Zurich University
of Applied Sciences

School of
Engineering
InES Institute of
Embedded Systems

# Augmented Reality Platform using Sensor Fusion

Zurich University of Applied Sciences

Institute of Embedded Systems

| | |
|---|---|
| Authors | Marcel Wegmann |

| | |
|---|---|
| Version | 0.1 |
| Last changes | November 18, 2021 |

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

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

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

Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering
InES Institute of
Embedded Systems

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

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

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

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

Ort, Datum: Unterschrift:

…… ……..

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

# Abstract

This is gonna be really abstract...

# Preface

Thanks! I guess....?

# Contents

# 1 Introduction

This introduction is very doge.

## 1.1 Scope

The scope is much wow.

## 1.2 Initial Situation

The initial situation is kinda sus.

## 1.3 Goals

There are so many goals!

## 1.4 Target audience

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

# 2 Motivation

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

# 3 Concept

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

# 4 Fundamentals

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

## 4.1 3D Cameras

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

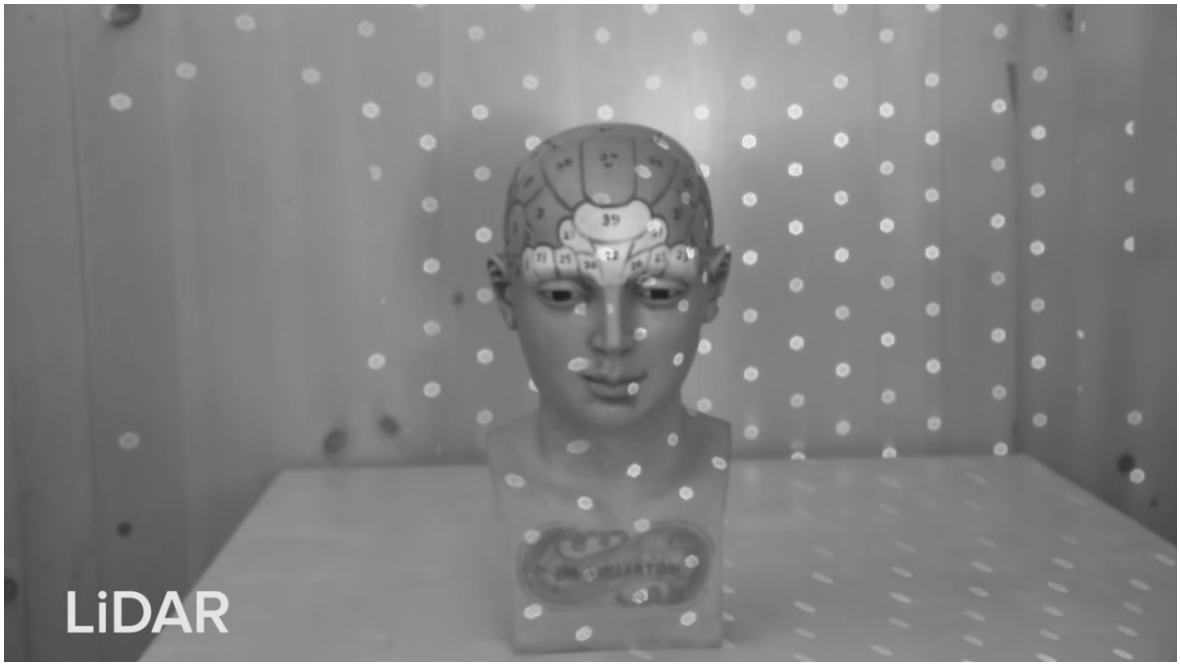


Figure 4.1: Projected dots from the LiDaR scanner of an Apple iPad Pro 2020, made visible with an infrared camera. Image source: iFixit.com

Another method for 3D mapping is, using one wide area infrared flash and measuring the time of flight on each individual sensor pixel. This approach in contrast is often referred to be a ToF camera. ToF cameras are used by different manufacturers of Android powered smart phones.
While the measurement principle in both technologies is the same, the LiDaR scanner generates a point cloud, while the ToF camera outputs a depth map image. With mathematical transformations, both outputs are equivalent. Another difference is that a ToF camera can double as a grayscale infrared camera, providing an image by itself.
The sensor used for this thesis follows the principle of a ToF camera. The measured radial distance from the sensor for each pixel allows the three dimensional reconstruction of the scene.

