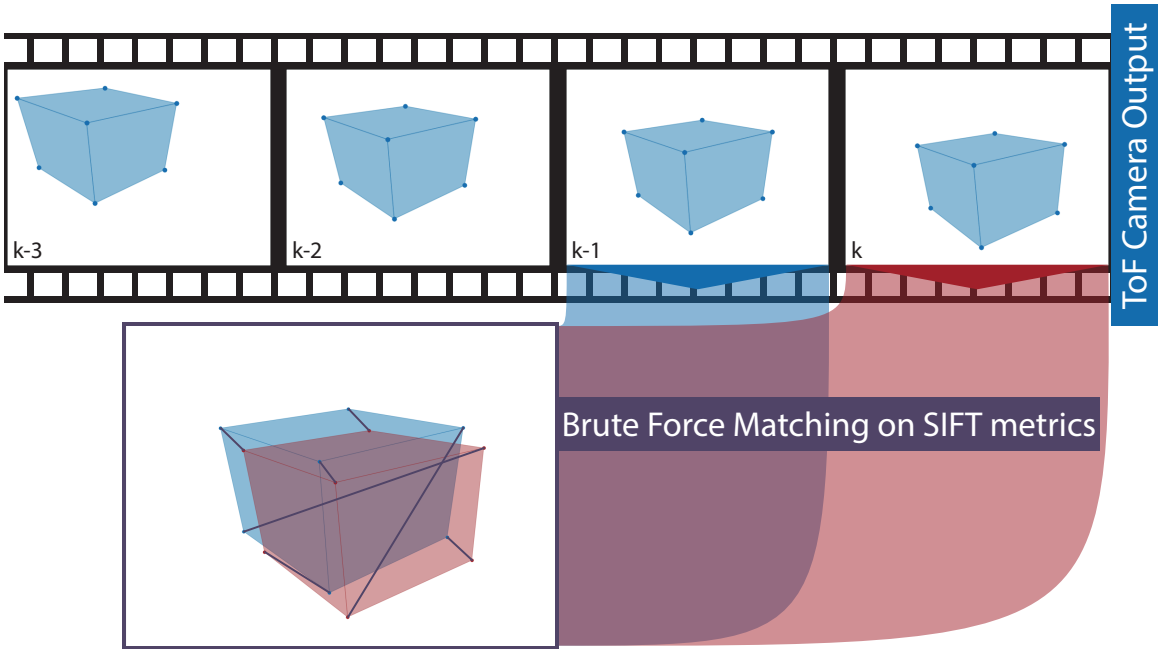


Figure 4.9: First step of ToF motion estimation: Extract SIFT features and brute-force matching with prior image. Note that this brute-force matcher also generates false matches.

Each feature coordinate on the picture gets mapped to the 3D space to generate a 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 Figure 4.10. Please note the coordinate convention described in Section 3.2.4. The 3D coordinates a, b and c correspond to the image coordinates u and v via

$$a = d \quad , \quad b = \frac{v}{f} \cdot x \quad , \quad c = \frac{u}{f} \cdot x.$$

d is the value of the ToF depth image on position (u, v). f denotes a virtual focal length

$$f = \frac{\frac{u_{max}}{2}}{tan(\frac{\alpha}{2})},$$

merging the camera's field of view (viewing angle α) and the image resolution in one number. 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 results for f for the horizontal case is 271 and 282 for the vertical case, thus value is set to $f = 280$.

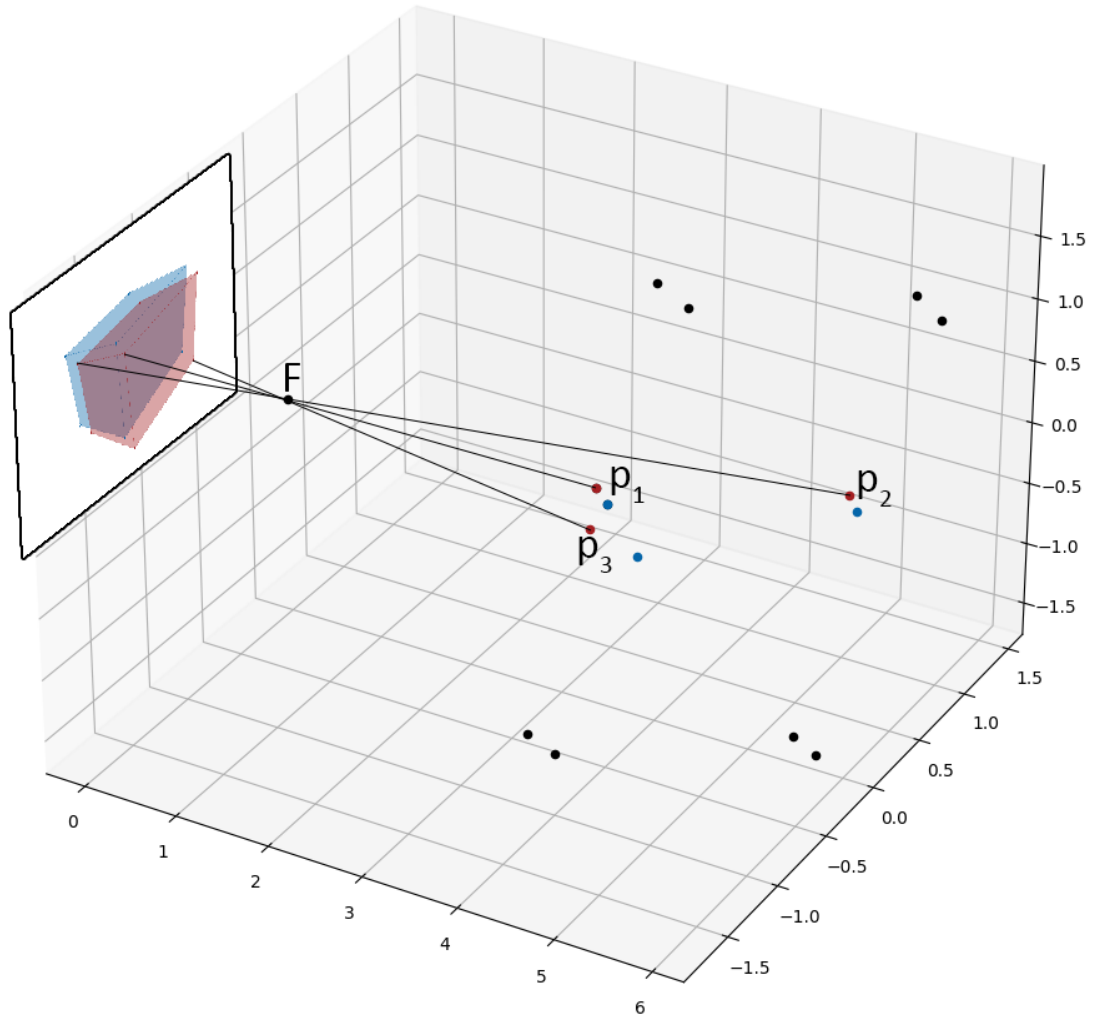


Figure 4.10: Second step of ToF motion estimation: Map features into 3D space by setting

4.3.3 Estimation of 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 the ETH Zurich [10], also containing the proof, the three points must be centered. 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 formula is n, 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:

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

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.

