Master Thesis

Online Gyroscope-Camera Autocalibration for Image Enhancement on Smartphones

Author(s):
Humair, Luc Léonce Humair

Publication Date:
2015

Permanent Link:
https://doi.org/10.3929/ethz-a-010510186

Rights / License:
In Copyright - Non-Commercial Use Permitted
Online Gyroscope-Camera Autocalibration for Image Enhancement on Smartphones

by Luc Léonce Humair

Spring 2015
Declarations of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Online Gyroscope-Camera Autocalibration for Image Enhancement on Smartphones

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s): 
Humair

First name(s): 
Luc Léonce

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the ‘Citation etiquette’ information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date 
Zurich, June 15, 2015

Signature(s)
## Contents

1 Introduction .......................... 1
   1.1 Motivation .......................... 1
   1.2 Contributions and thesis outline ... 3

2 Preliminaries ......................... 4
   2.1 Pinhole camera model ............... 4
   2.2 Lens distortion model ............... 5
   2.3 Rolling shutter camera model ...... 6
   2.4 Camera rotations ................... 7
   2.5 Camera translations ................. 8
   2.6 Camera–gyroscope timestamp delay ... 10
   2.7 Feature matching .................... 10
   2.8 Homography ......................... 12
   2.9 Epipolar geometry ................... 12
   2.10 Coplanarity constraint .......... ... 13
       2.10.1 Feature group selection .... 15
   2.11 Average pixel translation rate .... 15
   2.12 Kalman filtering ................. . 16
   2.13 Extended Kalman filtering ......... 19

3 Related work ......................... 22
   3.1 Camera calibration ................. 22
   3.2 Sensor synchronization ............. 24
   3.3 Combined camera calibration and sensor synchronization ....... 25

4 Algorithms ......................... 27
   4.1 Calibration using OpenCV .......... 27
   4.2 Calibration and online synchronization using extended Kalman filtering ... 28
       4.2.1 The state vector ............... 29
       4.2.2 Prediction update stage ....... 29
       4.2.3 Measurement update stage .... 29
   4.3 Partial calibration and synchronization using average pixel translation rates ... 30
       4.3.1 Synchronization using grid search ... 30
       4.3.2 Calibration using least squares .... 31
   4.4 Calibration and synchronization using coplanarity constraint and grid search ... 34
       4.4.1 Grid search for one feature group ...... 34
       4.4.2 Building the feature groups ....... 35
       4.4.3 Combining results for multiple feature groups ....... 36
## 5 Implementation

5.1 Overview ............................................. 40
5.2 Android Camera2 API ................................. 41
  5.2.1 Camera configuration .......................... 41
  5.2.2 Data output .................................... 42
  5.2.3 Metadata output ................................. 43
5.3 ETHCamera Android application ..................... 43
  5.3.1 Application initialization and camera preview 44
  5.3.2 Initializing the client module and capturing image and sensor data 47
  5.3.3 Other features ................................ 49
5.4 CMake building and testing framework ................. 50
  5.4.1 Single build description ....................... 51
  5.4.2 Other features ................................ 52
5.5 Our CMake-based framework .......................... 53
  5.5.1 Library and executable projects ............... 54
  5.5.2 ETHCamera projects .......................... 54
  5.5.3 Dependency management ....................... 54
5.6 Helper libraries ................................... 55
  5.6.1 The helpers library .......................... 56
  5.6.2 The localHelpers library ..................... 61
  5.6.3 ETHCamera simulator .......................... 64
5.7 Synthetic global shutter camera data .................. 64
5.8 Partial calibration and synchronization using average pixel translation rates 66
5.9 Calibration and synchronization using coplanarity constraint and grid search 67
5.10 Calibration and synchronization using pixel translation rates, coplanarity constraint and grid search 69
5.11 Image enhancement code base ....................... 70
5.12 Image enhancement Android application ............... 71

## 6 Experiments

6.1 Camera calibration using OpenCV ..................... 74
6.2 Extended Kalman filtering based online calibration and synchronization 75
  6.2.1 Verifying the relative orientations ............... 75
  6.2.2 State initialization ............................ 77
  6.2.3 Performance for camera motion around specific camera axes 79
  6.2.4 Conclusions .................................. 83
6.3 Synchronization and calibration using coplanarity constraint ......................... 83
  6.3.1 Impact of feature detection error .................. 85
  6.3.2 Impact of feature detection error on one coplanarity constraint group ........ 86
  6.3.3 Impact of feature detection error on combined coplanarity constraint groups 87
  6.3.4 Non-minimum interval rejection .................. 89
  6.3.5 Dominant groups ................................ 89
  6.3.6 Conclusion .................................. 90
Contents

6.4 Feature matching performance ........................................... 91
  6.4.1 Conclusion ................................................................. 93
6.5 Coplanarity constraint grid search ......................................... 93
  6.5.1 Combined roll and pitch motion ...................................... 93
  6.5.2 Performance for camera motion around specific camera axes .... 95
  6.5.3 Conclusions ............................................................... 96
6.6 ETHCamera simulator ......................................................... 97

7 Conclusion and Future Work .................................................. 100
  7.1 Conclusion ................................................................. 100
  7.2 Further work ............................................................. 101

8 Appendix ............................................................................ 103
  8.1 CMake based framework ................................................ 103
  8.2 Project configuration files .............................................. 103
  8.3 Library project structure ................................................ 104
  8.4 Executable project structure ............................................ 104
  8.5 Android module project structure ..................................... 105
  8.6 Declaring dependencies .................................................. 105
  8.7 Testing ........................................................................... 106
  8.8 Generate projects .......................................................... 106
  8.9 ETHCamera simulator delivery delays .................................. 107

Bibliography ............................................................................. 109
1 Introduction

1.1 Motivation

Most of today’s consumer electronic devices include a wide range of sophisticated sensors. For example, integrated cameras are able to capture high resolution images and videos. Many people constantly carry a smartphone or a tablet. According to Google’s Our Mobile Planet [1], in 2013 already 54% of the Swiss population owned a smartphone and this percentage is constantly increasing. Hence, it is for example very convenient and common practice to capture an interesting scene by just using the integrated camera. Further, inertial sensors, including accelerometers and gyroscopes, determine the motion of a device very accurately. Taking advantage of powerful multi-core processing units, smartphones and tablets are also increasingly capable of running complex operations. This combination creates a platform for a broad range of various applications. Popular examples are image stitching or inertial sensor based games. Besides the opportunities the camera and the sensors offer on their own, combining them is also very interesting for mobile applications. While the camera provides visual cues, the sensors measure the device’s environment and its motion with high accuracy. Hence, such a combination suits well for example for augmented reality applications.

However, taking pictures with mobile devices has a major drawback: as a consequence of several limiting factors (e.g., complexity, energy, cost or form) current integrated cameras are very susceptible to motion. In order to capture images or videos, usually users hold the device by hand. Due to the rather unusual arm pose, it is difficult for users to keep the device steady. Hence, their slight shaking causes image degradations such as motion blur (see Fig. 1.2a). Additionally, rolling shutter causes image distortions as the sensor’s rows are not exposed.

Figure 1.1: Mobile phones and other portable devices have become everyday objects which we carry with us all the time. Capable cameras and sensors enable a lot of applications simplifying or enriching our daily life.
simultaneously (see Fig. 1.2b). High-end cameras use mechanical image stabilization (MIS [3])
also called Optical Image Stabilization (OIS) by Canon or Vibration Reduction (VR) by Nikon to
cope with such motion. Actively shifting lenses or the sensor, this technique aims to compensate
for motion with small magnitude. Unfortunately, space and cost constraints imposed on mobile
hardware usually do not allow for such elaborate systems\(^1\). Another means to avoid unwanted
distortions or blur, rigidly mounting the camera on a tripod, is very inconvenient.

![Motion blur.](image1) ![Rolling shutter distortion (reprinted from [4]).](image2)

**Figure 1.2:** Image degradations introduced by handshake motion.

Given certain assumptions, knowing the actual motion of a device enables compensation of
geometrical distortion and motion blur. Suitable algorithms require knowledge about an accurate
camera model. However, camera manufacturers usually do not provide such parameters. Hence,
there is a need for camera calibration techniques recovering parameters linked to image distortion
and motion blur. Deducing the device’s motion from captured images provides only limited
reliability as the images are also affected by the aforementioned image degradations. In order to
overcome the susceptibility to motion typical for integrated cameras without MIS, motion sensors
are promising sources of additional information. They provide high-resolution, very accurate
and camera independent estimations of the device’s rigid body motion. However, combining
these measurements with image data requires knowledge of their mutual temporal relation.
Knowing the exact time when sensor measurements and camera images are recorded would for
example provide this relation. Integrated cameras and inertial sensors are both equipped with
virtual clocks. Their time measurements are used to assign monotonically increasing timestamps
(in nanoseconds for Android devices) to images and sensor measurements. Unfortunately,
the virtual clocks of current devices do not share a common time base. Hence, timestamps
generated by different clocks at the same point in time have an unknown offset. Identifying the
value of this offset (i.e., sensor synchronization) immediately provides the necessary temporal
relation between images and sensor measurements, hence enabling the aforementioned image
enhancement algorithms.

Precise reconstruction of camera motion through inertial sensors is an important step in
many other applications of smartphones. Inertial sensors may be used to implement sensitive
\(^1\) Actually, some high-end LG and Microsoft Lumia models have hardware stabilization, but this feature is not
common in 2015.
and accurate video rectification and stabilization. Further, applications like image stitching or augmented reality can be built without requiring distinct feature points in the images captured. Therefore, the missing gyro-camera synchronization is a crucial problem which is usually solved using special hardware. Only few related work have addressed this problem on mobile devices without hardware modifications.

1.2 Contributions and thesis outline

This thesis aims to develop an automated camera calibration and timestamp synchronization application. Incorporating gyroscope or accelerometer measurements, it intends to harness the information of the inertial sensors. Finally, it provides an adequate basis to factor in sensor measurements for compensating rolling shutter distortions and/or motion blur. The inputs for camera calibration and timestamp synchronization are sequences of captured images and recorded inertial sensor measurements. The application shall estimate the parameters of a suitable camera model and the delay between camera and sensor timestamps. This thesis includes a thorough literature study on state-of-the-art calibration and synchronization algorithms. Further, evaluating different methods for camera calibration and timestamp synchronization, it proposes a suitable algorithm for a mobile platform. By providing a C++ implementation running on Android’s native environment, it should perform the calibration and synchronization. Once the calibration is successful, we can show its effectiveness in image enhancement applications.

The contents of this thesis are structured as follows: Chapter 2 provides the theoretical background of camera models, properties used in stereo vision and the Kalman filtering algorithm. Chapter 3 presents related work decisive for my thesis and Chapter 4 explains the details of algorithms used or developed. Chapter 5 describes the implementations proposed by this thesis in detail. The experiments we conducted are presented in Chapter 6 along with evaluations of our algorithms. Finally, Chapter 7 concludes this thesis and shows possible future extensions.
2 Preliminaries

2.1 Pinhole camera model

The most basic model of mapping real-world 3D points to a projection on an image plane is described by the pinhole camera model. The camera’s aperture is modeled as a point shaped hole. Light rays emitted by real-world objects pass through such an aperture before being registered by a light sensor, as shown by Fig. 2.1. This model does not include any lenses used to focus light. Given a 3D point \( P = (X, Y, Z) \), its mapped position \( p = (u_x, u_y) \) on the image plane depends on focal length \( f \) and principal point \((c_x, c_y)\). This relationship is described by

\[
\begin{bmatrix}
  u_x \\
  u_y \\
  1
\end{bmatrix}
_{p} =
\begin{bmatrix}
  c_x + f \frac{X}{Z} \\
  c_y + f \frac{Y}{Z}
\end{bmatrix}
\]

what also leads to the basic form of the intrinsic camera matrix \( A \)

\[
\begin{bmatrix}
  u_x \\
  u_y \\
  1
\end{bmatrix}
_{p} =
\begin{bmatrix}
  f & 0 & c_x \\
  0 & f & c_y \\
  0 & 0 & 1
\end{bmatrix}
_{A}
\begin{bmatrix}
  X \\
  Y \\
  Z
\end{bmatrix}
_{P}
\]
where the point $p$ on the image plane is represented in homogenous coordinates.