### 4.1.1 Radial information from ToF cameras

The used ToF camera provides the measured distance from the sensor to the filmed object for each pixel, generating a depht map. Because of the radial measurement, flat surfaces appear warped. The distortion follows a curve which can be corrected by knowing the cosine of the angle $\alpha$ for each pixel.

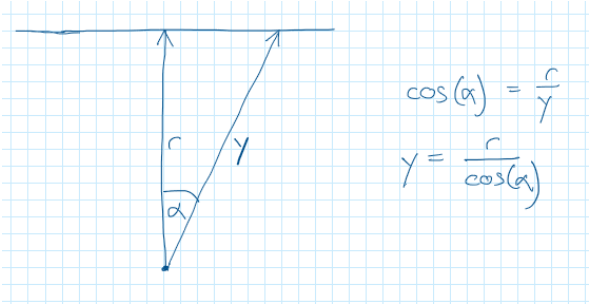


Figure 4.2: Geometrics of the radial measurement and its correction.

With the angle of the individual pixels being unknown, a map of cosine $\alpha$ is generated by taking an image of a flat surface.

## 4.2 Mathematics for Rotation and Translation

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

### 4.2.1 Euler Rotations and Linear Algebra

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

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

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

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

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

of the whole object around the origin $(0,0,0)$.

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

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

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

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

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

The dependency on the order of the rotations poses a problem visualized in image 4.3: The values returned by a gyroscope would need to be applied all at once and not one after another. Replacing rotation matrices by quaternions - described in section 4.2.3 - solves this problem.