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

Multiplying the transposed centered point group Q_k with the point group Q_{k-1} generates the 3×3 covariance matrix

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

The point groups are packed in matrix form, each point being a column of the respective matrix. When using n points, the point group matrices are of dimension $n \times 3$, whose covariance matrix remains of dimension 3×3.
The singular value decomposition (SVD), explained in Section 3.2.3, 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 [23] from GitHub [24] can be utilized.

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

The two rotation matrices U and V allow calculating the rotation between the point clouds

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

Without the correction term $det(VU^T)$ in the intermediate matrix, the method could generate a reflection instead of a rotation. This is numerically sound, but does 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.
The wanted translation is computed by applying the rotation to the centroid vectors.

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

The motion between the point clouds of consecutive the ToF images k and $k-1$ is given by

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

with R being the rotation matrix and $\vec{t}$ being the translation. For point clouds with $n = 3$ the result is exact, for $n > 3$, E is the mean square error [10]. The estimated translation is only the translation between two frames. To estimate the velocity, the translation is divided by the sampling time.

4.3.4 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. [19] 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, an extension to the RANSAC algorihm is proposed in the following.
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. To each group of three feature pairs, the rotation and translation is calculated using the method described in Section 4.3.3 as shown in Figure 4.11.

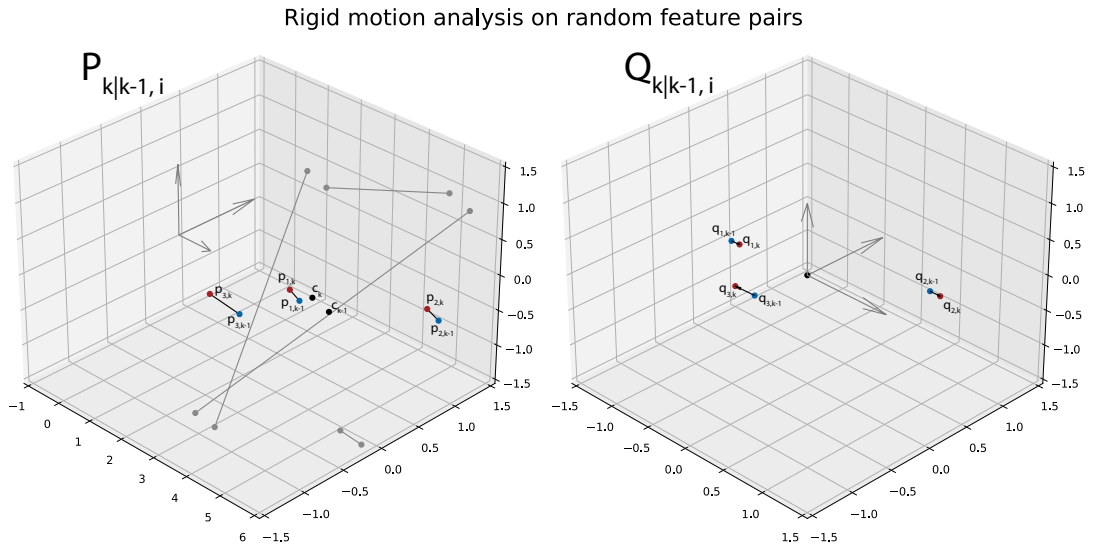


Figure 4.11: First step of the 3D RANSAC: Using the SVD method for estimating the rotation and translation from three randomly assigned points. This step is performed in parallel on multiple groups of three. Each group P_i generates the rotation R_i and translation $\vec{t}_i$.

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 transformed 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 lower than a threshold, the feature pair is marked as a proper motion, as shown in Figure 4.12. Over all these proper points, the average distance is calculated for each group of three feature pairs. The estimated matrix-vector-pair that generates the smallest average distance is likely suitable for further processing.

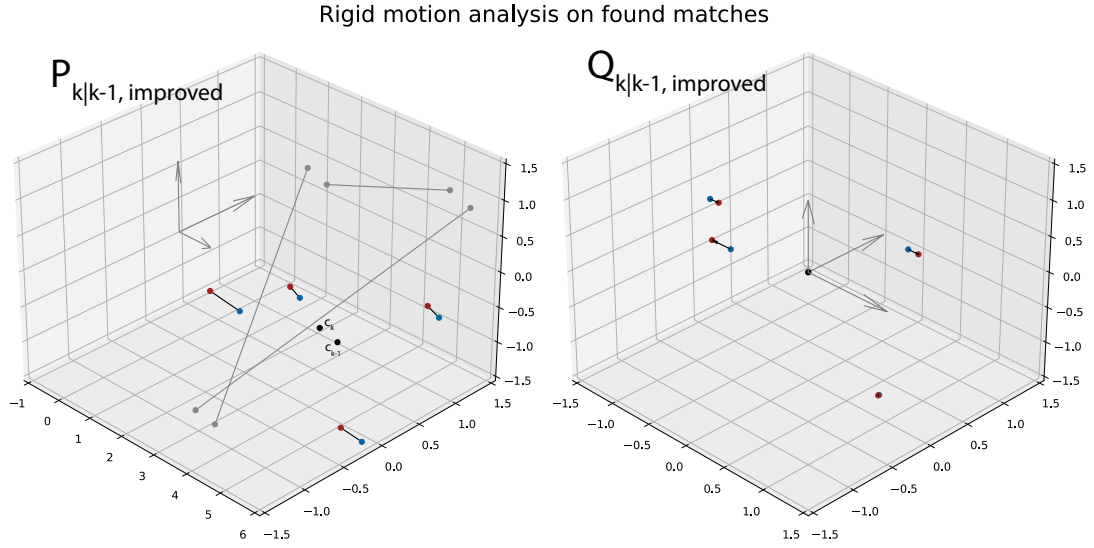


Figure 4.12: Second step of the 3D RANSAC: The calculated rotations and translations get evaluated against the whole dataset. In the example, the fourth matching point fulfills the criterion.

In a third step, applying the SVD method to all the proper feature pairs as larger point clouds generates the improved motion $R_{improved}$ and $\vec{t}_{improved}$.

Ignoring the SIFT metrics that lead to the brute force matching, all the features get matched again based on the position alone, using the rotation $R_{improved}$ and translation $\vec{t}_{improved}$ 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 marked for the last step.

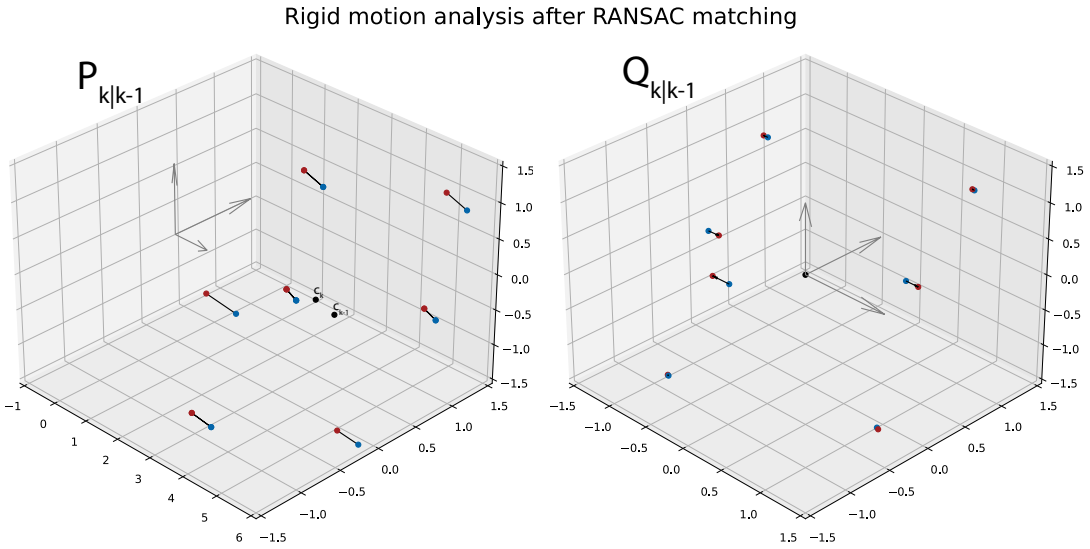


Figure 4.13: Third step of the 3D RANSAC: Applying the currently available rotation- and translation-information the the point clouds allows re-matching based alone on the position. The optimal rotation R_{opt} and translation $\vec{t}_{opt}$ result from these feature pairs.