## 2.2 Lens distortion model

The term image distortion describes the situation of captured images deviating from rectilinear projection. Thus, distorted projections of real-world lines do not result in straight lines on the image. Hence, the lines appear to be bent. We classify the camera’s distortion into categories of radial and tangential distortion. Both types are illustrated by Fig. 2.2. The former distortion is caused by the lens’ shape, e.g., by using a wide-angle lens, while the latter one is caused by misalignment of lens elements [6]. Radial distortion is separated into three different classes, i.e., pincushion, barrel and mustache distortion. Fig. 2.3 further illustrates the effects of both radial and tangential distortion. The most common model used for this kind of image defect is the Brown-Conrady lens model [7, 8]

\[
\begin{bmatrix}
    o'_x \\
    o'_y 
\end{bmatrix}
= \begin{bmatrix}
    o_x (1 + k_1 r^2 + k_2 r^4 + \cdots) + (p_2 r^2 + 2p_1 o_x o_y) (1 + p_3 r^2 + p_4 r^4 + \cdots) \\
    o_y (1 + k_1 r^2 + k_2 r^4 + \cdots) + (p_1 r^2 + 2p_3 o_x o_y) (1 + p_3 r^2 + p_4 r^4 + \cdots)
\end{bmatrix},
\]

where:

- radial contribution
- tangential contribution

![Figure 2.2: Image distorted by radial distortion.](image)

![Figure 2.3: Effects of distortion on the predicted position of a feature. Its actually observed position differs by a radial shift $\Delta r$ and a tangential one $\Delta \theta$.](image)
2 Preliminaries

\( o = \) The predicted (i.e., undistorted) position of projected feature.
\( o' = \) The observed position in distorted image.
\((c_x, c_y) = \) The principal point coordinates.
\( r = \sqrt{(o_x - c_x)^2 + (o_y - c_y)^2} \)
\( k_n = \) The \( n \)th radial distortion coefficient.
\( p_n = \) The \( n \)th tangential distortion coefficient.

2.3 Rolling shutter camera model

The shutter of a camera is responsible for exposing pixels of light sensors in a specific order for a certain amount of time. Digital camera devices use different shutter behaviors depending on the installed light sensors. Top shelf models employ more sensitive CCD\(^1\) sensors which usually come with a global shutter. Hence, the exposure starts and ends at the same time for all of the sensor’s pixels. Next, the registered charges are serially processed in an analog-to-digital converter. In contrast to the CCD technology, CMOS\(^2\) sensors allow to process different areas of the sensor simultaneously. This property enables rolling shutter technology where the rows of the sensor usually are exposed consecutively from top to bottom. Hence, they may be processed exploiting a certain amount of parallelism, leading to high frame rates with less expensive and complex hardware with regard to CCDs. Further, CMOS sensors usually are more energy efficient than CCDs. Due to these benefits, CMOS sensors are heavily used in consumer electronics, e.g., smartphones and camcorders where costs, complexity, size and energy efficiency are of high concern. Besides the benefits, rolling shutter may cause unwanted image deficiencies called “rolling shutter effects” as shown by Fig. 2.5. Exposing the rows at different points in

---

\(^1\) Abbreviation for charge-coupled devices.
\(^2\) Abbreviation for complementary metal-oxide-semiconductor.
Preliminaries

(a) Pan (left to right).

(b) Tilt (upwards).

(c) Varying illumination.

Figure 2.5: Rolling shutter artifacts. Fig 2.5a and Fig. 2.5b show the effects of camera or object motion during capturing. Both images are reprinted from [9]. The solid shape is the correct mesh while the white wire-frame is the distorted result of capturing. Fig. 2.5c shows the effect of varying illumination during capturing a scene using a rolling shutter camera (reprinted from [10]).

Figure 2.5b shows the effects of camera or object motion during capturing. Both images are reprinted from [9]. The solid shape is the correct mesh while the white wire-frame is the distorted result of capturing. Fig. 2.5c shows the effect of varying illumination during capturing a scene using a rolling shutter camera (reprinted from [10]).

time leads to image distortion in case of moving objects or camera motion. Additionally, it may lead to exposure issues if lighting conditions vary between the exposure intervals of the rows. We model the rolling shutter distortion effects by defining the exposure start time of pixel \( u = (u_x, u_y) \) in frame \( i \) as

\[
t(u, i) = t_i + t_r \frac{u_y}{h - 1}
\]

where \( t_i \) is the frame’s timestamp, \( t_r \) is the time between the start of the its first and last row exposure (i.e., rolling shutter skew) and \( h \) is the number of rows of an image frame.

2.4 Camera rotations

Rigid body orientation can be modeled as independent rotations around three distinct axes of a Cartesian coordinate system. This represents the rotation in Tait-Bryan (or Cardan) angles, a special kind of Euler-angles. The direction of rotation is defined according to the right hand rule, i.e., a positive rotation describes a counter-clockwise rotation. Each gyroscope measurement returned by the android API for timestamp \( t \) consists of a vector \( \omega^0(t) = [\omega_x^0(t), \omega_y^0(t), \omega_z^0(t)]^T \) describing the current rotation velocities in radians per second for the each axis. Fig. 2.6a shows the axes used by the Android API [11]. Our experiments were conducted holding the smartphone in landscape orientation. Accounting for this rotation, we always transform the original rotation velocities in order obtain values corresponding to the coordinate system shown in Fig. 2.6b:

\[
\omega(t) = \begin{bmatrix}
0 & -1 & 0 \\
1 & 0 & 0 \\
0 & 0 & -1 \\
\end{bmatrix} \begin{bmatrix}
\omega_x^0(t) \\
\omega_y^0(t) \\
\omega_z^0(t) \\
\end{bmatrix}
\]

Further, we define camera roll, pitch and yaw motion as rotations around \( y, x \) and \( z \) axis respectively, as illustrated by Fig. 2.6b.
### 2 Preliminaries

#### 2.5 Camera translations

Most of today’s combined camera calibration and synchronization techniques do not incorporate any information about device translation. This is because of multiple reasons. First, integrating it results in a large drift because the data is noisy. Second, gravity compensation is difficult. Hence, they assume for example zero translation between consecutive frames. However, knowing the exact translations would free these techniques from making such potentially inaccurate assumptions. The estimations would get better as they would match the real world conditions more exactly.

![Coordinate systems used by Android’s Sensor API and our thesis.](image)

**Figure 2.6:** Coordinate systems used by the Android API and our thesis. Fig. 2.6b also visualizes roll, pitch and yaw camera motions.

![Integration of accelerometer data.](image)

**Figure 2.7:** Integration of accelerometer data. The measurements were measured by a smartphone steadily resting on a flat surface. We expect zero translation, however, measurement errors introduce a drift when integrating the data. We omitted the z-axis rotation as it captures acceleration caused by gravity.
State of the art smartphones are usually equipped with an accelerometer sensor. As it reports acceleration in \([m/s^2]\), the measured data has to be integrated twice in order to obtain translation in \([m]\). Unfortunately, each measurement has a small error imposed. Integrating the accelerations twice amplifies these errors quite severely, leading to a so called drift which has to be accounted for. Fig. 2.7 shows the integration of accelerometer measurements captured by a device steadily resting on a flat surface. Given the fixed position of the device, we would expect zero translation after integrating the acceleration twice. However, the graphs clearly show the influence of measurement errors causing a drift, leading to non-zero translation. Thus, translation measurements are significantly less accurate than orientation measurements \([12, 13]\). Besides drift compensation, getting a clean picture of ground truth translations would also require to properly compensate the acceleration introduced by gravity.

Including translation into feature matching based estimations would also require knowledge regarding the scene depth. Obtaining this value for each pixel is quite difficult unless a depth sensing camera is used. Actually there are existing different types of devices capable of such estimations, i.e., light field \([14]\), time of flight or structured light cameras. However, gyroscope-only based techniques are independent of such information. Further, rotations have far more leverage on image motion than translation, given a suitable scene depth. Assuming a feature at a distance of 3\(m\) from the camera, a translation of 1\(cm\) is equivalent to a rotation of 0.19\(^{\circ}\), which is far more likely to occur \([15]\). Fig 2.8 further illustrates such a situation. A camera observes a feature \(P\) on a planar scene while having a distance of 3\(m\). The intersection between the red cone and the planar scene indicates the amount of scene translation caused by a rotation of 0.19\(^{\circ}\). This amount equals to 1\(cm\), as illustrated by the figure. Translating the scene is equivalent to translating the camera by an equal amount into the opposite direction. Hence,

\[\text{Figure 2.8: Equivalence of camera translations and rotations. A camera a with distance of 3m to a planar surface captures a feature P. The red cone represents a camera rotation of 0.19^{\circ}. This shows that a slight camera rotation of 0.19^{\circ} is enough to produce a blur equivalent to a large 1cm camera translation.}\]
the camera rotation of $0.19^\circ$ causes at least the same magnitude of relative scene translation as a camera translation of $1 cm$.

### 2.6 Camera–gyroscope timestamp delay

Most of today’s smartphones do not offer synchronized clocks for camera image and inertial sensor timestamps. Starting with the new Camera2 API shipped with Android Lollipop (API 21), there is support implemented for such a feature. However, even the Motorola Nexus 6 (flagship Android phone in 2014) does not actually realize it. Therefore, we need to recover the amount of this delay by computation.

![Image](image.png)

**Figure 2.9:** Visualization of average image and gyroscope pixel translation rates represented by $r_f(t)$ and $r_g(t, f)$. We assume a known focal length $f$. The timestamp difference causes a horizontal shift between the translation rates.

### 2.7 Feature matching

This technique is used to establish correspondences between distinctive regions in consecutive frames. Such regions, called feature points, are usually located where image intensities vary strongly, e.g., at corners or strong edges. There are different feature detectors existing that determine the locations of these regions. Popular examples are SIFT [16], FAST [17] or GFTT [18]. Since feature points are located in distinctive image regions, it is very likely to find corresponding ones in adjacent frames. A matching as shown in Fig 2.10 then contains information about either the camera or the object motion between the frames.

Feature correspondences are usually determined using either optical flow or descriptor matching. The former uses spatio-temporal image intensity variations to estimate a the displacement of a feature between to frames. It analyzes changes of image intensities at certain locations between frames to predict the corresponding motion. For example, the Kanade–Lucas–Tomasi
(KLT) feature tracker \cite{18, 19} is a well-known and extensively used optical flow based algorithm for feature tracking. In contrast, descriptor matching compares the similarity of image regions to identify similar ones between consecutive frames. As detected image features are located in distinctive parts of the image, they serve well for descriptor matching. As illustrated in Figure 2.11, each feature gets a descriptor assigned describing its surrounding area. Afterwards, these descriptors are used to determine the similarity of features detected in consecutive frames.

Feature matching by its own is likely to include wrong feature pairs. The accuracy of the correspondences is usually increased using extra algorithms. Random sample consensus (RANSAC) \cite{21} is a very popular feature matching refinement method. It computes different fundamental matrices\footnote{See Section 2.9 for more details concerning fundamental matrices and the epipolar constraint.} between two consecutive frames using randomly selected feature correspondences. Next, it searches for all the inliers, i.e., feature correspondences that are well explained by a fundamental matrix. Finally, the set consisting of the largest number of inliers is kept. Track-re-track \cite{22, 23} is a method to increase the accuracy of optical flow based feature matching. After performing feature tracking from preceding frames to succeeding ones, the tracking is also performed in the reverse direction. In the end, only the features matching in both directions are kept.
2 Preliminaries

2.8 Homography

A homography describes the relationship between two images of the same planar surface. Given the same setup as shown by Fig. 2.12 in Section 2.9, a homography $H$ maps coplanar feature points $p_i$ of camera $C_1$ to feature points $p'_i$ of camera $C_0$:

$$p'_i = K_0 H C_1 C_2 K_1^{-1} p_i$$

where $K_0$ and $K_1$ are the camera intrinsic matrices for camera $C_0$ and $C_1$ respectively.

Homographies are also applicable to non-coplanar points if the poses of two cameras $C_0$ and $C_1$ do only differ by a rotation around the optical centers. More formally, if $O_0 \equiv O_1$ then homographies describe the relationship between non-coplanar scenes. Calibration and synchronization techniques using homographies and assuming no translations make use of this fact.