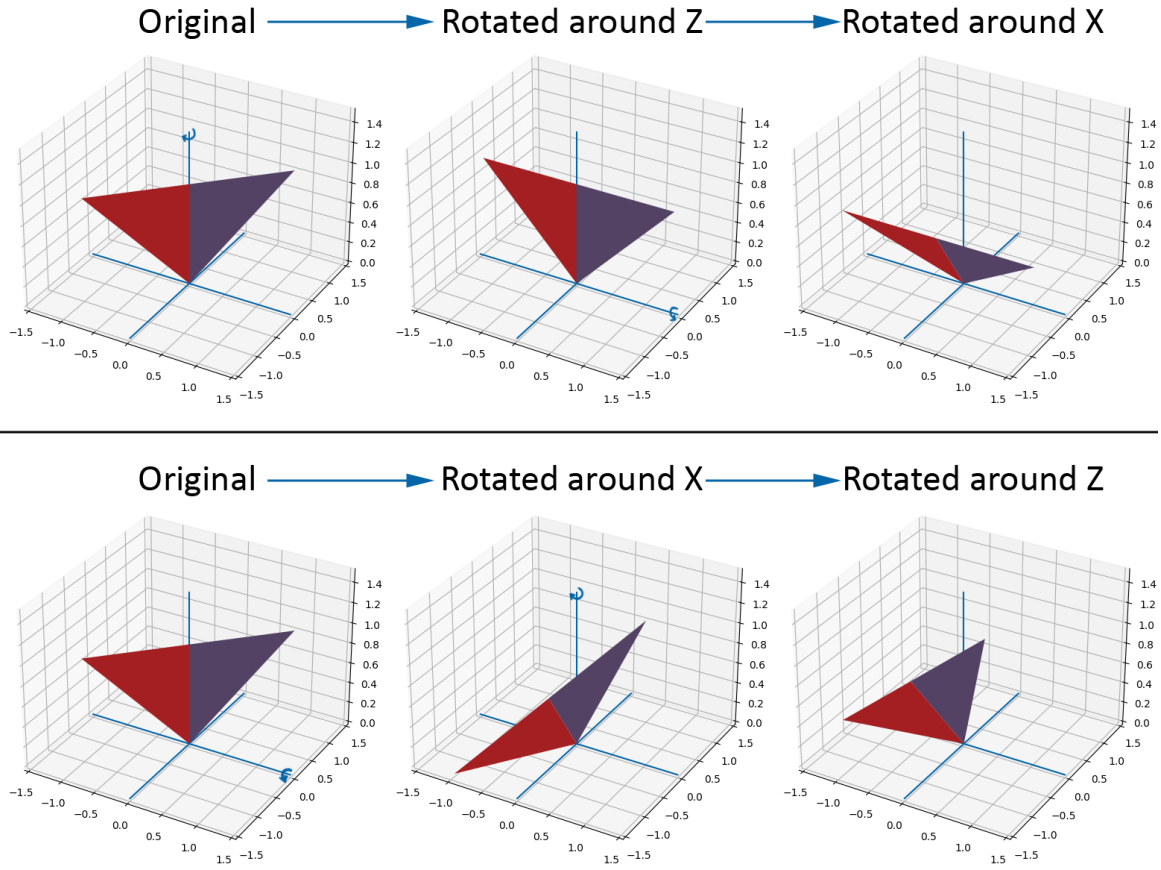


Figure 4.3: Euler rotations are dependent on the order of the individual rotations. Rotating around X and then Z results in a different outcome, than first rotating around Z and then X.

### 4.2.2 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 "view coordinates" via "clip coordinates" towards "normalized device coordinates" to the "pixel coordinates".
A 3D object is a cloud of vertex coordinates described by a list three-dimensional vectors $\overrightarrow{v_v} = (x_v, y_v, z_v)$ (the subscript $v$ denotes the "view coordinates"). These coordinates usually do not contain data regarding the whole object's scale, position, and rotation in 3D space - this information is added with matrices in the shader step.
Vulkan expects the output of the shader step to be in clip coordinates. Clip coordinates are four-dimensional vectors $\overrightarrow{v_c} = (x_c, y_c, z_c, w_c)$ (the subscript $c$ denotes the "clip coordinates") and the result of a matrix multiplication operation.

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

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

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 and is a standard 4x4 matrix as explained in section 4.2. The view matrix ($A_{View}$) is also is a rotation and translation matrix, but describes the position and direction of the viewport camera (or eye). Rotating the camera rotates the rendered virtual space, which indirectly moves and turns the models in the viewport. The linmath-library offers the "4x4_look_at" function that calculates the view matrix based on camera position, a viewing angle, and the information regarding the "upwards" direction.
The projection matrix ($A_{Projection}$) is not a rotation and translation matrix besides the model and the view matrix. As its name suggests, the projection matrix reduces the vertex' 3d coordinates to the viewport plane by projecting them onto a virtual screen. By taking a field of view angle, the projection matrix allows camera distortion. The linmath-library offers the "4x4_perspective" function that calculates the projection matrix based on a given field of view angle Within the vertex shader, these three matrices are chained to perform the desired transformation.

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

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

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

The transformation to the pixel coordinates are also done by Vulkan without requiring any action by the developer. The only noticable detail is the alignment of the normalized device coordinates. 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 $(1/1)$.

### 4.2.3 Rotation with Quaternions

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

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

The neutral element of the quaternion multiplication is:

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

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

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

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

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

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

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

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

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

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

## 4.3 Motion detection by camera and depth information

Turning or moving the camera head influences the image of an object differently, based on its distance to the camera. The position of a single point within the image depends on the coordinates of the point in the real space, as shown in image 4.4. The optical axis matches with the $x$ axis while the sensor plane lies in the $YZ$ plane.

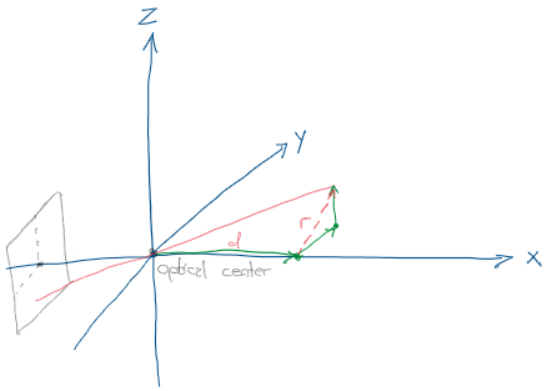


Figure 4.4: Projection of a point in space to the image sensor.