Finally, the optimal rotation matrix R_{opt} and translation vector $\vec{t}_{opt}$ are estimated using the same SVD method, using all RANSAC matches.

4.4 Sensor Fusion

The following section describes the implementation of the accelerometer and gyroscope processing, marked blue in Figure 4.6, and the Kalman filter, marked in cyan in the same image, which fuses the data with the ToF camera motion data.

4.4.1 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 compensate for gravity or for detecting movement - especially because of positional information being the result of integrating 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 was calibrated in 24 orientations to compensate for angular errors in the IMU, on the PCB and of the calibration table. For each orientation, 100 raw measurements have been averaged to eliminate noise. The largest error has been 38 mG, that lies within the sensor's specification of ±40 mG [25]. 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 is in the sensor's specification of ±0.5% full scale[25].
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 is well in the specified ±3 degrees per second[25].
The IMU shows a hysteresis that is 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.

4.5 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 Kalman filter, explained in Section 3.5, fuses the motion information of the ToF camera with the measurements from the IMU. 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 is

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

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 overdots.

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

4.5.1 Prediction

The model matrix F describes the evolution of the system state $\vec{x}$ via

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

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.

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

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

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.
The second step of the prediction is the prediction of the covariance matrix of the errors P.

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

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.
While P_0 starts as identity matrix, N is the gaussian process noise, see Section 3.5 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.

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

4.5.2 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.

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

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.

$$R = \sigma \cdot I$$

The rotation quaternion $Q_{Rot} = (r_c, a_c\mathbf{i}, b_c\mathbf{j}, c_c\mathbf{k})$ represents the new observation $\vec{z}_k = r_c, a_c, b_c, c_c$. With the following equations, the system state vector and the covariance matrix of the errors get calculated

for the next iteration.

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

The quaternion multiplication inside the model matrix F adds up the subsequent rotation speed in each iteration to the cumulated rotation. The Kalman correction step for rotations is carried out for the data of the ToF camera and the data of the Gyroscope, one after another.

4.5.3 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 quaternion 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 3.2.2. If the orientation drifted, it would be different from the measured orientation Q_m derived 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 4.5.2.

4.5.4 Correction: Translation

Data from the accelerometer and the translational velocity from the ToF camera get sampled in a, b and c coordinates, 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 3.2.4. After processing all the rotation sensors, the system rotation in the system state vector $\vec{x}$ is used to transform 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.

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

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

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.

4.6 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.
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 3.4 describes the process of the lens calibration.

4.7 Video display

Vulkan is a graphics-oriented low-level API that allows rendering and computation on the GPU. Like OpenGL, Vulkan is maintained by Khronos and is open-source software. Vulkan is – next to DirectX, which is limited to Windows – the go-to standard for computer games because of the developer having superior control over the whole graphics pipeline, including CUDA-like computation. While requiring more preparation than OpenGL, Vulkan is the framework of choice because the author has prior experience. The window in which the scene is rendered is managed by GLFW, a helper library for OpenGL.
The implementation renders two rectangles consisting of four vertices – corner points - to the GLFW window. The Raspberry Pi video stream should cover the entire window, as it acts as the viewfinder of the system. Knowing what Vulkan expects after the shader stage, as explained in Section 3.3, the easiest way to achieve a full and undistorted coverage is to provide the clip coordinates (-1,-1), (-1,1), (1,1), and (1,-1). The given vertex shader transformation matrix is the identity matrix. This

technique will not disable the bilinear and anisotropic filtering, the image sampler of Vulkan provides. Additional to the Raspberry Pi video stream, a projected surface gets rendered in front. As the projected plane should react to the motion data from the Kalman filter, hardcoding the coordinates does not work. Section 3.3 contains an example of the vertices and the matrix transformations used to achieve that goal. The motion data of the Kalman filter gets fed in a 4x4 matrix to move and rotate the three defining vectors of the «view»-matrix – namely the eye, the center, and the up vector.

5 Testing and Results

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

5.1 Performance

Even though, no performance optimization was done, the frame rates got measured. Activating the FFMPEG capture for the Raspberry Pi Camera causes a performance hit, that not only affects other CUDA processes, but also Vulkan. The framerates have thus been measured with and without activated FFMPEG capture.

Measurement	ToF Camera	IMU	Vulkan	RPi Camera
With RPi Camera	7.85 fps	202.01 Hz	42.08 fps	42.50 fps
Without RPi Camera	11.19 fps	200.66 Hz	107.41 fps	-

Table 5.1: Average Framerates with and without activated FFMPEG Raspberry Pi Camera capture

The thread with the ToF Camera involves the whole SIFT feature extraction, RANSAC matching and the evaluation of the rigid motion. The IMU thread only polls an open I2C connection and sums up values.Because of this performance hit, the following measurements got recorded without activated FFMPEG RPi Camera. Instead of the RaspberryPi Camera, the ToF camera's debug image got drawn onto the viewfinder.

5.2 ToF Camera

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

5.2.1 Setup

As described in Section 4.1.2, 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 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 was set to 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.
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 3.2.4.

5.2.2 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 folding meter and a comparison with the sensor pixel value. Fitting a line to the measured data points gives the linear relation from the distance

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

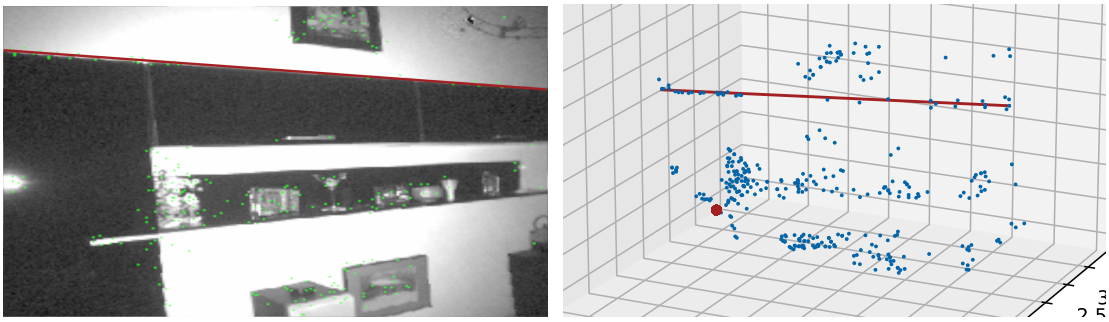
to the camera value val.