2.9 Epipolar geometry

Given a 3D scene and its 2D projections (as recorded by two pinhole cameras), epipolar geometry describes relations between the 3D point and the corresponding 2D projections. These relations imply constraints on the projected 2D points’ locations based on the camera’s rigid body motion. Fig. 2.12 shows a illustrates the projections of a point onto two images. $O_0$ and $O_1$ represent optical centers of a left and a right camera, both capturing the same 3D feature $P$. $p$ and $p'$ indicate the projections of $P$ on the camera’s image planes. The epipolar plane is determined by the points $P$, $O_0$ and $O_1$. The baseline connecting both cameras’ optical centers determines the epipoles $E_0$ and $E_1$ on the intersections with the image planes. Further, $R$ and $t$ describe the relative rotation and translation from camera pose 0 to pose 1. $\vec{p}$ and $\vec{p}'$ show the projection lines indicating the direction from the respective optical center to the 3D feature. Further, the epipolar lines $l$ and $l'$ are defined as the lines intersecting $\{E_1, p\}$ and $\{E_0, p'\}$ respectively. Assuming known camera translation and rotation, these geometric relations lead to the following observations:

- Assuming $p$ is known, then the epipolar line $l'$ is known. It defines the set of possible locations of projection $p'$ of $P$ on the image plane of the other camera.

- Assuming both projections $p$ and $p'$ are known, then $P$ can be triangulated by using the respective projection vectors $\vec{p}$ and $\vec{p}'$.

which finally result in the epipolar constraint

$$((\vec{p}' \times R \vec{p}) \cdot \vec{t}) = 0$$

indicating the coplanarity of $\vec{p}'$, $R \vec{p}$ and $\vec{t}$.

---

4 The point where the other camera would be visible in the image.
The fundamental matrix $F$ expresses the epipolar constraint in linear algebra. Given a known feature point $p$, $Fp$ describes the epipolar line $l'$ where the corresponding point $p'$ may be located. Hence, the following holds for each pair of corresponding points $p$ and $p'$:

$$p'^T F p = 0$$

### 2.10 Coplanarity constraint

The epipolar constraint still requires knowledge of translation between successive image frames. Combining multiple epipolar planes enables to formulate a constraint independent of translation. This fact results in the coplanarity constraint and will be the basis of our calibration methods. Based on the epipolar constraint, the coplanarity constraint describes a relationship between the projections of different features captured by different camera poses. Fig. 2.13 shows an example of three pairs of camera poses $C_i$ and $C_{i+3}$ capturing 3D features $P_i$ where $i \in [0, 1, 2]$. Translations $t_i$ and rotations $Q_i$ describe the camera’s rigid body motion between frame $C_i$ and $C_{i+3}$. For the sake of completeness, we introduce an individual rotation $R_j$ for each pose $C_j$. $R_j$ and $R_{j+3}$ share a common base rotation, hence

$$Q_j = R_j^{-1} * R_{j+3}.$$ 

Each pair of camera poses is associated with exactly one feature, such that each one is captured by exactly two poses. Such a constellation of features and poses builds one group.
Figure 2.13: Visualization of one group used for coplanarity constraint computation. It consists of three 3D features $P_i$, each captured by two camera poses $C_i$ and $C_{i+3}$. Hence, each 3D feature defines two feature vectors $p_i'$ and $p_i$. Further, in combinations with rotations $Q_i$ between camera poses $C_i$ and $C_{i+3}$, these vectors define the epipolar plane normals $n_i$.

Using the epipolar constraint, we know that $p_i', Q_i p_i'$ and $\vec{t}$ define an epipolar plane of a matched feature, leading to

$$\vec{n}_i = R_i (p_i' \times Q_i p_i') = R_i p_i' \times R_{i+3} p_i$$

indicating the plane’s normal. Note that $\vec{n}_i$ is perpendicular to $\vec{t}$, i.e.,

$$\vec{n}_i \cdot \vec{t}_i = 0.$$

Assuming a capturing frequency of about 30 frames per second, all features belonging to one group are captured within a time slot of $33ms + t_r \leq 66ms$. Hence, translations $\vec{t}_i$ can be well approximated to be collinear in such a short period of time [24]. As previously stated, the epipolar plane normals $\vec{n}_i$ are normal to the translation vectors $\vec{t}_i$. Therefore, the normals are linearly dependent, which means the matrix built by stacking these vectors as columns must have determinant zero.

$$\det \left[ (R_0 p_0' \times R_3 p_0) | (R_1 p_1' \times R_4 p_1) | (R_2 p_2' \times R_5 p_2) \right] = 0.$$

Using the determinant of the matrix consisting of column vectors has several advantages:

- **Quantization of coplanarity:**
  The determinant represents the volume of a parallelepiped spanned by the column vectors [25]. Hence, it establishes a measurement for the actual coplanarity of the vectors.

- **Also covers corner cases:**
  Using the determinant also works for zero-translation scenarios. In contrast, a measurement like $\text{norm}(n_1 \times n_2) \cdot \text{norm}(n_3)$ would hit a singularity in such a situation.

- **Ease of computation:**
  Computing a $3 \times 3$ matrix’s determinant just requires a small number of additions and
multiplications. Hence, it only requires very little computational effort compared to more evolved measurements.

### 2.10.1 Feature group selection

Generally the features for computing the coplanarity constraint may be grouped randomly. As described in Section 2.3, the CMOS camera used by this thesis consecutively exposes sensor rows from top to bottom. Hence, the closer the $y$-axis coordinates of features, the closer the respective exposure time instances. As mentioned in the previous section, the coplanarity constraint assumes collinear translations for all pairs of matched features. Thus, the constraint’s accuracy is potentially improved by grouping features with close $y$-axis coordinates in each frame. Such a heuristic ensures short periods of time between the exposure of features of each frame compared the delay between the capturing of two consecutive frames. Hence, the collinearity of translations is way more likely \[24\]. Choosing relatively distant $x$-axis coordinates makes sure to have features from different image regions in each group. Therefore, dependencies between matched feature pairs are reduced. Hence, it is beneficial to choose vertically close and horizontally relatively far features for each three-group used for computing the coplanarity constraint. The coplanarity constraint measure then can be generalized to lots of frames and many measurements using averaging as described by Section 4.4.3.

### 2.11 Average pixel translation rate

An alternative measure of camera motion is the average pixel translation rate between consecutive frames. It has been used by Karpenko et. al. \[12\], Hwangbo et al. \[26\] and Jia \[24\] to compare the motion captured by the camera and measured by the gyroscope. The measurement is quantified as the average magnitudes of feature displacement of all of the frames’ matched image features. Primarily used for visualizing camera calibration and timestamp synchronization performance by \[12\], \[24\], Hwangbo et al. \[26\] actually did timestamp synchronization by taking advantage of this measurement. Assuming a global shutter camera, zero translation and ignoring $z$-axis rotation, we can compute the pixel translation rate for both the gyroscope rotations and the image feature matches\[6\].

The average $x$-axis pixel translation rate of the frame with exposure start timestamp $t_i$ is computed as

$$r_{x}^{f}(t_i) = \frac{\sum_{m \in M(t_i)} (m_x - m'_x)}{|M(t_i)|(t_i - t_{i-1})}$$

\[1\] I.e., approximately 33$ms$ for a camera that captures 30 frames per second.

\[6\] Note that the effect of $z$-axis rotation on image features should cancel out by averaging, given features exhibiting suitable accuracy and uniform distribution.
where $M(t_i)$ represents all matched features of frame with timestamp $t_i$, $m$ is the location of a feature in the current frame and $m'$ the corresponding (i.e., matched) location in the preceding frame. Hence, each frame has an average pixel translation rate of

$$r^f(t_i) = \frac{(r^f_x(t_i), r^f_y(t_i))}{2}.$$ 

The average x-axis pixel translation rate of a gyroscope measurement with timestamp $t$ and focal length $f$ is computed as

$$r^g(t, f) = f \omega_x(t)$$

resulting in an average pixel translation rate of $r^g(t, f) = (r^g_x(t, f), r^g_y(t, f))$ for each gyroscope measurement.

![Figure 2.14: Average pixel translation rate calculated from camera and gyroscope data, assuming a given intrinsic camera matrix. The gyroscope measurements were recorded while periodically rotating around a distinct camera axis. Note how well the motion estimated from images and the motion estimated from the gyroscope measurements are aligned.](image)

### 2.12 Kalman filtering

The Kalman filter is an algorithm to estimate unknown parameters of a linear system. The filter only requires the model of a system and a series of (possibly defected) measurements as input. It then iteratively computes estimates of the system’s unknown variables while minimizing the estimation’s error. Hence, using this filter on a series of measurements results in a better estimation with respect to only using one measurement. More formally, given a stream of noisy input measurements, this filter produces the best possible estimate of the underlying state in the sense of statistical optimality. Invented in 1960 by Rudolph Emil Kálmán [27], the filter was first incorporated in the Apollo computer in 1960s for trajectory estimation. It became an important algorithm for a broad class of problems and is pervasively used in fields like navigation, health, economics or defense [28]. Further, it is also applied increasingly in computer graphics applications [29]. The article by Ramsey Faragher published in the IEEE Signal Processing Magazine [30] provides a simple and intuitive derivation of the filter based on an example.
Mathematical model

The filter’s model assumes a state $t$ of some underlying system that is derived from a prior state $t - 1$ according to

$$x_t = Fx_{t-1} + Bu_t + w_t$$

where:

- $x_t$ The system’s actual state at time $t$ as a vector. This state may not be directly observable but measurable (e.g., position, velocity, ...).
- $u_t$ A vector containing all control inputs that change the system’s state (e.g., acceleration).
- $F$ State transition matrix that maps the knowledge of the system at time $t$ to time $t + 1$.
- $B$ Control input matrix which applies the control inputs onto the state variables.
- $w_t$ Vector containing process noise. Assumed to be drawn from a multivariate normal distribution defined by covariance matrix $Q_t$.

Measurements of the actual state are modeled by

$$z_t = Hx_t + v_t$$

where:

- $z_t$ The vector of measurements.
- $H_t$ The transformation matrix that derives measurement values from the actual system state.
- $v_t$ Measurement noise that is assumed to be Gaussian white noise, i.e., each sample has a normal distribution with zero mean, with covariance $R_t$.

Algorithm

The algorithm incorporates at each point in time the prior knowledge of the system and then enhances it using measurement values. As the model contains errors/noise, the prediction and measurement values basically define PDFs (probability density functions). Hence, Kalman filtering aims to compute the best estimate of the system’s state by combining both PDFs. The key point of the filter is the fact that combining two Gaussian PDFs results in another PDF having a smaller variance than both of the combined ones (what actually increases the accuracy).

The Kalman filtering algorithm proceeds iteratively by performing prediction and measurement update steps as explained below. Fig. 2.15 shows a complete overview of the steps performed by the algorithm.
Prediction (time) update stage

This stage is defined by the following equations

\[ \hat{x}_{t|t-1} = F_t \hat{x}_{t-1|t-1} + B_t u_t \]  
(2.3)

\[ P_{t|t-1} = F_t P_{t-1|t-1} F^T_t + Q_t \]  
(2.4)

where:

\( \hat{x}_{t|t-1} \) A priori estimate (before incorporating the measurement results) of the system’s current state, given the previous state.

\( \hat{x}_{t-1|t-1} \) A posteriori estimate of the system’s (previous) state.

\( P_{t-1|...} \) A priori/posteriori covariances between the terms in the state vectors.

Equation (2.3) projects the internal state to the next time step. Further, Equation (2.4) projects the error covariance to the next step.

Measurement update stage

\[ \hat{x}_{t|t} = \hat{x}_{t|t-1} + K_t (z_t - H_t \hat{x}_{t|t-1}) \]  
(2.5)

\[ P_{t|t} = (1 - K_t H_t) P_{t|t-1} \]  
(2.6)

where:

\( K_t \) The Kalman gain. It represents the importance of the difference between measurement and prediction and is computed before the update as:

\[ K_t = P_{t|t-1} H^T_t (H_t P_{t|t-1} H_T^T + R_t)^{-1} \]

Equations (2.5) and (2.6) update the state and the error covariance estimations using measurements.
2 Preliminaries

2.13 Extended Kalman filtering

The basic Kalman filter explained in the previous section enables to perform state estimation for linear processes. In contrast, its extension also allows state estimation for non-linear processes. This extension linearizes about the current mean and covariance [29] in order to get an estimate of the system’s state. It does so by using partial derivatives of the process and measurement functions.