### 4.3.1 Singular Value Decomposition (SVD)

The singular value decomposition is a linear algebra method that allows finding the optimal rotation matrix between two matching point clouds. Adding additional algebraic steps makes it possible to find the optimal translation vector in addition.[11] For demonstration purposes, the recipe is carried out in a 3D example in section 5.4.4 in the Implementation chapter.
The SVD decomposes any given matrix $M$ of the dimension $n \times m$ into three matrices $U$ of dimension $n \times n$, $\Sigma$ of dimension $n \times m$, and $V$ of dimension $m \times m$.[12] If $M$ is real, $U$ and $V$ are guaranteed to be orthogonal, while $\Sigma$ is a diagonal matrix. The matrices fulfill the following equation:

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

The individual matrices generated by the SVD act als two rotations and one distortion. As visible in image 4.5, the unit circle $\vec{x}$ first gets rotated by $V^T$ that the distortion is in the directions of the coordinate system. After applying the distortion $\Sigma$ to the rotated circle $\vec{y}_1$, the intermediate ellipse $\vec{y}_2$ gets rotated and mirrored to match $\vec{y}$. The following recipe[12] shows how to find the matrices of the singular value decomposition. The matrix $M$ that is used in visualization 4.5 gets decomposed.

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

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

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

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

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

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

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

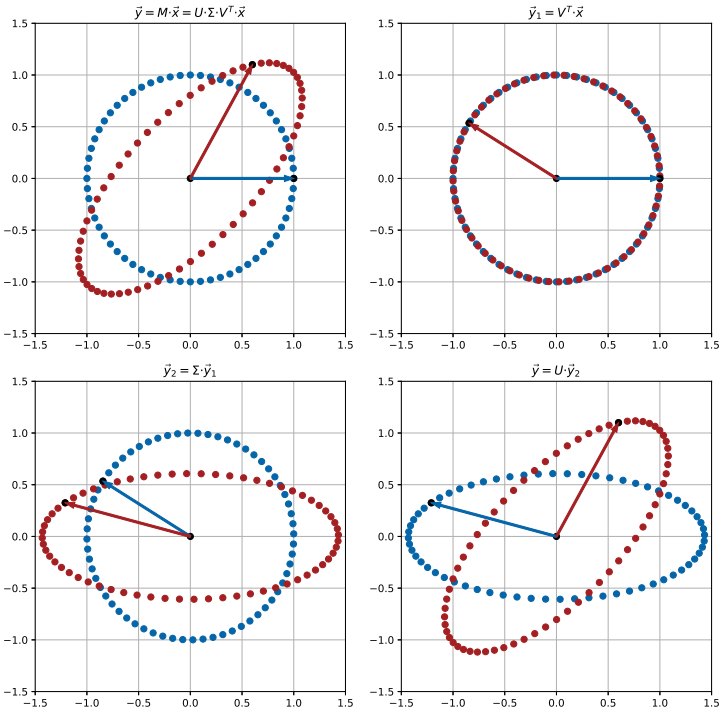


Figure 4.5: Singular Value Decomposition in 2D shown in the individual stages. Blue is before and red is after the applied transformation.

## 4.4 Camera Calibration

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

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

Tangential distortion is modeled as [13]:

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

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

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

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

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

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

Figure 4.6: Before correction

Figure 4.7: After correction

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

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

TODO

Figure 4.9: Storing the final calibration values into an array keeps the implementation in CUDA simple.

# 5 Implementation

This chapter describes the implementation and the connections of the individual components of the augmented reality system. There are two main tasks, the system needs to perform: A three dimensional estimation of the surrounding and the precise measurement of motion of the camera head.