Figure 5.1: Sensor values measured against a folding meter. The red curve shows a linear fit of the data points.

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 causes the distance measurement to have an standard deviation in the a-axis of 7mm.
The formulae for the 3D scene reconstruction propagate the uncertainty to the b- and c-axes to about 4mm and 3mm respectively.

5.2.3 3D Scene Reconstruction

After calibrating the lens and correcting the radial nature of the distance measurement, as described in Section 4.3.1, the generated point cloud directly reconstructed the three-dimensional scene recorded by the ToF camera.
Straight lines in the real world appear straight in the point cloud, which was veryfied by analyzing a straight line in the chosen scene using SIFT features, as shown in Figure 5.2.



(a) Image (b) Cloud

Figure 5.2: 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 left image is a SIFT feature. The red dot in the point cloud is the position of the camera.

5.2.4 RANSAC feature matching

The RANSAC algorithm improves the quality of matched features over the flawed brute-force matcher. The data of the ToF camera is prone to noise, the RANSAC algorithm needs to cope with positional uncertainties. As discussed in Section 5.2.2, 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.
A threshold based on the sum of square differences gives the RANSAC algorithm the flexibility to create suitable matches. 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, however the number of matches is very small as seen in Figure 5.3 The error on the other axis is smaller as discussed in Section 5.2.2, this does not explain the poor matching performance at this threshold, as seen in Figure . Likely, some other source of noise diminishes the matching performance. The image noise on the black-and-white image causes 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.

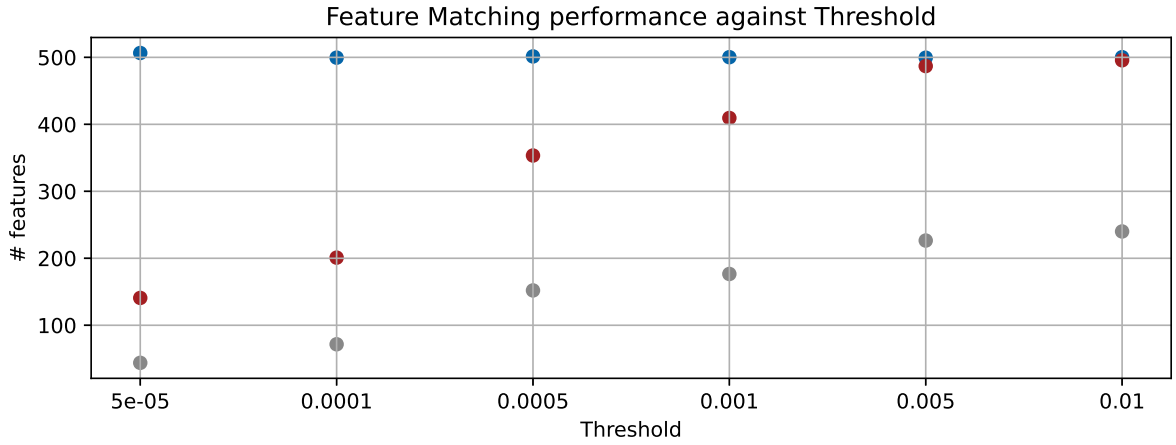


Figure 5.3: Plotting the feature matching performance against different thresholds shows the quality of the RANSAC feature matcher. In blue, the raw number of unmatched features in each test, in gray the brute force matches and in red the RANSAC matches.

Higher tresholds in Figure 5.3 show the expected performance of the brute-force matcher in grey. The

RANSAC feature detector vastly improves the matching performance, seemingly up to over 97% at the threshold of 0.005. At a threshold of 0.005, the sphere around the expected position has more than 14cm in diameter, leading to the undesired situation of the RANSAC matcher finding feature points. The RANSAC matcher might find the same match for different unsuitable feature points. The matches are not exclusive, combined with a 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 outliers. Figure 5.4 on the next page demonstrates the matching with the chosen threshold at the example of a rotation of the camera head. Two consecutive frames generated a point cloud each. The estimated rotation and translation move each data point of one cloud to a hypothetical position. The closest data point of the other cloud 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.

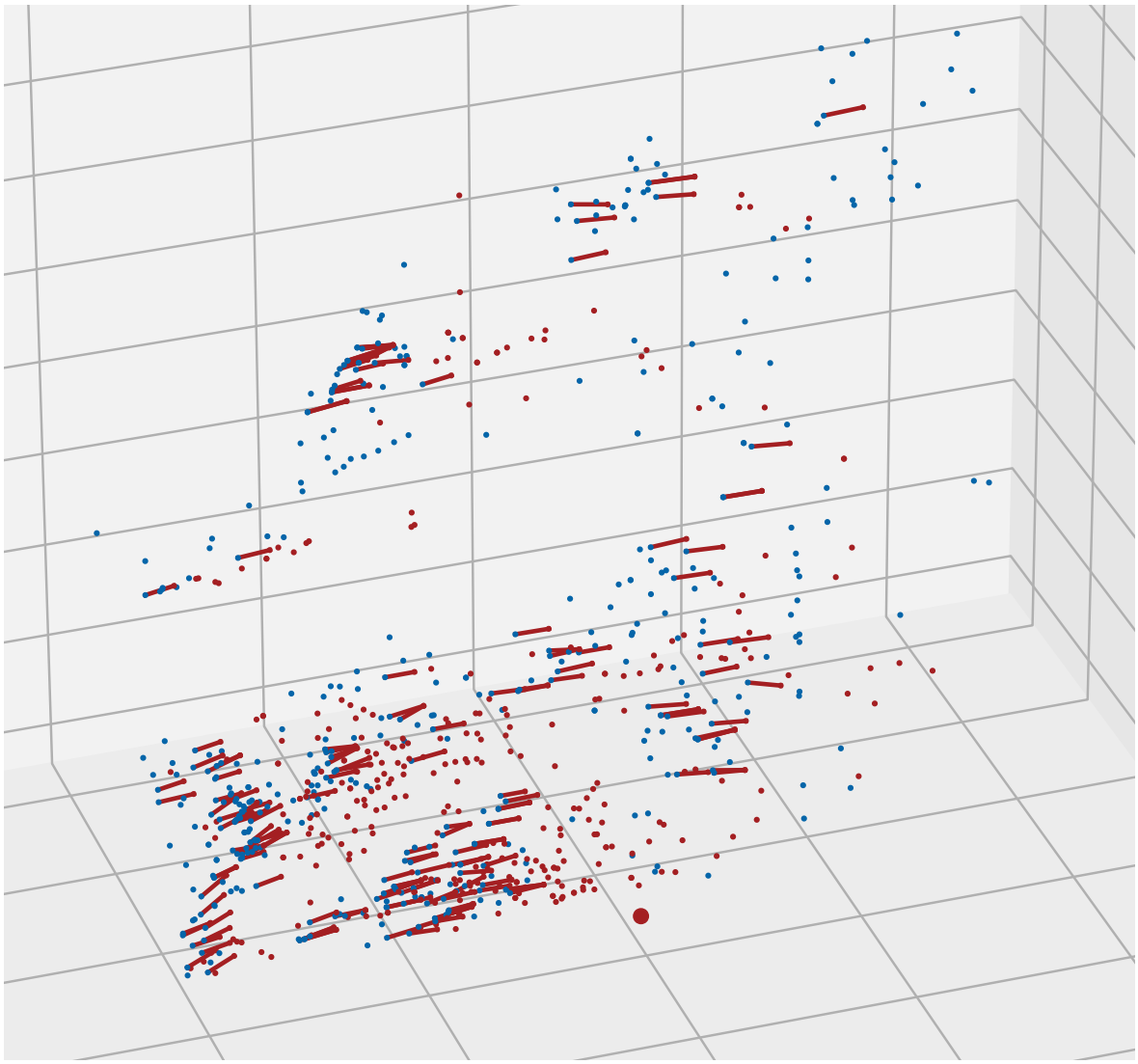


Figure 5.4: Two raw point clouds with the found matches connecting them. The camera is rotated in between the frames, which leads to the displacement of the point cloud. The connecting lines are roughly 10cm long, depending on the position. The large red dot in the foreground is the position of the camera.