Mathematical model

The stochastic difference equation is now formulated as

\[ x_k = f(x_{k-1}, u_k, w_k) \]
\[ z_k = h(x_k, v_k). \]
2 Preliminaries

Basically we replace the state transition matrix $F$ by a non-linear state transition function $f$ and the transformation matrix $H$ by a non-linear transformation function $h$. As before, $w_k$ and $v_k$ represent process and measurement noise, respectively. In practice we do not have access to the actual values of these variables. However, we can approximate the state and measurement vectors by

$$
\tilde{x}_k = f(\hat{x}_{k-1|k-1}, u_k, 0) \\
\tilde{z}_k = h(\tilde{x}_k, 0)
$$

finally leading to the estimates using a linearization akin to a Taylor series

$$
x_k \approx f(\hat{x}_{k-1|k-1}, u_k, 0) + A(x_{k-1} - \hat{x}_{k-1|k-1}) + W w_k \\
z_k \approx h(\tilde{x}_k, 0) + H(x_k - f(\hat{x}_{k-1|k-1}, u_k, 0)) + V v_k
$$

where:

- **A** The Jacobian matrix of partial derivatives of $f$ with respect to $x$.

\[
A_{[i,j]} = \frac{\partial f_i}{\partial x_j}(\hat{x}_{k-1|k-1}, u_k, 0)
\]

- **W** The Jacobian matrix of partial derivatives of $f$ with respect to $w$.

\[
W_{[i,j]} = \frac{\partial f_i}{\partial w_j}(\hat{x}_{k-1|k-1}, u_k, 0)
\]

- **H** The Jacobian matrix of partial derivatives of $h$ with respect to $x$.

\[
H_{[i,j]} = \frac{\partial h_i}{\partial x_j}(f(\hat{x}_{k-1|k-1}, u_k, 0), 0)
\]

- **V** The Jacobian matrix of partial derivatives of $h$ with respect to $v$.

\[
V_{[i,j]} = \frac{\partial h_i}{\partial v_j}(f(\hat{x}_{k-1|k-1}, u_k, 0), 0)
\]

**Algorithm**

The previously stated approximations allow to formulate an iterative solver that works similar to conventional Kalman filtering by iteratively performing prediction and measurement update steps.
Prediction update stage

$$\hat{x}_{t|t-1} = f(\hat{x}_{t-1|t-1}, u_t, 0)$$

$$P_{t|t-1} = A_t P_{t-1|t-1} A_t^T + W_k Q_t W_k^T$$

Measurement update stage

$$K_t = P_{t|t-1} H_t^T (H_t P_{t|t-1} H_t^T + V_t R_t V_t^T)^{-1}$$

$$\hat{x}_{t|t} = \hat{x}_{t|t-1} + K_t (z_t - h(\hat{x}_{t|t-1}, 0))$$

$$P_{t|t} = (1 - K_t H_t) P_{t|t-1}$$

Update estimation using measurements $z_t$

Figure 2.16: Overview of the extended Kalman filter’s iterative scheme.
3 Related work

In this section we give a brief overview of related work that inspired our thesis. First, we show the most recent approaches for camera calibration. We continue by describing other work for sensor synchronization. Finally, we also present techniques combining both camera calibration and sensor synchronization.

3.1 Camera calibration

A simple yet common technique to estimate camera parameters of global shutter cameras consists of using the OpenCV camera calibration tool [31] based on an algorithm by Zhengyou Zhang [32] and a Matlab toolbox by Jean-Yves Bouguet [33]. It requires a set of images of a known planar calibration pattern in order to recover camera parameters including distortion coefficients. This method is very convenient to compute baseline estimations of the camera parameters. However, it does not consider rolling shutter effects. Therefore users have to be very careful while capturing the pattern using a rolling shutter camera. Joshi et al. [13] implemented an elaborate system consisting of a high speed camera attached to the camera to be calibrated, as shown by Fig 3.1. The images captured by the high speed camera were used to reconstruct the camera positions. Finally, they used these camera position estimations to do the camera calibration. This method is too complex for our application as it requires additional hardware, well aligned cameras and a suitable scene to be captured. Another technique proposed by
Hwangbo et al. [26] uses a homography based transformation of matched feature points based on gyroscope measurements. This results in a non-linear optimization problem. In contrast to the coplanarity constraint it ignores the translation between frames. The previously mentioned approaches assume a global shutter camera. It thus lacks of an estimation for the rolling shutter delay. Thus, the approaches described in [15, 22, 34] extend the camera calibration method with rolling shutter parameter estimation using recorded images of a flashing light source with a known frequency [35]. In 2012, Ringaby and Forssén [36] improved this method for partly illuminated sensors in case the device’s lenses are not removed. Fig. 3.2 shows the complex setup necessary for this method. Besides being impractical for regular users, this technique also tends to be imprecise and error prone [12]. Hence, Oth et al. [37] propose a calibration method for rolling shutter cameras only requiring a video of a known planar calibration pattern, as shown by Fig 3.3. Other approaches use combined calibration and synchronization involving

**Figure 3.2:** Calibration of rolling shutter using a flashing light source. Fig. 3.2a shows the necessary equipment; an oscilloscope, a function generator and a flashing LED. Further, Fig. 3.2b shows a captured image of the LED.

**Figure 3.3:** Rolling shutter camera calibration using a video sequence of a planar pattern. The figure shows impact of rolling shutter effects. The red and the green dots indicate reprojected checkerboard corners. The former are generated by using a rolling shutter model while the latter are produced by a global shutter model. The red ones, accounting for the global shutter effects, match the actual locations of the chessboard corners. In contrast, the green ones do not factor in the rolling shutter effects. The displacement of these points demonstrates the necessity of considering rolling shutter effects for camera calibration (reprinted from [37]).
3 Related work

feature correspondences and inertial sensor measurements.

3.2 Sensor synchronization

Normal smartphones without special hardware modifications usually have some delay between image and inertial sensor timestamps. Such delays may be caused by different hardware clocks, unknown processing paths or scheduling decisions in the operating system. Hence, besides spatial calibration, temporal synchronization is also necessary. Hanning et al. [34] propose a method which projects a point \( y_i \) to the preceding frame by interpolating device orientations. The orientations are computed by incorporating both gyroscope and accelerometer measurements based in a Kalman filter [27]. Combining the gyroscope and the accelerometer by Kalman filtering yields more accurate rotation measurements while only imposing very little extra computational effort. They form a minimization problem using a cost function consisting of the distance of the projected points to their corresponding features \( x_i \). The timestamp delay is recovered by solving this minimization problem via grid search. Grid search suits very well if results only need a certain degree of accuracy and the search interval is small enough. Hence, this minimization approach is also interesting for our thesis. Bell et al. [15] formulate a similar minimization problem while only considering gyroscope measurements. Further, they use an asymmetric point grid that is easy to track, where the centroids serve as feature points. Using such a calibration pattern allows to generate accurate point correspondences, even in the presence of motion blur and rolling shutter effects. As we aim to develop an as simple as possible calibration algorithm, we try to avoid the usage of a calibration patterns. Hwangbo et al. [26] use the phase lags of pixel translation rates computed by gyroscope data and feature

![Figure 3.4: The cost function used by Hanning et al. evaluated at different values of \( t_d \) (reprinted from [34]).](image)

![Figure 3.5: Pixel translation rates used by Hwangbo et al. for sensor synchronization.](image)
matching when given a small sinusoidal camera motion around a specific axis. Using such pixel translation rates ignores important aspects of CMOS cameras. Further, it does not support all camera motion patterns. Nevertheless, it is a fast yet reasonably accurate algorithm to compute an initial estimate of the timestamp shift. Also using pixel translation rates, Ovrvén et al. [9] perform an initial sensor synchronization. As the results only have limited accuracy they use the previously described method by Hanning et al. [34] to improve the synchronization to sub-frame accuracy. Ovrvén et al. replace the grid search used by Hanning et al. with the much more efficient Brent’s method [38] as they already have an initial synchronization. Further, this work also estimates alignment errors between the gyroscope and the camera axes by analyzing device rotations around two orthogonal axes. Other approaches [39, 40] require users to compare captured images of point grids and synthetic blur kernels generated by using gyroscope measurements.

![Figure 3.6: User driven sensor synchronization by Šroubek at al. Fig. 3.6a shows a point grid displayed on a LCD screen affected by the rolling shutter effect. Fig. 3.6b to Fig. 3.6h show blur kernels generated from gyroscope data with different user specified timestamp delays (reprinted from [40]).](image)

### 3.3 Combined camera calibration and sensor synchronization

The methods described in this section establish all ground-truth parameters simultaneously by combining matched features from captured frames with measurements from inertial sensors. Quite a few techniques omit translation as its computation from accelerometer data is error prone. Further, rotations usually dominate the camera motion when taking pictures and videos as explained in Section 2.5. Karpenko et al. [12] and Ringaby et al. [41] derive homographies between consecutive frames by incorporating gyroscope rotations. Very similar to Hanning et al. [34] they formulate a non-linear optimization problem by applying these homographies to all features $x_i$ and comparing the distance of the resulting points to the corresponding matched points $y_i$. Finally, they solve the problem using coordinate descent by direct objective function evaluation. Confronted with non-negligible camera translation with respect to feature distance,
3 Related work

Figure 3.7: Evolution of the online estimation produced by Kalman filtering and the coplanarity constraint (reprinted from [24]).

pure rotational models lose on accuracy [26]. Hence, the resulting estimations will not match ground-truth values anymore. Jia and Evans [24] cope with the problem of translation by proposing an algorithm based on the coplanarity constraint. It assumes a fixed translation direction between consecutive frames. This assumption is justified by the short time interval between adjacent frames. Further, their algorithm is based on Kalman filtering. Hence, this is the only approach for an online computation so far. Further the Kalman filter is able to dynamically adapt to variations in the estimated parameters.
4 Algorithms

We tested different algorithms for camera calibration and sensor synchronization. We will show a well established camera calibration technique using known planar pattern. Further, we are going to present three methods for joint calibration and synchronization. The following sections will explain them in more detail.

4.1 Calibration using OpenCV

Based on an algorithm by Zhengyou Zhang [32] and a Matlab toolbox by Jean-Yves Bouguet [33] OpenCV, an open-source computer vision library, offers a very accurate global shutter camera calibration tool\(^1\). This technique only requires set of images capturing a known planar calibration pattern from different camera poses. The algorithm finds 3D-2D correspondences and computes the optimal camera parameters that minimize the reprojection error over all images. Using a planar black and white pattern with a known shape allows to implement fast yet accurate feature detection. The library currently supports a chessboard or a point grid where the latter one may be either symmetric or asymmetric. Fig. 4.1 shows OpenCV's pattern detection on a chessboard and a asymmetric point grid. This grid is the only pattern allowing to perform unambiguous camera pose estimation. The symmetric patterns do not contain any information from which side the pattern actually has been captured. Using the captured images, the following camera intrinsic parameters are estimated:

\( f_x, f_y \) Focal length in horizontal and vertical direction, respectively.
\( c_x, c_y \) Principal point coordinates.
\( k_1, k_2, \ldots, k_6 \) Radial distortion coefficients.
\( p_1, p_2 \) Tangential distortion coefficients.

Additionally this calibration method also outputs the camera's extrinsic parameter

\([R|t]\) Joint rotation-translation matrix.

for each individual input image.

The calibration algorithm has to be supplied with object coordinates describing the shape of the input pattern. Note that the estimated camera intrinsic parameters are returned in pixels i.e., they are computed independently of the unit of the object coordinates. In contrast, the

4 Algorithms

(a) Chessboard pattern detection.  
(b) Circle pattern detection.

Figure 4.1: Visualization of OpenCV’s calibration pattern detection.

returned camera extrinsics map the object coordinates to the camera coordinate system. Hence, the respective unit is dependent of the units in which the object coordinates are given.

4.2 Calibration and online synchronization using extended Kalman filtering

Chao Jia proposed in 2014 an online\(^2\) camera calibration and timestamp synchronization algorithm [24] based on the coplanarity constraint and extended Kalman filtering (EKF). Taking an initial guess of the following parameters, matched image features and gyroscope measurements, the algorithm iteratively refines the estimations of:

- \( f \) Focal length.
- \( c_x, c_y \) Principal point coordinates.
- \( t_r \) Readout time for each frame.
- \( t_d \) Constant delay between recorded timestamps of camera and gyroscope.
- \( q_c \) Relative orientation of camera to the gyroscope.
- \( b_g \) Bias of the gyroscope.

Finally the algorithm is expected to converge to the actual values after a suitable number of input frames. Further, it suits as an online algorithm as the Kalman filtering is able to adapt to changes of the underlying system over time. Later versions of the algorithm also include estimations of:

- \( \kappa_1, \kappa_2 \) Radial lens distortion.

\(^2\) I.e., continuously running updates on a smartphone close to real time.
4 Algorithms

4.2.1 The state vector

The Matlab implementation which we used to experiment is based on Jia’s paper from 2013 [42]. It maintains a state vector
\[ \hat{x} = [f, c_x, c_y, t_r, t_d, b_g^T, q_c^T, \omega_1^T, \ldots, \omega_M^T]^T \]
containing all parameters to be estimated and the measured gyroscope velocities \( \omega_i \). Later versions of the algorithm [24] omit the rotations \( \omega_i \) from the explicit formulation of the state vector, hence leading to a simpler Jacobian matrix required by the extended Kalman filter.