## 5.1 Hardware

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

### 5.1.1 Evaluation of ToF Camera

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

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

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

**Other image sensors**
During the internet research, the Texas Instruments OPT8241 has shown up. TI declared the chip

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

### 5.1.2 Camera Head

The camera head contains the PiEye Nimbus ToF camera, mounted on a Raspberry Pi 4B, a Bosch BMI160 IMU, attached to a USB to I²C bridge, and a standard Raspberry Pi camera v2.1 which is connected to the processing system via an FPDLink module.
All is held together by a plywood structure glued onto a tripod-baseplate, shown in figure 5.3.


Figure 5.1: Frontside


Figure 5.2: Backside

Figure 5.3: The camera head consisting of the ToF camera, the IMU and the Raspberry Pi camera

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

### 5.1.3 Processing System

An Nvidia Jetson Xavier AGX in the 8GB version carries out the processing, rendering, and data acquisition. An Anyvision baseboard - shown in image 5.4 - carries the Nvidia Jetson module and allows the direct attachment of the used data cables.


Figure 5.4: The Nvidia Jetson Xavier AGX on the Anyvision Baseboard.

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.

## 5.2 Video inputs

The system utilizes two cameras, a Raspberry Pi Camera v2 and a PiEye ToF camera that doubles as an infrared camera. The Sony IMX 219 based Raspberry Pi Camera v2 features a color image with an 8 megapixel resolution for single images, 1080p with 30 frames per second, 720p with 60 frames per second or 480p with 90 frames per second.[14] The Raspberry Pi Camera v2 is a Mipi CSI2 attached camera module whichs cable length got extended by an FPDLink serializer and deserializer.
The PiEye ToF camera is based on the Infineon REAL3 IRS1125A sensor which has a resolution of 352 x 288 pixels[15]. The ToF Camera is paired with infrared LED flashes to measure the time of flight of the light emitted and has an intended measurement range from 10 centimeters up to five meters. The entire ToF software stack runs on a Raspberry Pi that sends its video data via Ethernet to the augmented reality system.

### 5.2.1 ToF Camera calibration

On the ToF Camera, two parts need to be calibrated: The optics, as described in section 4.4, and the distance measurement. The ToF camera has a barrel type distortion that is corrected by the camera calibration algorithm implemented in OpenCV[13] and shown in image 4.8. As the process of lens correction cuts off parts of the image, the image size gets lowered to 265 x 205 pixels.
For the distance measurement, first the radial distances need to be flattened as described in section 4.1.1. As the angle $\alpha$ is not known for each pixel, a reference measurement provides the necessary information. To reduce noise, 19 images of the same flat wall has been taken, smoothed by a gaussian and averaged onto one reference image $I_{Ref}$. The wireframe image in figure 5.5 shows the curvature of the reference image. The rounding at the edges is an arfifact of the gaussian smoothing.

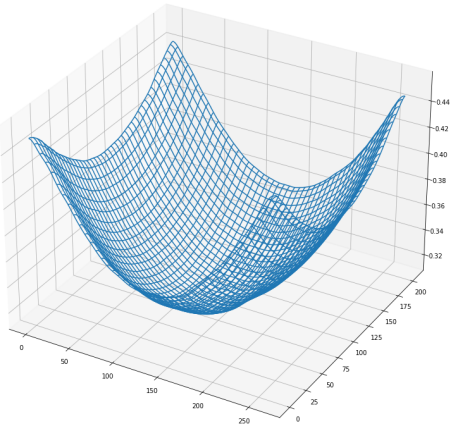


Figure 5.5: Wireframe rendering of the reference image $I_{Ref}$ provided by the ToF camera

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

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

Pixel by pixel multiplication of any other image $I_{Any}$ with $I_{cos}$ will correct the influence of the radial measurement as shown in image 5.6.