Measurement	Rotation	Translation X	Translation Y
Total Features (avg)	435.9	400.0	488.4
Brute-Force Matches (avg)	117.9	104.2	112.6
RANSAC Matches (avg)	280.2	262.5	284.3

Table 5.2: Per-Frame average performance of the feature matching approaches

The claimed performance for the brute-force matcher is below 50% when applied to two identical images, according to the developers of the CudaSift library.[19] On noisy frames, even with motion between frames, the quality is expected to be lower.
Table 5.2 shows the performance of the RANSAC implementation in comparison to the brute-force matcher. On average an improvement of over 100% in the number of accepted matches is achieved by running the three-dimensional RANSAC algorithm.

5.2.5 Rotation from ToF camera

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 two sensory systems. Analyzing the imaginary parts is sufficient, as explained in Section 3.2.2. In this experiment, the camera head was rotated in each direction and directly compared to the gyroscope data. The camera was mounted on a standard camera tripod and rotated by hand, as the author has no access to a device allowing automated rotations. Figure 5.5 shows the results.

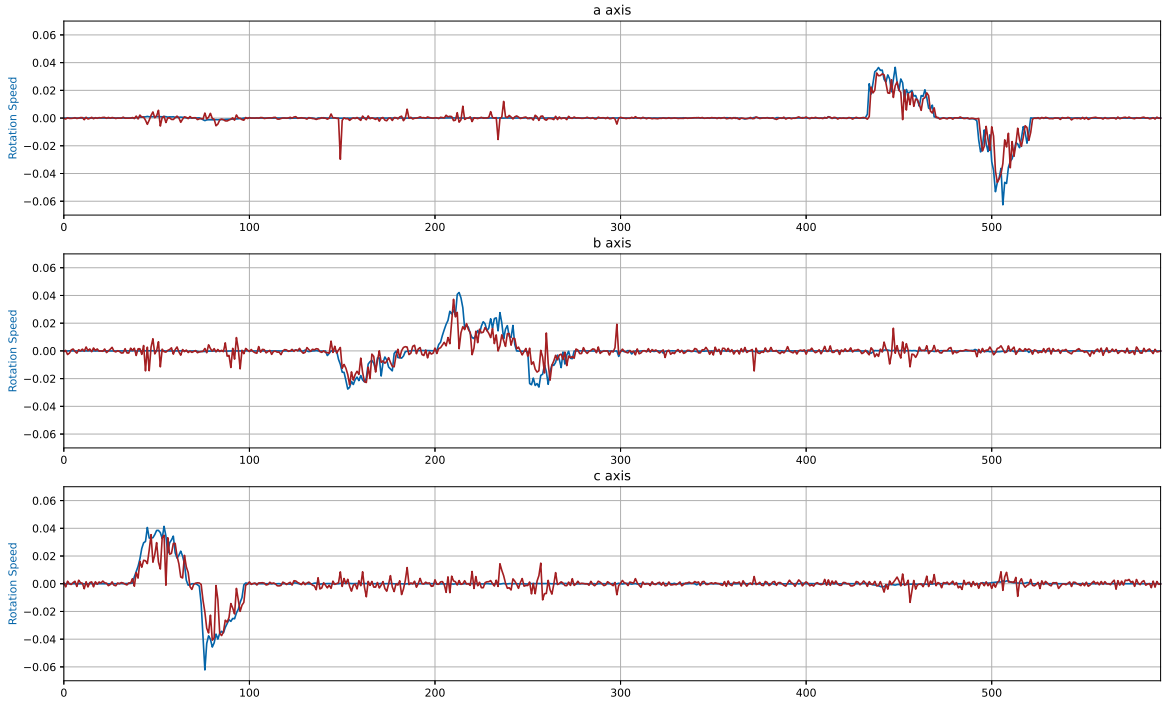


Figure 5.5: The individual axis of the ToF rotation quaternion in red plotted against the gyroscope quaternion in blue. The camera head got rotated in each direction after another, first sideways around the *c*-axis, then downwards and upwards around the *b*-axis and at last tilted around the *a*-axis. The plotted values resemble imaginary parts of a quaternion and are therefore without units.

The measurement on the motion estimation from the ToF camera shows significant noise, especially

on the rotation alongside the b- and c-axis, and does not entirely follow the gyroscope curve in blue. Nevertheless, measurement shows the proof of concept.

5.2.6 Translation from ToF camera

Like for the rotation measurement, the translation is measured against the IMU. In the case of translation, the acceleration data from the IMU is compared to the translation estimation of the ToF camera. The camera is moved on a toy rail in x- and y-directions. Motion along the z-axis is not measured, as its priniciple is the same as for the y-direction. 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 was corrected. As the camera did not rotate during these measurements, the values are equivalent to the abc-notation. The camera was slid from its origin to the front, respectively to the side, and to its origin twice on each run. The first motion was kept fast, the second motion was kept slow. The length of the track is about 0.5 m in both directions.
As visible in the Figures 5.6 and 5.7 on the next page, the velocity data from the ToF camera shows significant noise, but the motion is detected correctly.