4.2.2 Prediction update stage

All parameters except gyroscope bias \( b_g \) are constant and can just be copied:
\[ \hat{x}_{k|k-1} = \hat{x}_{k-1|k-1} \]
\( b_g \) is modeled as a random-walk process. Hence, its estimation is updated at each iteration:
\[ \hat{x}_{k|k-1} = \text{update}_{b_g}(\hat{x}_{k|k-1}) \]
The state vector also contains all recorded rotations. Thus, it is expanded with newly captured gyroscope measurements:
\[ \hat{x}_{k|k-1} = \text{addNewGyroMeasurements}(\hat{x}_{k|k-1}) \]

4.2.3 Measurement update stage

As previously mentioned, the algorithm relies on the coplanarity constraint. Each coplanarity constraint value is determined by a group of three matched features. The current Matlab implementation randomly selects as many groups as possible for each pair of adjacent frames. Jia also suggested improvements possibly yielding more accurate results, as discussed in section 2.10.1. The update stage starts by randomly selecting \( N \) groups of matched features, each having a cardinality of three. The resulting coplanarity constraint values then implicitly form the measurements (i.e., they are all expected to be equal to zero) leading to the following \( N \) observations for the \( k \)th filtering iteration
\[ 0 = z_{k,1} = h(\hat{x}_{k|k-1}, u_1, v_k) \]
\[ 0 = z_{k,2} = h(\hat{x}_{k|k-1}, u_2, v_k) \]
\[ \vdots \]
\[ 0 = z_{k,N} = h(\hat{x}_{k|k-1}, u_N, v_k) \]
where \( u_i \) contains all coordinates of group \( i \), \( v_k \) represents the measurement errors and \( h(\ldots) \) computes the value of the coplanarity constraint group defined by its arguments.
4.3 Partial calibration and synchronization using average pixel translation rates

The average pixel translation rate explained in Section 2.11 is a simple yet efficient measurement that suits very well for computing an initial estimation of

\( f \) Focal length.

\( t_d \) Constant delay between recorded timestamps of camera and gyroscope. when ignoring rolling shutter skew, translation and z-axis rotation. This algorithm first computes \( t_d \) using grid search based on a function similarity measurement. Second, it computes an estimation of the focal length using least squares. This algorithm combines the translation rates of both \( x \)- and \( y \)-axis. Hence, equations 2.1 and 2.2 are adapted to:

\[
 rf(t) = \frac{\sum_{m \in M(t)}(m_x - m'_x) + (m_y - m'_y)}{2|M(t)| + (t_i - t_{i-1})} \tag{4.1}
\]

\[
 rg(t, f) = \frac{f \omega_x(t) + \omega_y(t)}{2} \tag{4.2}
\]

Fig 4.2 visualizes average image and gyroscope pixel translation rates before synchronization and calibration. Thus, the figure’s pixel translation rates computed from gyroscope measurement and captured images exhibit a horizontal shift and a vertical scaling factor that do not (yet) match.

\[\begin{array}{c}
\text{Translation rate [px/s]} \\
\end{array}\]

\[\begin{array}{c}
\text{Image } rf(t) \\
\text{Gyroscope } rg(t, f)
\end{array}\]

\( t \) [s]

0 2 4 6 8 10 12 14 16 18 20

**Figure 4.2:** Visualization of average pixel translation rates computed from camera images and gyroscope measurements before synchronization and calibration.

4.3.1 Synchronization using grid search

The algorithm starts by synchronizing the timestamps of image frames and gyroscope measurements using grid search based on a similarity measurement. Given timestamps \( t_1, \ldots, t_n \in T \)
indicating the locations used to compare the average translation rates and candidate shifts \( t_d^1, \ldots, t_d^m \in T_d \) we search for

\[
t_d = \arg \max_{t_d \in T_d} \sum_{t \in T} s \left( r^I(t + t_d), r^g(t, f) \right)
\]

(4.3)

where \( s(a, b) \) is a function establishing a similarity measurement for the values \( a \) and \( b \). We analyzed cross correlation

\[
s(a, b) = a \cdot b
\]

(4.4)

and function similarity

\[
s(a, b) = -\text{abs}(a - b)
\]

(4.5)

as similarity measurements.

Cross correlation has the advantage of working independently of any scaling factor between the compared signals. Further, it favors shifts where as large as possible peaks overlap. Hence, the actual shape of amplitudes may become secondary, as a peak might be able to dominate only based on its amplitude. Alternatively, function similarity puts more emphasis on the signals actual shape if we are able to choose a reasonable initial scaling factor. Fig. 4.3 shows an example how well they measure the overlap between the two curves. The measurements are based on a longer image and a shorter gyroscope signal. The former consists of two peaks, the first one being of a box like shape, while the second one has the shape of a triangle. The gyroscope signal also consists of a triangle shaped peak fully matching the image signal’s second peak. Fig. 4.3b shows the similarity values computed using cross correlation. It returns the highest value for a shift of \( t_d = 100 \). It thus matches the signal originating from the gyroscope to the first peak of the image signal even though the peaks’ shapes do not really match. In contrast, the result shown by Fig. 4.3c, obtained by using function similarity, shows a clear maximum for the image’s second peak.

As shown above, the function similarity puts more emphases on the actual shape of the signal, rather than only the locations and amplitudes of the peaks. Hence we decided to use this measurement for the computation of the timestamp difference \( t_d \). Fig. 4.4 shows image and gyroscope signals after synchronization. The horizontal timestamp shift \( t_d \) has been accounted for, leaving a vertical shift depending on the actual focal length \( f \) to be estimated which is done in the following section.

### 4.3.2 Calibration using least squares

After having computed an estimation of \( t_d \), it remains to compute the focal length \( f \). Equation 4.2 uses the focal length as a scaling factor for the average pixel rates computed from gyroscope measurements. Hence, extracting \( f \) yields

\[
r^g(t, f) = f \cdot \frac{r^g(t)}{\sqrt{\omega_x(t)^2 + \omega_y(t)^2}}
\]

(4.6)
leading directly to a least squares solution for focal length $f$:

$$
\begin{bmatrix}
    r^g(t_1) \\
r^g(t_2) \\
\vdots \\
r^g(t_n)
\end{bmatrix}
\begin{bmatrix}
f
\end{bmatrix}
\begin{bmatrix}
r^f(t_1 + t_d) \\
r^f(t_2 + t_d) \\
\vdots \\
r^f(t_n + t_d)
\end{bmatrix}
$$

This equation states an overdetermined system where $f$ is the only unknown to be estimated. Using least squares is a very popular technique in data fitting (for example to find a trend measurements.

Figure 4.3: Comparing function similarity (Fig. 4.3c) and cross correlation (Fig. 4.3b) similarity measurements.
of some measurements). It computes an optimal solution for $f$ such that the sum of squared residuals is minimized:

$$f = \arg \min_f \sum_{i=1}^{n} (\frac{r_f(t_i + t_d) - r_g(t_i) \cdot f}{\text{residual}})^2$$

Fig. 4.5 shows the aligned pixel translation rates computed from camera images and gyroscope rotations after $f$ has been estimated.

**Figure 4.5:** Visualization of average pixel translation rates computed from camera images and gyroscope measurements after gyroscope-camera synchronization and camera calibration.
4 Algorithms

4.4 Calibration and synchronization using coplanarity constraint and grid search

Calibration and synchronization using average pixel translation rates, as presented in Section 2.11, suits very well for computing an initial estimation of $f$ and $t_d$. However, the estimation is not very accurate as it ignores many important factors such as rolling shutter skew, z-axis rotation and translation. However, the coplanarity constraint accounts for all of these effects. Thus, we decided to use it in combination with grid search to get a more accurate estimations of the following parameters:

- $f$ Focal length.
- $c_x, c_y$ Principal point coordinates.
- $t_d$ Constant delay between recorded timestamps of camera and gyroscope.

We explicitly omit rolling shutter delay $t_r$ as this value is provided by Android’s Camera2 API (see Section 5.2.3). Further, the algorithm could be extended to estimate lens distortion coefficients, as mentioned in Section 4.2.

4.4.1 Grid search for one feature group

A feature group $g_i$ consists of three features $p_i^0 \ldots p_i^2$ and corresponding matches $p_i^0 \ldots p_i^2$ used to compute the coplanarity measure $v^i$. Each feature $p_i^j$ and match $p_i^j$ pair describes one distinct 3D feature from two different poses. First, the algorithm computes normalized direction vectors $\vec{p}$ for each feature $p = (x, y)$ as

$$\vec{p} = \text{norm}\left(\begin{bmatrix} x - c_x \\ y - c_y \\ f \end{bmatrix}\right)$$

using an estimation of focal length $f$ and principal point coordinates $(c_x, c_y)$. Further, it has to interpolate all the intermediate rotations $R_i^0 \ldots R_i^5$ using the feature’s timestamps, an estimation of timestamp delay $t_d$ and the gyroscope measurements. Section 2.3 describes the details of computing such timestamps for rolling shutter cameras. Next, the algorithm computes the coplanarity constraint value as

$$v_i^0 = \det\left( (R_i^0 p_i^0 \times R_i^2 p_i^0) (R_i^1 p_i^1 \times R_i^3 p_i^1) (R_i^2 p_i^2 \times R_i^4 p_i^2) \right)$$

where $v_i^0$ represents how well the used estimations match the ground truth. The smaller $v_i^0$, the better are our estimations. Finally, grid search is performed by repeating the above steps while constantly altering one estimated parameter using different shifts $x \in [-r, \ldots, 0, \ldots, +r]$ where $r$ denotes a predefined search radius. This finally results in coplanarity values $[v_i^{-r}, \ldots, v_i^0, \ldots, v_i^r]$. The best parameter estimation for group $g_i$ is then located at shift $s_i = \arg\min_x v_i^x$

Fig. 4.6 shows the determinant values $v_i^x$ for an exemplary group $g_i$ for different shifts $x$. The
4 Algorithms

![Figure 4.6: Coplanarity constraint determinant values $v_i^x$ of a feature group $g_i$ for different shifts $x \in [-200, ..., 0, ..., 200]$. The optimal shift $s_i$, indicated by the red mark, is located at $x = 26$.](image)

A minimal determinant value is computed at shift $x = 26$. Hence, an initial parameter value of for example 240 for $c_y$ would be adapted to $240 + 26 = 266$. Algorithm 1 further illustrates the above explanations.

**Algorithm 1** One feature group grid search. Gyroscope measurements are assumed to be globally available.

```plaintext
1: procedure groupGridSearch(params, $g$, r, par)
2:   for shifts $s \in [-r, ..., 0, ..., +r]$ do
3:     Shift parameter $par$ in $params$ by $s$ from initial value
4:     Compute feature vectors $\vec{p}_0$ to $\vec{p}_2$ and $\vec{p}_0'$ to $\vec{p}_2'$
5:     Interpolate rotations $R_0$ to $R_5$
6:     Compute coplanarity constraint value $x$
7:     $v_s \leftarrow x$
8:   end for
9:  return $v$
10: end procedure
```

**4.4.2 Building the feature groups**

Even though the features can be combined into groups in a randomized fashion, it might be beneficial to use vertically close and horizontally (relatively) distant features (see Section 2.10.1). We use a simple yet efficient algorithm to build such groups. It proceeds for each frame separately. The following explanations are also illustrated by Algorithm 2. First, the frame is separated horizontally into three equal bins. The detected features are assigned to exactly one of these bins depending on their horizontal position, as shown by Fig. 4.7.
Next, each bin’s features are sorted vertically, hence allowing a line sweep from top to bottom. The line sweep keeps track of currently available features in each bin while traversing. It greedily builds a group consisting of one feature of each bin if their vertical distance is smaller than five percent of the image height.

Algorithm 2 Computes groups consisting of vertically close and relatively distant features.

```plaintext
1: procedure COMPUTEGROUPS(frames)
2:     g ← ∅
3:     for frame ∈ frames do
4:         Define three equal bins separating frame horizontally.
5:         for feature ∈ frame do
6:             Put feature in to correct bin based on horizontal position
7:         end for
8:     Sort bins vertically
9:     while Scan from top to bottom do
10:        if Horizontal distance of currently registered features ≤ 0.05·image height then
11:            Push group to g
12:        end if
13:     end while
14:     end for
15:     return g
16: end procedure
```

4.4.3 Combining results for multiple feature groups

The previous sections illustrated how to build groups and how to perform grid search for one of them. This section shows how we combine the coplanarity constraint values of different groups into one estimation. Algorithm 3 and Fig. 4.8 provide an overview of the steps involved.
Figure 4.8: Combination of coplanarity constraint values from multiple feature groups into one estimation by averaging. The resulting shift of $S = 6$ is indicated by the red marks in the lower right plot.