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

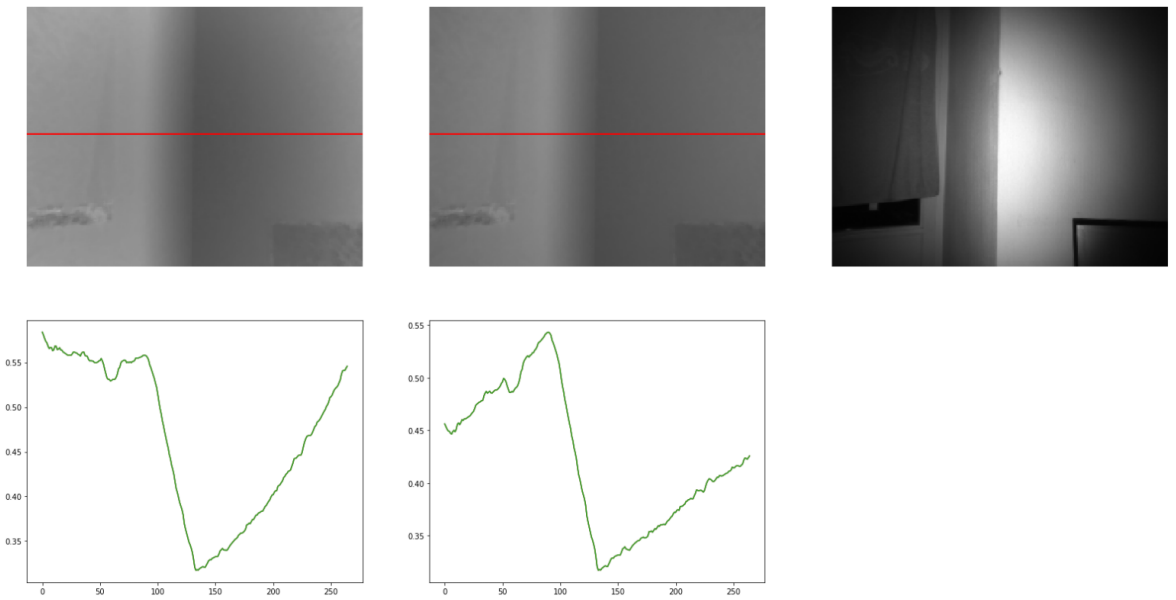


Figure 5.6: Left, the uncorrected ToF image $I_{Any}$, in the middle the corrected image $I_{Corr}$ and on the right, the infrared grayscale image of the scene. To make the effect more apparent, the brightness accross the red lines have been plotted.

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

### 5.2.2 Raspberry Pi Camera calibration

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

## 5.3 Gyroscope and Accelerometer Calibration

The used inertial movement unit (IMU) is a Bosch BMI160, that is sold soldered on a PCB by DFRobot.
As the gyroscope and accelerometer output incremental movement, the values need to be integrated

over time. Accurate data is crucial for finding the direction of gravity or detecting movement - especially because of positional information being the result of derivating the acceleration twice.
The range of the acclerometer is variable and has been set to ±8 G, it has an output resolution of 16 bit and an output data rate of 200 Hz. The accelerometer has been calibrated in 24 orientations to even out angle errors inside the IMU, on the PCB and of the calibration table, as shown im image 5.7. For each orientation, 100 raw measurements have been averaged to even out noise.


Figure 5.7: The 24 orientations used for calibration - four directions for each of the six sides.

The largest error has been 38 mG, that lies within the sensor's specification of ±40 mG[16]. In addition, the gain for the accelerometer has been corrected based on gravity. A maximum error of 1.8% has been measured and corrected, which also lies in the sensor's specification of ±0.5% full scale[16].
For the gyroscope, the range is set to ±2000 degrees per second with an output data rate of 200 Hz. The gyroscope has been calibrated only for zero offset, whose maximum error was 0.2 degrees per second which lies well in the specified ±3 degrees per second[16]. A gain correction was not made, because of not having the required equipment to do so.

## 5.4 Position and Orientation Estimation

The position estimation of the camera head is crucial for the augmented reality platform. The following section describes the implementation of the different systems involved in sensing motion.
While the measurement using the IMU is standard implementation via I²C communication, the implementation of the ToF-Camera based measurement is more complex and got split into multiple subsections.