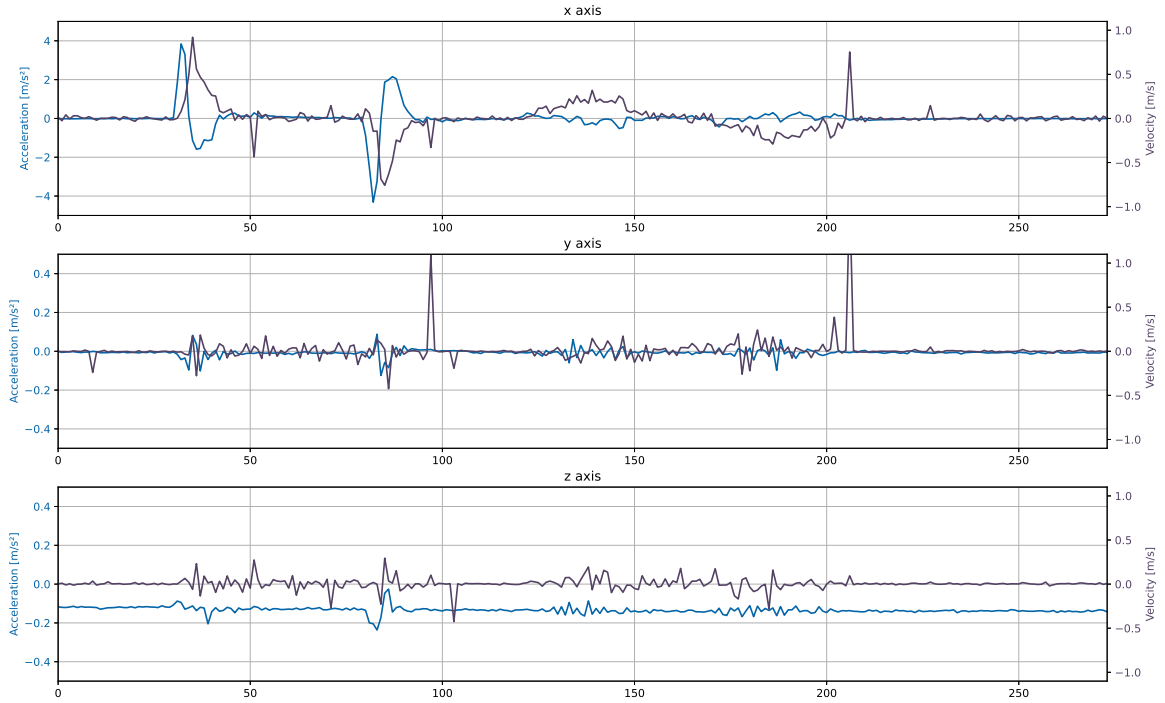


Figure 5.6: Raw measurement of the translation alongside the x-axis. Before and around frame 50, the camera is slid fast forward, kept in pause just to slide back before reaching frame number 100. After frame 100 and before frame 220, the same motion is repeated but slower. Blue is the acceleration from the IMU and purple the velocity from the TOF sensor.

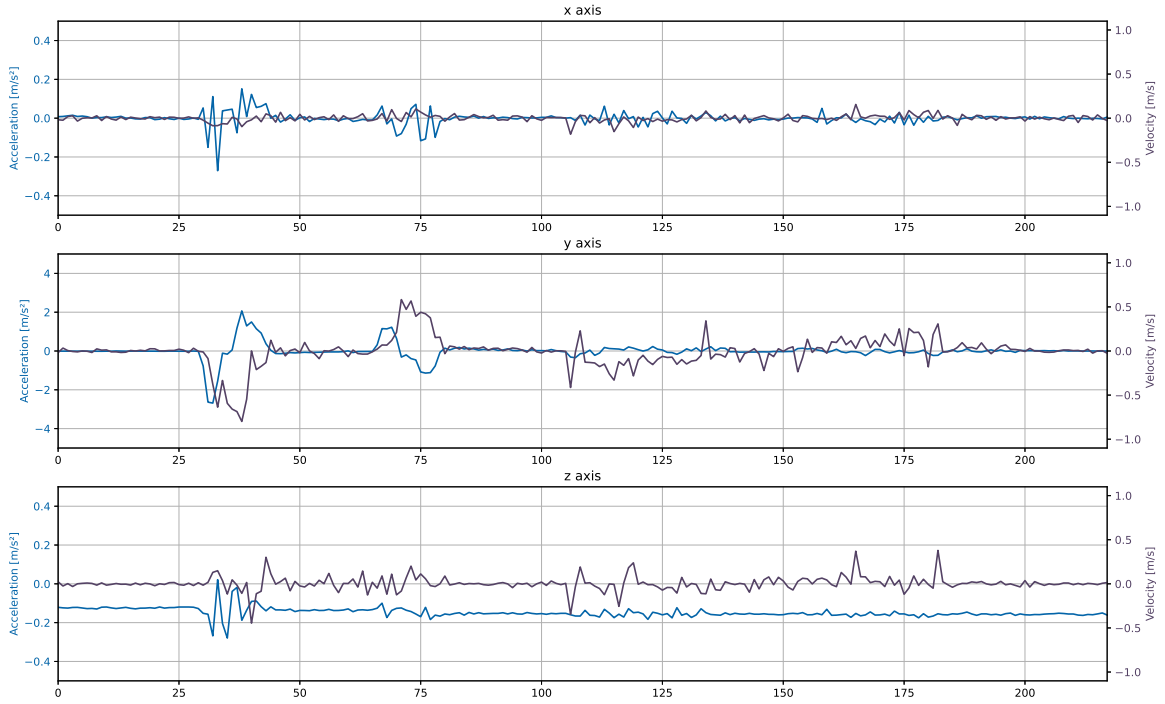


Figure 5.7: Raw measurement of the translation alongside the y-axis. Between the frames 25 and 50, the camera is slid to the right, kept there and slid back to the origin around frame 75. The motion is repeated slower after frame 100 and before 200. Blue is the acceleration and purple the velocity. Note how noise increased on the other axis, while the camera was in motion.

Raw integration of the ToF velocity values gives insight into the measurement's quality. As visible in Figure 5.8, the data reconstructs the linear motion in both directions, even though it gets jagged by noise as expected. Noteable outliers, like at the end of the second motion on the x-axis, induce larger errors.

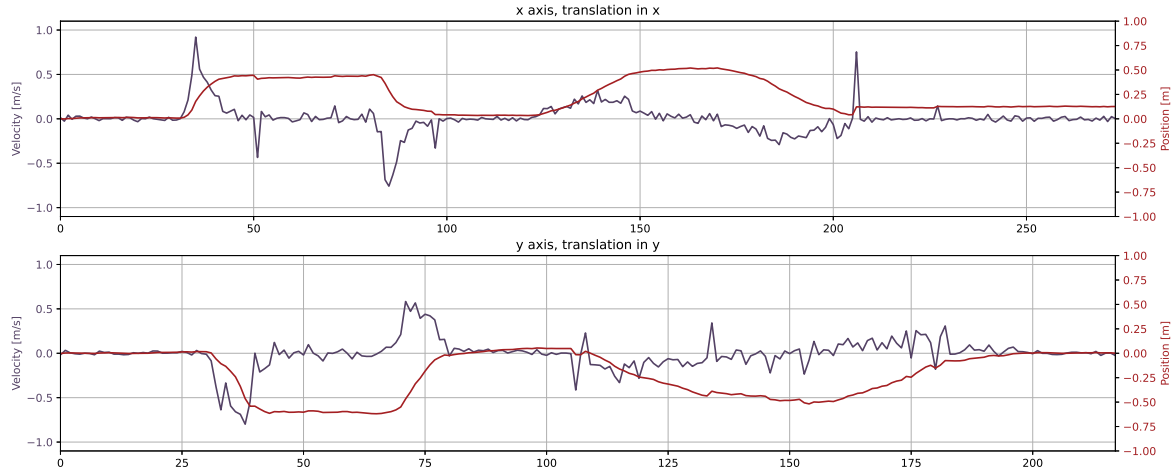


Figure 5.8: Raw integration of the relevant axis in both translations of the ToF velocity data. Purple is the velocity from the ToF camera and red its integration.

5.3 Inertial Measurement Unit

The 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 5.2.5, the accelerometer data requires more profound analysis.

5.3.1 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 showed that a hysteresis did prevent 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 causes an integrated velocity and devastatingly affects the further integrated position, as visible in Figure 5.9 on the next page. The aforementioned offset is visible 5.6 and 5.7 on the z-axis, thanks to the narrowed scale.
The added offset error worsens the drift compared to the simulation in Section 3.5 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 row of Figure 5.9 shows the critical drift, if the gravitational pull is not compensated entirely. The data for the third plot was recorded during the measurement of the motion alongside the x-axis.