Each group $g_i$ where $i \in [1, \ldots, n]$ results in a set of coplanarity constraint values $v^i = [v^i_{-r, \ldots, v^i_0, \ldots, v^i_r}]$. We combine them by computing the mean value for each shift. Hence, the combined estimation is computed as

$$V_x = \frac{1}{n} \sum_{i=0}^{n} v^i_x$$

where $s \in [-r, \ldots, 0, \ldots, +r]$. Finally, we proceed as with a single group by taking

$$S = \arg \min_x V_x$$

as optimal shift.

Section 6.3 discusses the problem of feature detection and matching defects introducing errors in our estimation. Local maxima in the search intervals of certain groups or groups dominating the computed average may lead to wrong parameter estimations. Faced with such cases, just combining all coplanarity constraint values $v_i$ may degrade our combined estimation. To avoid such problems, we exclude all these groups before computing the final average. Groups with a maximum in the search interval are removed. Further, we also reject all groups with very small or large coplanarity constraint values, as described by Algorithm 3.
Algorithm 3 Combines the coplanarity constraints' values of different groups. It computes the location of the minimal combined coplanarity value and adapts par based on this location.

1: procedure ADAPT_PARAMS(params, par, v = [v₁, ..., vₙ])
2:   minThresh ← threshold for smallest 1% in v
3:   maxThresh ← threshold for largest 1% in v
4:   for each vᵢ ∈ v do
5:       if vᵢ has local maximum
6:           and min(vᵢ) < minThresh
7:               and max(vᵢ) > maxThresh then
8:       v ← v \ vᵢ
9:   end if
10: end for
11: n ← |v|
12: for each computed shifts s do
13:   Vₛ ← \[ \frac{1}{n} \sum_{i=0}^{n} vᵢ \] s
14: end for
15: bestShift ← arg minₜ Vₛ
16: Adapt parameter par in params by adding bestShift
17: end procedure

4.4.4 Alternating parameter grid search

Finally, the coplanarity constraint based grid search may be used to estimate timestamp difference tₜ, focal length f and principal point coordinates (cₓ, cᵧ). To do so, we perform the search alternating on the different parameters for a given number of iterations.

Algorithm 4 Algorithm performing coplanarity constraint based grid search for different parameters.

1: procedure COMPUTE_PARAMS(params, frames, r)
2:   g ← COMPUTEGROUPS(frames)
3:   for Some number of iterations do
4:       for each p ∈ tₜ, f, cₓ, cᵧ do
5:           for each gᵢ ∈ g do
6:             vᵢ ← GROUPGRIDSEARCH(params, g, r, p)
7:         end for
8:       ADAPT_PARAMS(params, p, v)
9:   end for
10: end procedure
11: return params
12: end procedure
4.4.5 Parameter initialization

Accurate parameter initialization is of concern as it directly affects the required search radius and hence the algorithm’s performance. Thus, we intend to keep our search intervals as small as possible. The Android Camera2 API provides us with each frame’s rolling shutter skew $t_r$, hence there is no need to even estimate it. Further, the camera’s principal point $(c_x, c_y)$ is very likely to be near the center of the frame. Hence, the only two parameters for which we do not have trivial defaults at hand are the focal length $f$ and timestamp delay $t_d$. Fortunately, we can use the proposed algorithm based on pixel translation rates (see Section 4.3) in order to get a sufficiently accurate initial estimation of these parameters quite efficiently.
5 Implementation

5.1 Overview

The goal of this thesis is to provide a fast yet robust sensor synchronization and camera calibration algorithm, capable of running directly on a smartphone. We realized our proposed algorithms of Section 4.3 and Section 4.4 as Matlab implementations for the ease of experimentation and visualization. Further, we provide a selection of useful Matlab functions. There are helper tools for visualization and modification of recorded camera images and sensor measurements. We also facilitate generating synthetic feature matches based on given camera rotations which helps to develop, test and evaluate camera and sensor based algorithms.

This thesis also provides platform-independent C++ library implementations of the algorithms described in Section 4.3 and Section 4.4. Additionally, we implemented helper libraries offering flexible and extensible buffering data structures and facilities for debugging, profiling and visualization. Reusing libraries in different projects while maintaining various dependencies and build descriptions potentially causes large overhead. Hence, we developed a CMake based build framework facilitating the compilation of (platform-independent) executables for testing and Android native code as proof of concept. Further, we present a framework for developing camera and sensor based Android applications which is directly integrated into our CMake build system. Based on this framework, we developed different Android applications for recording data, performing camera calibration, synchronizing sensors and deblurring images. Our implementations make extensive use of the portable OpenCV library.

Android is an open-source operating system developed for mobile devices. As it currently has by far the largest market share, it is very attractive for application development. Android applications are generally developed in Java. Nevertheless, it also facilitates the deployment of native C or C++ code, for example for performance critical applications. Android offers a Java Native Interface (JNI) which allows to load native C or C++ libraries from Java contexts. It further facilitates communication from Java to the native code and the other way around. C++ code intended for Android is developed using Android’s Native Development Kit (NDK). It ships with all necessary tools, including a customized cross-compiler, required to build C or C++ code as libraries for the target architecture. Due to the popularity of Android and the existing facilities to deploy (fast) native code, we targeted this operating system for our project.

This thesis makes extensive use of Android’s camera and sensor APIs. The camera API recently changed dramatically, offering a new (more complex and versatile) interface for camera configuration. Hence, we will present this new API in the following subsection.
5.2 Android Camera2 API

Android’s new Camera2 API recently released in API level 21 (i.e., Lollipop) introduces a new Hardware Abstraction Layer (HAL) v3\(^1\) enabling more versatile but also more complex camera functionality. The old Camera API was designed as a black box only offering access to high-level controls, such as registering image callbacks or running it in preview, video recording or still capture mode. The new API aims to be more generic, thus offering a stable interface to different camera types supporting a broad range of capabilities. Fig. 5.1 shows a high-level overview of the new camera API’s core operation model.

Figure 5.1: Overview showing the new Camera2 API’s core operation model (reprinted from [43]).

5.2.1 Camera configuration

While the old camera was globally configured, the new API maintains a queue of pending CaptureRequests\(^2\), each one having its own parameters. This way, the user has very fine

grained control over each request’s parameters including e.g., exposure time, focus distance, lens aperture size and so on. It is the developer’s responsibility to keep the pending request queue filled. Fig. 5.1 and Fig. 5.2 indicate the API’s building blocks including the request queue.

### 5.2.2 Data output

While the old system supported multiple ways of extracting images, the new one unifies it using output streams\(^3\) as pipelines for captured images. These streams may have different image formats, resolutions or buffer depths. Further, each of them has an interface called a **Surface**\(^4\) which is registered at the camera instance providing empty buffers to be populated by data from image capturings. The **Camera2** API allows to register multiple such streams at once, enabling to send an image automatically and asynchronously to different destinations as indicated by Fig. 5.2. Hence, different versions of a captured image may be transferred simultaneously to the GPU, actual views or an **ImageReader**\(^5\).

![Diagram of input and output streams](image-url)

**Figure 5.2:** Overview showing input and multiple output streams used by cameras supporting HAL v3 (reprinted from [44]).

---


5 Implementation

5.2.3 Metadata output

The new API also facilitates access to various camera parameters, an important feature e.g., for image post-processing. As indicated by Fig. 5.1, each capture calls the `onCaptureComplete` callback exactly once. This call comes with a `TotalCaptureResult` argument containing the camera metadata, e.g., rolling shutter skew, sensor exposure time and many more. The availability of certain parameters is dependent on the actual hardware support. As images are delivered asynchronously (see Section 5.2.2 for more details) they may arrive substantially later than the corresponding metadata. Therefore, synchronization may be necessary to map the metadata to the corresponding image.

5.3 ETHCamera Android application

We needed an Android application capable of capturing a stream of images and sensor measurements. The `ETHCamera` application, developed as part of this thesis, is built as a framework performing this task. It allows to retrieve image and sensor measurement streams and forwards them to different back-ends (so-called clients). This setup allows to reuse the necessary application logic for the camera and the sensors for different clients. The `CVGCamera` Android application (by Petri Tanskanen and Lorenz Meier, extended by Gábor Sörös, all PhD students at ETH) is the basis of our ETHCamera application. It is build for recording a series of camera images and sensor measurements to the device’s storage. The camera is managed using the Camera API offered by Android. This API has been superseded by the new Camera2 API offering extensive image metadata and more fine grained control of the camera. Due to the need of integrating an Android application into our framework and having access to Android’s advanced Camera2 API features, we decided to refactor and adapt the original CVGCamera application. The current implementation consists of a Java and a native (i.e., C++) part. The former handles the GUI, user inputs and the camera logic while the latter is responsible for processing captured images and sensor measurements. Further, the native part manages clients which are interchangeable and contain the processing logic handling the captured data. Fig. 5.3 and Fig. 5.4 show the major building blocks involved in setting up and running the preview and capturing modes of the application. The following listing explains the involved components in more detail:

MainActivity

The main activity which is started at the very beginning containing the preview `SurfaceView`.

GlSurfaceView

A dedicated drawing surface embedded in the MainActivity. It is responsible for rendering the device’s GPU contents.

---

Controller
A class instantiated by the MainActivity which is responsible for the application’s setup and handling user requests. It manages instantiation of the camera and different parts contained in the native code. Further, it coordinates the communication between different Java callbacks and the native back-end.

SurfaceTextureRenderer
An instance of a GLSurfaceView.Renderer\(^9\) responsible for making OpenGL calls to render frames. This renderer is registered on the GlSurfaceView which in turn triggers the renderer. OpenGL receives the images using a SurfaceTexture\(^{10}\). This special type in Android allows to render images as textures that are coming from the camera and are stored in special memory that is directly written by the camera and read by the GPU (i.e., no copy needed).

CameraFactory
A factory class responsible for instantiating an appropriate camera instance depending on the device’s hardware support. Given a device running with Android Lollipop (i.e., API level 21) or above, the factory instantiates a camera utilizing the more versatile Camera2\(^{11}\) API. Otherwise, it will create a camera instance using the ordinary Camera\(^{12}\) API.

AbstractCamera
A common interface for our Camera1 and Camera2 implementations instantiated by the CameraFactory.

Camera1
The camera implementation for the traditional (and recently deprecated) Camera API.

Camera2
The camera implementation for the recently released Camera2 API.

Jni
Acts as an interface between Java and native (i.e., C++) code. Coordinates the communication between the shared library and the Java application. Hence, this building block is defined in Jni.java and jni.cpp.

FrameworkState
Implements the interface described in Section 5.5.2 and Fig. 5.7 which is used by the client code.

Gui
Code responsible for creating and maintaining an OpenGL state including shaders and textures. Further, it does the rendering upon arrival of a new frame.

Sensors
Logic responsible for handling sensor measurements, e.g., from gyroscope or accelerometer.

---


\(^12\) See http://developer.android.com/reference/android/hardware/Camera.html.
This code spawns a new thread constantly retrieving new sensor measurements. Depending on a boolean flag, the measurements are either directly discarded or pushed to a registered Client.

**Client**

The code provided by the client responsible for handling the captured images and sensor measurements.

**UserInput**

User input triggered e.g., by a button press. The **MainActivity** forwards these to the controller.

### 5.3.1 Application initialization and camera preview

Upon start of the application, a new camera instance is initialized. Depending on the API level supported by the device, the application either makes use of the advanced features offered by the Camera2 API or it uses the deprecated Camera API. After the application has been initialized, it shows a preview of the captured images. These images are transferred from the camera to the GPU where the final image is rendered before being displayed by a GlSurfaceView. Fig. 5.3 shows the process of application setup and displaying a camera preview in more detail. The following listing provides detailed information for each step involved:

1. The **MainActivity** code instantiates a **Controller** object to which it passes a GlSurfaceView for initialization.

2. Next, the **Controller** instance creates an object of type **SurfaceTextureRenderer** which it registers on the GlSurfaceView.

3. The **onSurfaceCreated()**\(^{13}\) callback is invoked when rendering starts. It is intended to set up necessary OpenGL resources.