### 5.4.1 Spatial coordinates and device coordinates

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

$Q_{rot}$.

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

Section 4.2.3 describes how to carry out the mathematics of this quaternion multiplication.

### 5.4.2 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 IMU generates measurements regarding acceleration in the directions $a$, $b$, and $c$, and measurements regarding rotation speed around these axes.
The IMU shows a hysteresis that lies within the specification of the datasheet; therefore, a simple offset correction does not yield the best results. A moving average – that gets updated whenever no motion of the camera head is detected – helps deal with the hysteresis by subtraction from the measurement value.

### 5.4.3 ToF Camera: SIFT feature extraction

The PiEye Nimbus ToF camera generates three different image channels for each picture taken: The depth map, the confidence, and the greyscale infrared image. By correcting the lens distortion as described in chapter 4.4, straight lines in the real world get projected as straight lines on the images. By further correcting the characteristic of the ToF camera to measure radial distances, as described in section 4.1.1, flat surfaces in the real world are also flat on the depth map. The implementations of both corrections are explained in section 5.2.1.
The points of the point clouds need to be matched to estimate rotation and translation between two consecutive ToF images. The CudaSift library [17] [18] extracts SIFT [19] features on each greyscale infrared image the ToF camera sends $k$, which are then brute-force matched with the features of the prior image $k-1$, as visualized in image 5.8. SIFT features were chosen because of the author's prior experience, and it is a well-known feature extraction algorithm whose patent expired. [20]

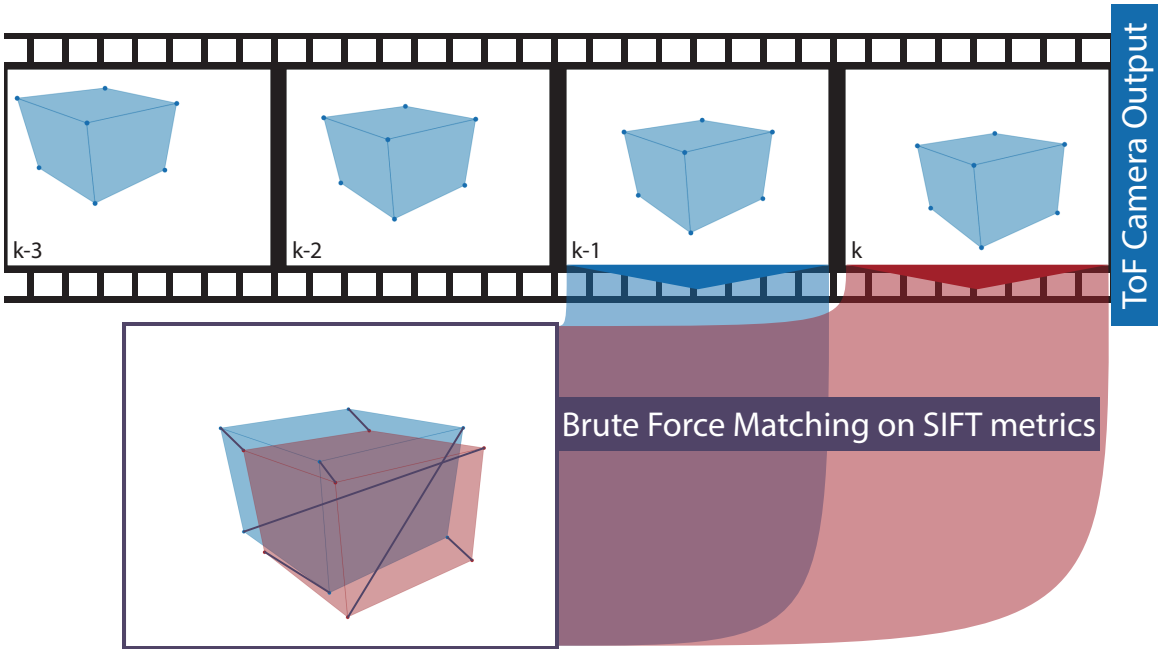


Figure 5.8: First step of ToF motion estimation: Extract SIFT features and brute-force match with prior image. Note that the brute-force matcher also generates false matches.