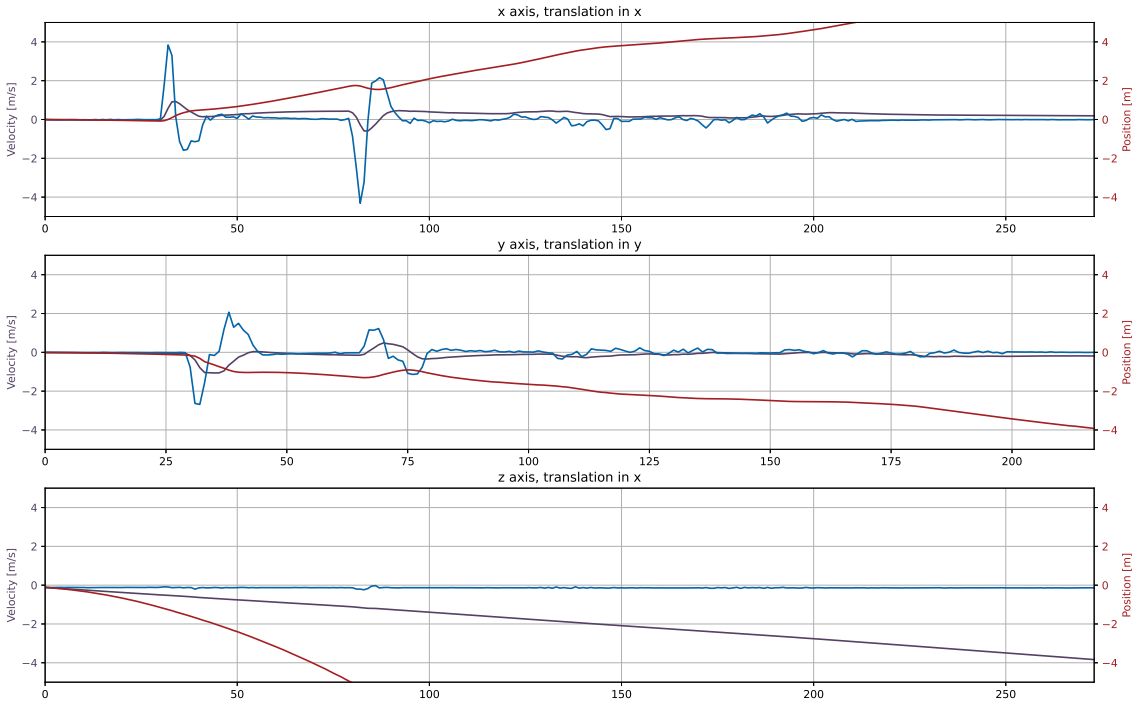


Figure 5.9: Integrations of the accelerometer data alongside relevant axis in both translations. Blue is the acceleration, purple the velocity (single integration) and red the position (double integration).

5.4 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. The only coupling between the rotation and the translation is the transformation from ABC-coordinates to the XYZ-space.

5.4.1 Rotation

As mentioned in Section 5.2.5, the gyroscope and the ToF camera generate a rotation speed of comparable magnitude and shape. The gyroscope data is less distorted by noise, therefore the motion model can rely more heavily on this data source.
The Kalman filter smoothes data for the rotational speed through its model function and calculates the rotation from there. Additional to the gyroscope and the ToF camera, the drift compensation with the accelerometer stabilizes the values when the camera head is stationary.
In the comparison of the output of the Kalman filter against the sensory data, the strong weight towards the gyroscope becomes apparent, as shown in Figure 5.10.
The shown result in Figure 5.11 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, thus, the 3D object moving in the image was expected. 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.

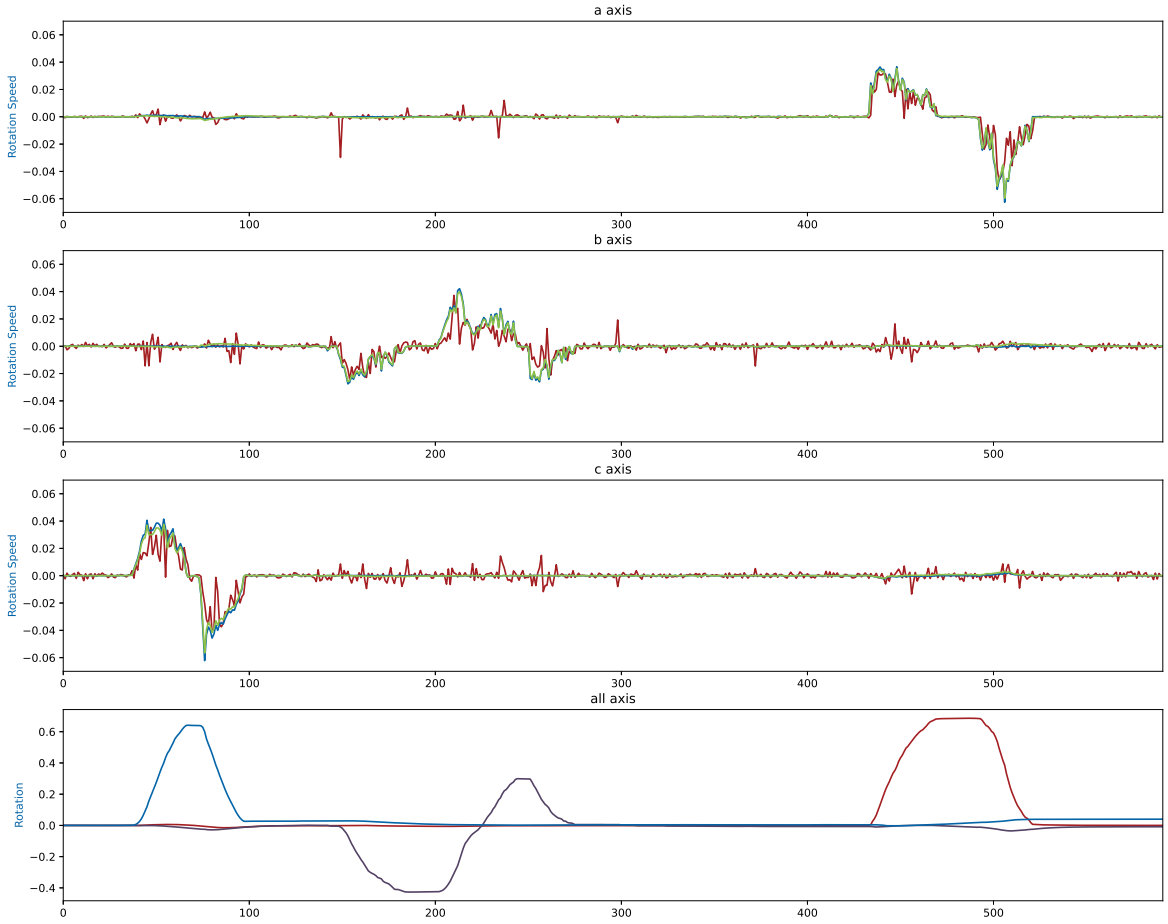


Figure 5.10: The output of the Kalman filter is plotted in green, against the gyroscope data in blue and the ToF camera data in red in first three rows. The fourth row shows the calculated rotation. The a-axis in red, the b-axis in purple and the c-axis in blue.