4. The callback is forwarded to our native code, where the routines in **Gui.cpp** initialize the OpenGL state including a texture.

5. After all the required OpenGL resources have been initialized, the texture’s ID is returned.

6. As the texture is required later on by the camera, its ID is propagated to the Java code.

7. The texture ID is used for instantiating a **SurfaceTexture**\(^{14}\) object which captures frames from an image stream as an OpenGL texture.

8. The **SurfaceTexture** is returned to the **Controller** via a registered callback.


Next the **Controller** instantiates a camera using the **SurfaceTexture** object as destination for the stream of captured images. Further, it registers a listener on the camera instance returned by the **CameraFactory** which is called upon arrival of a new image. Finally, it starts the camera in preview mode.
Our application supports the new Camera2 API as well as the legacy Camera API. Depending on the current device’s API level, the CameraFactory returns either a Camera2 instance for API level 21 or above or a Camera1 instance otherwise. The client (i.e., the Controller) receives an instance of type AbstractCamera which abstracts the actual camera type used.

As soon as a new image has been captured, the surfaceTextureReady callback gets invoked, indicating a new image available for the surface texture.

Next, the Controller reacts to the callback invocation by executing requestRender() on the GLSurfaceView. The surface view has been initialized using the RENDERMODE_WHEN_DIRTY flag. Hence, the SurfaceTextureRenderer does the rendering only after requestRender() has been called.

The renderer is triggered by invoking onDrawFrame() for rendering a new image. The call is forwarded to the native code.

The onDrawFrame() routine contained in Gui.cpp first calls updateTexImage() before rendering the current frame. If the module is active, this code also displays a blinking read circle.

First, a valid OpenGL state is loaded before propagating the updateTexImage() call back to the Java code.

Calling updateTexImage() on the SurfaceTexture updates the texture to the most recent frame of the image stream. This may only be called while the OpenGL ES context that owns the texture is current on the calling thread.

5.3.2 Initializing the client module and capturing image and sensor data

Users may start or stop the Client module contained in the native code. When started, sensor measurements and camera images are delivered to the registered Client object until the user stops the module again. Further, the native Client may request the values of properties of the camera. If the current device does not support the Camera2 API, then the Client will get some default value for the requested property. Fig. 5.4 shows the processes of starting the module and delivering sensor measurements and camera images. The following listing provides detailed information for each step involved:

S₁ Some user input (e.g., a button press) requests the start of the module from the controller.

---

Overview of the steps necessary for initializing the client module and capturing image and sensor data.

The controller instructs the camera instance to fix its parameters. I.e., the camera fixes its exposure time and focus distance. Having constant values for these parameters makes the analysis easier.

**Figure 5.4:** Basic building blocks of ETHCamera capturing mode.
Next, the framework is prepared for a new run. This includes determining a new output directory and initializing compatibility helpers. After notifying the **Client** code of the activation, as indicated by step \( S_4 \), a flag is set indicating the module’s activation.

\( S_4 \) The client code gets notified of the start.

\( S_5 \) The sensor code runs an own thread spinning on the sensors and constantly extracting data. Depending on the flag set by step \( S_3 \) the measurements are directly discarded or pushed to the **Client** using its `acceptSensorData()` method.

\( C_1 \) As soon as the camera captured a new frame, it returns it to the **Controller** by calling the registered `frameReady()` callback.

\( C_2 \) The controller forwards the frame’s data to the native library.

\( C_3 \) If the module has been started (i.e., the flag has been set in step \( S_3 \)) then the measurements are pushed to the registered **Client** using its `acceptFrameData()` method.

\( C_4 \) A client may optionally request camera parameters for the most recent frame it got delivered.

\( C_5 \) This request for parameters is forwarded to the camera. Depending on the API support, either the actual value or some default is returned.

### 5.3.3 Other features

The ETHCamera application also includes features not mentioned in the previous sections. The following listing enumerates these for the sake of completeness:

- **Output folder scanning**
  Modern Android devices use the *media transfer protocol* (MTP) to offer access to their storage via USB. This protocol superseded the *USB mass storage device class* (USB MSC) protocol used by older devices. In practice, MTP looks a lot like USB MSC to users. After plugging the device into a computer using USB, a file system browser, e.g., Explorer (Windows) or Files (Ubuntu), opens and enables the user to browse the device’s storage. Besides this similarity, the protocols differ a lot on the implementation level. USB MSC exposes a raw file system to the computer. Windows does only support a very limited set of file-systems. Hence, the storage has to be formatted as FAT which is old and rather slow. Further, exposing the raw file system is only possible if the access is exclusive. In contrast, MTP basically exposes a list of files and the computer actively requests them for accessing the contents. Besides abstracting the underlying file system format, the protocol also enables simultaneous access by the device and the connected computer. MTP requires applications to actively register newly created files. Otherwise, the computer is not guaranteed to see the new files. Certain Android devices even require a complete restart before new files actually become visible to the computer. Hence, using Android’s
MediaScannerConnection\textsuperscript{19}, our application rescans the output folder provided by the framework after a run has been terminated. After scanning, new files created in the output folder are immediately visible to the computer.

- Fixing camera parameters for multiple runs
  The Application allows to fix camera parameters such as the focus distance and exposure time globally. Using this feature allows to compare different runs.

- Compatibility helpers
  Different devices may require different camera settings or specific handling of the output images. E.g., Google’s Nexus 6 uses a different image row stride on the output data than the Nexus 5. Further, there may exist bugs in camera drivers. As for example, the \texttt{FOCUS\_MODE\_FIXED}\textsuperscript{20} parameter does not work properly on Samsung S5 devices. Thus, the application includes a \texttt{Config.java} file allowing to address such issues in a device specific manner.

5.4 CMake building and testing framework

\textit{CMake} \textsuperscript{[45]} is a cross-platform open-source system maintaining the software build process in a compiler and operating system independent manner. Using \textit{CMake}, a user only has to maintain a \texttt{CMakeLists.txt} file describing the project’s contents and dependencies. This single configuration file allows CMake to generate specific project files for a variety of integrated development environments and operating systems such as Eclipse, QtCreator or Visual Studio. Moreover, it is also supports generating build descriptions for command line tools, e.g., Unix Make, NMake, MinGW, or Ninja. Unix Make or Eclipse makefile projects are suitable for Android C++ development. Such projects are configured to include Android’s \textit{native development kit} (NDK) which then compiles the code for a predefined device architecture. Besides generating build descriptions, \textit{CMake} also facilitates dependency management, unit tests and out-of-source builds.

\begin{figure}[h]
\centering
\includegraphics[width=0.3\textwidth]{cmake.png}
\caption{CMake’s logo.}
\end{figure}

\textsuperscript{19} Documentation: http://developer.android.com/reference/android/media/MediaScannerConnection.html.
\textsuperscript{20} Documentation: http://developer.android.com/reference/android/hardware/Camera.Parameters.html#FOCUS\_MODE\_FIXED.
5.4.1 Single build description

Consider a C++ project intended to be built on Windows, Linux and Android. Fig. 5.6a illustrates this situation without using CMake. It requires having e.g., a Visual Studio project for Windows, a Make file or an Eclipse project for Linux and an Android NDK project. Each of these projects or build files has to be maintained individually, which is quite time consuming and tedious to avoid inconsistencies. Using CMake, as shown in Fig. 5.6b, simplifies this task a lot since we only have to maintain the build configuration once. CMake then generates build files for the Android compiler, Make files for GCC or a variety of different IDE projects and Visual Studio projects.
5 Implementation

(a) Minimum cross-platform build setup not using CMake. Each build description has to be maintained separately.

(b) Cross-platform/compiler setup using CMake. The user only maintains the CMake build configuration, while CMake automatically manages generated projects.

(c) Legend for Fig. 5.6a and Fig. 5.6b.

Figure 5.6: OS/compiler specific build setups versus taking advantage of CMake’s build facilities.

5.4.2 Other features

Besides the single build description, using CMake has quite a few other important features:
5 Implementation

- **Dependency management**
  It allows us to specify our dependencies once and reuse them among different projects. Hence, we avoid duplicating such dependency descriptions.

- **Out-of-source builds**
  It facilitates building projects outside of the source tree. This feature is quite handy, as it helps avoiding to unnecessarily clutter the source directory tree.

- **Cross compiling**
  Using so called toolchain files, CMake is able to perform cross compilation for different target architectures on a single system.

- **Unit testing**
  CMake seamlessly integrates with Google’s testing framework [46].

5.5 Our CMake-based framework

The framework developed in this thesis consists of the following four files:

- **globalConfig.cmake**
  Contains platform independent settings, e.g., debug or release build type or whether to enable testing.

- **localConfig.cmake**
  Contains settings used by local builds (i.e., not Android builds). It configures local dependencies, e.g., OpenCV, Boost or Qt.

- **androidConfig.cmake**
  Configures settings for Android builds like the location of the toolchain file, target API level, where to put the resulting shared library and so on.

- **globals.cmake**
  This file contains all the logic used for building the projects. It defines how library, executable or Android module projects have to be assembled. Further, it also defines how OpenCV, Boost or other library dependencies have to be dealt with. This file has to be included in every project description.

Starting from the top-level directory of each project (i.e., the one containing CMakeLists.txt, as described in Section 8.1), the framework recursively searches all child directories for source and header files and includes them automatically into the build. Hence, the user does not have to declare all the files contained in the project separately. We support three different build types:

- **Library**
  This build type specifies static libraries to be used with executables or Android modules.

- **Executable**
  Builds an executable for the local architecture.
5 Implementation

- **Android module**
  Cross compiles a shared library for the target architecture specified by `androidConfig.cmake`. The library is copied to `OUTPUT_PATH` also defined in the aforementioned configuration file.

### 5.5.1 Library and executable projects

Library projects are used to write code that gets compiled into a static library. This code is accessible to other projects by declaring dependencies (see Section 5.5.3). Section 8.3 shows how to define and structure library projects. Executable projects are used to build executables for the current system’s architecture. Section 8.4 shows how to define and structure such projects.

### 5.5.2 ETHCamera projects

Our main project is an Android application that handles camera and sensor input. All the previously described projects are integrated into this Android application. The CMake based framework developed by this thesis integrates the projects as native code using the Android NDK [47]. Directly debugging native Android code on the target device can become quite difficult and always being required to deploy the whole application during development is time consuming. Hence, we decided to add a simulator (Section 5.6.3 provides more details concerning the simulator’s implementation), enabling to run the code locally on previously recorded images and sensor measurements. To do so, we defined an interface, as illustrated by Fig. 5.7, between client projects and data-producing back-ends (i.e., the framework). The client code is responsible for providing an object implementing a predefined interface which then receives images and sensor measurements from the framework.

Depending on the build type (See Section 8.8.), the code is either compiled to an executable for the local target architecture or a shared library for the Android device’s architecture. The executable takes the path of a directory containing recorded camera images and gyroscope measurements as argument and internally uses code that emulates (see Section 5.6.3) the capturing device. The shared library combines the module’s code with the native part of the ETHCamera application. Further, the binary file is automatically deployed to the ETHCamera Java project (see Section 5.3).

### 5.5.3 Dependency management

Our framework distinguishes two kinds of dependencies. There are external and internal ones. External dependencies are C++ libraries not defined by ourselves, e.g., OpenCV [48], Boost [49] or Qt [50]. OpenCV is a computer vision library, Boost offers a collection of cross-platform scientific helper functions and Qt is a cross-platform framework for creating graphical user interfaces. These libraries are used by different projects and their respective CMake configurations may be a bit more involved. Hence, `globals.cmake` provides macros offering a convenient way of specifying such dependencies (see Section 8.6). Internal dependencies cover
our own library projects. Using our flat project hierarchy\textsuperscript{21}, we only have to add a one-liner in a project’s \emph{CMakeLists.txt} to specify an internal dependency (see Section 8.6).

\section{Helper libraries}

While developing C++ implementations of our algorithm, we repeatedly encountered situations required similar solutions. Buffering data structures, debugging or profiling helpers, facilities to access file system resources are examples of code which is re-used a lot. Hence, using our CMake framework as described in Section 5.5, we built helper libraries hosting such shared code that can be conveniently included into other projects. Further, using our CMake framework, the libraries also ship with tests for its core functionality. We offer developed different such libraries, either only hosting completely platform independent code (for example also for Android projects) or hosting code which is intended for local development (for example facilitating filesystem access).