Each feature coordinate on the picture gets mapped to the 3D space to generate the point cloud. The lens projects objects within the space of a pyramid onto the sensor, which leads to the following coordinate mapping. The coordinate transformation is visualized in image 5.9. Please note the coordinate convention described in section 5.4.1. $a$, $b$ and $c$ being the 3D coordinates, $u$ and $v$ being the coordinates on the image and $d$ being the value of the ToF depth image on position $(u,v)$. $f$ denotes a virtual focal length, merging the camera's field of view (viewing angle $\alpha$) and the image resolution in one number.

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

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


Figure 5.9: Second step of ToF motion estimation: Map features into 3D space

### 5.4.4 ToF Camera: Find rotation and translation

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

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

For example visualized in image 5.10 and with the following calculation:

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

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


Figure 5.10: Rotation and Translation step by step.

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

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

In the example, the results are:

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

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

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

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

In the example:

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

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

The singular value decomposition (SVD), briefly explained in section 4.3.1, splits the covariance matrix $S$ into three separate $3 \times 3$ matrices $U$, $\Sigma$, and $V$. As the SVD in this use case is always applied to matrices of dimension $3 \times 3$, the optimized variant [21] from GitHub [22] can be utilized.

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

In the example:

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

The wanted rotation matrix then follows by calculating:

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

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

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

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

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

In the example:

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

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

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

### 5.4.5 ToF Camera: 3D 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. [17] Low matching accuracy is a known problem in other fields of image processing - like panorama stitching - and is there often solved with an algorithm named RANSAC (random sample consensus).
These applications of the RANSAC algorithm work on two-dimensional images and are not suitable

for three-dimensional point clouds; therefore, the RANSAC algorithm got extended.
The first step of the three-dimensional RANSAC is finding a proper rotation and translation from the brute-force matches. To find these transformations, each matched feature pair gets two other feature pairs randomly assigned. Using the method described in section 5.4.4, the corresponding rotations and translations are calculated on these groups of three feature pairs.
Each group's calculated rotation matrices and translation vectors get checked against all the other matched feature pairs. The data point from the previous image of the feature pair gets moved by the matrix-vector-pair and compared to the data point of the current picture. If the distance of the two points is underneath a threshold, the feature pair is counted for the matrix-vector-pair. On these feature pairs, the average distance from the calculated points to the matched points is measured. The matrix-vector-pair that generates the smallest average distance is likely suitable for further processing.
In a third step, this matrix-vector-pair is recalculated using all brute-force matches within a certain proximity to the calculated point.
Ignoring the SIFT metrics that lead to the brute force matching, all the features get matched anew based on the position alone, using the rotation and translation estimated before. The matrix-vector pair transforms every data point from the previous image. The distance between the transformed point and the closest data point of the current picture determines the new match. If the proximity is below a threshold, the match is accepted.
Finally, the optimal rotation matrix and translation vector are calculated using the method described in section 5.4.4, this time not with only three points but all accepted matches. With more than three points, the calculated rotation and translation is not the exact result, but the result that yields the least square error. [11]
The whole methodology relies on finding the correct rotation and translation from randomly grouped matches.

### 5.4.6 Sensor Fusion with Kalman Filter

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

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

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

contains the sampling rate and gets added by a quaternion multiplication.

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

## 5.5 Video display

TODO: Vulkan Output

# 6 Testing and Results

# 7 Conclusion

## 7.1 Possible improvement

## 7.2 Outlook

# Bibliography


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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[21] 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).

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

# List of Figures

# List of Tables

# List of Abbreviations

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

Table 7.1: List of Abbreviations

# 8 Appendix

## 8.1 Structure of the Git-Repository

### 8.1.1 Documentation

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

### 8.1.2 Source

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

### 8.1.3 Literature

Contains 3rd party documents for technologies used in this thesis

## 8.2 Used software

The following software has been used for this thesis:

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