(a) Initial situation (b) After rotation

Figure 5.11: 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.

5.4.2 Translation

Figure 5.12 shows the output of the Kalman filter in green against the data from the accelerometer and the ToF camera. Without other input to compare against, the Kalman filter follows the accelerometer directly. 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.
The accelerometer impacts the filter output, 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.

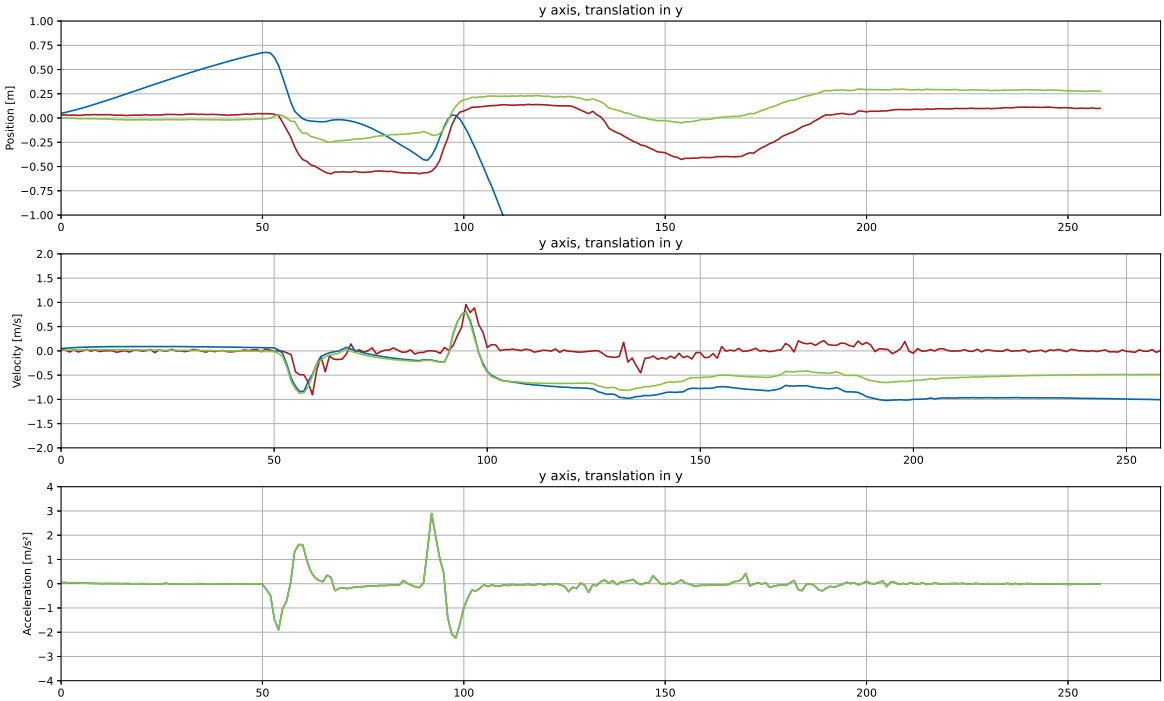


Figure 5.12: Motion in y direction. Blue lines are data from the accelerometer and its integrations, red are the velocity from the ToF camera and its integration and in green are the different Kalman outputs.

6 Conclusion

This work provides a proof of concept for extraction of rotation and translation speed from a ToF camera on a frame-by-frame basis works. The reconstruction of a three-dimensional scene from the depth image and SIFT feature points and with up to 512 parallel singular value transforms on brute-forced matches allows finding a good rotation and translation transformation. Enhancing the brute-force matcher with a three-dimensional RANSAC algorithm more than doubles the matching performance. Applying the singular value transform to each improved match results in an optimal rigid motion transform. As the system relies on finding SIFT features on its grayscale image – the algorithm will possibly not find enough features in an empty room.
The noise of the ToF camera was a significant problem during this thesis. The noise of the grayscale image causes the extracted SIFT features to jitter, even when the camera is stationary. The noise of the depth information increases the error on the three-dimensional scene reconstruction that forces the RANSAC algorithm to run with loose constraints. These loose constraints allow the RANSAC algorithm to falsely match features only because they are close enough, which induces an error on the rotation and translation estimation. This error again causes the motion data to be noisy.
The low framerate of the ToF camera limits the iteration rate of the Kalman filter. The frame-by-frame translation and rotation information is an average of the motion between the frames, not the velocity at the current frame's time. The low sampling rate requires a downsampling of the IMU data, which induces problems with the gravity compensation.

6.1 Possible improvement

A system that estimates the position directly based on visual key points is required to avoid drift on the rigid motion. A visual key point might be a specific cloud of SIFT features or an object classified by an machine learning algorithm.
The system would need to detect new key points, store them in a list, store the position and maybe even improve its position information when new data is available. Estimating the camera's position becomes possible from the external objects' position.
The position estimation from an image is valid for the moment where the frame got recorded – unlike the velocity, which is an average between two images. Therefore, other parts of a Kalman filter could run at higher speeds.
A different sensor fusion approach, which allows the IMU to run at its sampling rate without requiring a cumulative sum, would improve the position estimation. The orientation of gravity could be estimated at any point and could correct every single accelerometer measurement.

6.2 Outlook

Augmented Reality is a vast field with various problems that need to be solved. When motion tracking works reliably, it enables multiple applications. Increasing the frame rate and accuracy with a better ToF sensor, as currently investigated by the Institute of Signal Processing and Wireless Communications of ZHAW, might enable the system for further topics. Motion tracking by a ToF camera might be implemented in an autonomous driving platform, with a custom-tailored Kalman filter that takes the wheel motion and steering action into account.
Developing specialized hardware would be a larger project, as this involves many technologies, like optics, eye tracking, translucent displays and a portable processing platform.

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

[5] 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.

[6] 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: 28.11.2021.

[7] Ar.js - augmented reality on the web. https://ar-js-org.github.io/AR.js/. Accessed: 21.12.2021.

[8] argon.js - a javascript framework for adding augmented reality content to web applications. https://www.argonjs.io/. Accessed: 21.12.2021.

[9] 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.

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

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

[12] Linmath library inside the cuda-samples repository. https://github.com/NVIDIA/cuda-samples/blob/master/Samples/vulkanImageCUDA/linmath.h. Accessed: 21.12.2021.

[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] 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.

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

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

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

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

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

[23] 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).

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

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

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Meaning
ZHAW	Zurich University of Applied Sciences
InES	Institute of Embedded Systems
HPMM	High Performance Multimedia - a research group ant ZHAW-InES
ToF	time-of-flight
LiDaR	Light-Detection and Ranging
MEMS	Microelectromechanical systems
IMU	Inertial Measurement Unit, a sensor for acceleration, rotation and magnetic fields
SVD	Singular Value Decomposition
RANSAC	Random Sample Consensus
SSD	Sum of Square Differences
GLFW	Graphics Library Framework
GLM	Graphics Library Mathematics
OpenGL	Open Graphics Library - Predecessor of Vulkan
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 6.1: List of Abbreviations

7 Appendix

7.1 Structure of the Git-Repository

7.1.1 Documentation

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

7.1.2 Source

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

7.1.3 Literature

Contains 3rd party documents for technologies used in this thesis

7.2 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
- NVIDIA JetPack 4.4: For flashing the TX2 and profiling the application