\textsuperscript{21} Described in Section 8.1.
5 Implementation

5.6.1 The *helpers* library

This library is intended to be used for both Android and executable projects. It contains implementations of buffer solutions and other data structures, debugging and profiling code, compatibility helpers and visualization tools using OpenCV. The following sections will provide an overview over the most important parts. In general, for handling measurements from any sensor that can be ordered by timestamps, and we want to search for data in specific time intervals. Further, the buffer implementation should still be extensible and versatile. Therefore, we designed an appropriate buffering structure as presented in the following section.

**Buffer implementations**

Our buffers use a tick based model, as visualized by Fig. 5.8. Each buffer slot is associated with a tick, starting at zero. Clients may request new input storage of a certain size, extending the active portion of the buffer by the respective number of ticks. It is preferable to release unused slots, as the buffer may exhibit limited capacity or there may be memory restrictions imposed by the target platform. Hence, our buffer implementations enable a client to indicate up to which tick the slots should be released. The library divides buffering related responsibilities into two parts. There is a template `BufferInterface<T>` interface. Classes implementing this interface (e.g., `VectorBuffer<T>` or `RingBuffer<T>`) are primarily responsible for managing the memory used for storing values. Further, it offers functionality to retrieve values of consecutive tick intervals.

The remaining portion of buffer logic is located in the `BaseBufferIntervals<T>` template. It offers an extensible interface for accessing and modifying buffer contents. Objects of this type are passed to a buffer when requesting access to values of some tick range. The buffer then populates the `intervals` field, an array of `begin` and `end` pointers describing the locations of the contents. Using such a representation has the advantage of serving for a wide variety of buffer types while not requiring to provide specific implementations of the `BaseBufferIntervals<T>` template. In combination with the generic nature of `VectorBuffer<T>` and `BaseBufferIntervals<T>` clients can add new functionality to the buffers without actually changing any buffer implementations. Further, all the previously existing features are directly available if `BaseBufferIntervals<T>` is used as base class. For

![Figure 5.8: Tick based buffer model. Each buffer slot is identified by a corresponding tick. The buffer maintains three consecutive intervals of ticks. The released interval contain ticks for which the values have been freed and thus are not accessible anymore. Active values are available and can be queried using the corresponding tick. The remaining ticks belong to free values that still can be populated by the client.](image-url)
example, Fig. 5.9 shows TimestampedBufferInterval<T : TimestampedContainer> which adds functionality to search for timestamp ranges if the underlying buffer stores items with timestamps.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{buffering_implementation.png}
\caption{Illustration of the buffering implementation offered by our \textit{helpers} library.}
\end{figure}

BufferInterface<T> in Fig. 5.9 formulates the interface each concrete buffer has to offer. The most important methods are summarized by the following listing:

\begin{verbatim}
...::getInputStorage(uint32_t ticks, MutableType &intervals)
Requests new buffer storage. The buffer's maximum tick is increased by ticks. The client
\end{verbatim}
may access the newly allocated storage using the `intervals` object that is initialized by the buffer.

```cpp
...::getValues(uint64_t firstTick, uint32_t tickCount, ConstType &intervals)
Requests existing storage for read-only access. This method populates `intervals` by
`tickCount` elements starting at tick `firstTick`.

...::release(uint32_t ticks)
Releases the least recent `ticks` ticks.

...::releaseUntil(uint64_t tick)
Releases all data having a tick smaller or equal `tick`.
```

The interface also offers some other methods for convenience which are declared in `Buffer-
Interface.hpp`. The concrete `VectorBuffer<T>` implementation uses a `std::vector<T>` as
backing buffer whereas the `RingBuffer<T>` implements a ring-buffer data structure. While
`VectorBuffer<T>` does not actually physically release any data, it is important to make sure to
release unused elements in `RingBuffer<T>`. The former buffer may theoretically grow without
any limit (besides the device’s physical memory). The latter one has only limited size and will
throw an exception if it runs out of space.

`BaseBufferIntervals<T>` in Fig. 5.9 describes the data structure identifying the location of
actual data stored inside buffers. The most important methods are summarized by the following
listing:

```cpp
...::MutableType
A typedef describing a mutable version of this object. It is used for read/write access of
buffered data.

...::ConstType
A typedef describing a constant version of this object. It is used for read-only access of
buffered data.

::size() : uint64_t
The number of contained buffered objects.

...::getSubset(uint32_t start, ticks, BaseBufferIntervals<T> &target)
Retrieves a buffered object interval starting at `start` ticks after the current start and
having a length of `ticks` ticks.

...::duplicateContent(BaseBufferIntervals<T> &target)
Copies the current data to a buffer managed by `target`. Further, `target` is initialized
using the copied data. Hence, clients get a mutable version of the buffered data.

...::operator[](uint64_t index) : T &
Used to index into the values currently represented. 0 refers to the very first value
contained in this interval.
```
...::begin() : Iterator
Returns a forward iterator\textsuperscript{22} pointing to the beginning of the current interval.

...::end() : Iterator
Returns a forward iterator pointing to the end of the current interval. The combination with ...::begin() : Iterator also enables range based for loops as define by the C++11 standard\textsuperscript{23} using our interval representation.

The TimestampedBufferInterval$<$T : TimestampedContainer$>$ is a specialized interval descriptor inheriting from BaseBufferIntervals$<$T$>$. It may be used for every buffer implementation using a type inheriting from TimestampedContainer as template argument. This container encapsulates a \texttt{uint64\_t} timestamp and data T such that the TimestampedBufferInterval can apply specialized operations on the data. It adds a \texttt{getBoundingValues\(\texttt{uint64\_t}\ \text{lowerTimestamp},\ \text{uint64\_t}\ \text{upperTimestamp},\ \text{ConstType} &\text{intervals}\)} method returning an interval based on a timestamp range using binary search.

The library already defines specific buffers in \texttt{SensorDataBuffer.hpp} and \texttt{FrameDataBuffer.hpp} tailored to our captured sensor measurements and camera images. The FrameDataBuffer stores images with timestamp, while the SensorDataBuffer stores gyroscope and accelerometer measurements with timestamps.

\textbf{Data visualization}

Data visualization is very important for debugging code. Hence, developers need to have a flexible yet easy to use tool at hand. \texttt{GraphUtils.hpp} and \texttt{GraphUtils.cpp} contain code originally written by Shervin Emami \textsuperscript{51} for displaying \texttt{std::vector<double>} data as graphs. We adapted the code for our buffer implementation described in Section 5.6.1. Fig. 5.10 shows

![Figure 5.10: Pixel translation rates visualized by our helpers library.](image)

\textsuperscript{22} Documentation: \url{http://www.cplusplus.com/reference/iterator/}.
\textsuperscript{23} See \url{http://en.cppreference.com/w/cpp/language/range-for}.
a screenshot of visualized pixel translation rates. The tool runs cross-platform as it displays windows using OpenCV [48] which supports Windows, Linux, Mac OS, iOS and Android.

**Profiling**

Being able to efficiently profile our code is crucial in order to detect performance bottlenecks. Hence, `Profiling.hpp` facilitates profiling by providing macros to measure execution times of specific parts of the code, each identified by a name. Further, the different results are structured

```cpp
#include "helpers/Profiling.hpp"

#include <thread>
#include <random>
#include <chrono>

uint32_t randWait(uint32_t maxWait)
{
    uint32_t r = rand() % maxWait;
    std::this_thread::sleep_for(std::chrono::milliseconds(r));
    return r;
}

int main(int argc, char **argv)
{
    assert(argc == 2); srand(time(NULL));

    for (int iteration = 0; iteration < 100; ++iteration) {
        uint32_t returnedTicks;
        PROFILE_TIME("demoGroup", "longWait",
                     returnedTicks = randWait(700);
        PROFILE_COUNT("demoGroup", "longWaitTime", returnedTicks);

        PROFILE_TIME("demoGroup", "shortWait",
                     returnedTicks = randWait(300);
        PROFILE_COUNT("demoGroup", "shortWaitTime", returnedTicks);
    }

    PROFILE_WRITE_MATLAB_OUTPUT(argv[1]);
}
```

**Figure 5.11**: A small example demonstrating the simple usage of our C++ profiling framework. Wrapping a piece of code with the `PROFILE_TIME` macro measures and stores the respective execution time. Using the `PROFILE_COUNTS` macro allows to store values. Finally, `PROFILE_WRITE_MATLAB_OUTPUT(argv[1])` is called to generate a Matlab script in the directory specified by `argv[1]` containing the measured execution times and the stored values.
5 Implementation

into groups. Additionally, the library may be used to keep track of counts such as for example sizes of arrays. Fig. 5.11 shows a simple code example. To profile a specific piece of code, it is sufficient to wrap it with the `PROFILE_TIME` macro. The `PROFILE_COUNT` macro is used to push a count to the profiling framework. Upon finishing the profiling, `PROFILE_WRITE_MATLAB_OUTPUT` is called which writes a Matlab script containing the stored execution durations and counts to directory. Calling this script using Matlab returns the stored values. Besides retrieving the values, the script also offers facilities to directly inspect them. The user may request the script to visualize the execution times and counts as shown by Fig 5.12.

![Execution times of demoGroup](image1)

![Counts of demoGroup](image2)

(a) Measured execution times.
(b) Stored counts.

**Figure 5.12:** Visualization of profiling data produced by the Matlab script generated by our profiling framework. It shows a graph for the measured execution times (see Fig. 5.12b) and for the stored counts (see Fig. 5.12b).

### 5.6.2 The `localHelpers` library

This library is intended to be used with executable projects only. It contains iterator implementations to read captured image and sensor measurements from the file system. Further, it contains code for offline simulating previously captured and recorded data when no real camera is available.

**Iterators**

These classes are used for reading captured images and sensor measurements stored on the local file system. Our data recording Android application stores image frames and sensor readings such as gyroscope and accelerometer measurements to specific files. By providing the iterators with a path holding the respective files, the iterators return the recorded measurements to clients.
Axes data iterator

This type of iterator is used for reading rotation or acceleration data captured for the three axis of the smartphone. All the iterators implement a bool getNext(double& xAxisData, double& yAxisData, double& zAxisData, uint64_t *timestamp) method, as defined by \textit{AxisDataIteratorInterface}. It returns a new set of axes measurements and corresponding timestamp in the same order as the they were captured. Once the returned boolean equals to false, all of the measurements has been delivered and the iterator has reached the end. Fig. 5.13 shows an overview of the classes used for reading axes measurements. The following listing explains them in more detail:

\textbf{AxesDataIteratorInterface}

Defines the interface which the clients of iterators use. Specifically it declares the bool getNext(double& xAxisData, double& yAxisData, double& zAxisData, uint64_t *timestamp) method returning the actual measurements.

\textbf{AxesDataIterator}

Implements the \textit{AxesDataIteratorInterface} by returning measurements which it reads from a file in each iteration. As both the gyroscope and the accelerometer data are exported using the same format, this implementation suits for both kinds of measurements.

\textbf{GyroscopeIterator}

This class selects the file containing the gyroscope measurements while inheriting the actual iterator implementation from \textit{AxesDataIterator}. 

\textbf{AccelerometerIterator}

This class selects the file containing the accelerometer measurements while inheriting the actual iterator implementation from \textit{AxesDataIterator}.
5 Implementation

AccelerometerIterator

This class selects the file containing the accelerometer measurements while inheriting the actual iterator implementation from AxesDataIterator.

CachedAxesDataIterator

Given a scenario where access latency matters, e.g., our ETHCamera simulator (see Section 5.6.3), the data has to reside in memory. Hence, this iterator reads the all measurements from disk into memory upon initialization. Using a strategy pattern enables this iterator to work for different underlying iterator implementations.

Frame data iterator

This type of iterator is used for reading images captured by a any camera. All the frame iterators implement a bool getNext(cv::Mat &image, uint64_t *timestamp) method as defined by FrameIteratorInterface. This method returns a new set consisting of an image and corresponding timestamp in the same order as they were captured. Once the returned boolean equals to false, all of the data has been returned and the iterator has reached the end. High resolution images easily require a few megabytes of storage. Hence, unnecessarily copying images potentially introduces a non-negligible performance penalty. When working with captured images, we have to be careful to use references whenever possible as this avoids such copies. Fortunately, OpenCV’s cv::Mat object already handles the image data internally as a reference (quite similar to smart pointers\textsuperscript{24}). Thus, copying such an object does not duplicate the whole image content. OpenCV facilitates development a lot, as cv::Mat may be safely passed to or returned from functions as value. Fig. 5.14 shows an overview over the classes used for reading image data. The following listing explains them in more detail:

\textsuperscript{24} See http://en.cppreference.com/w/cpp/memory/shared_ptr for more details on smart pointers.

---

Figure 5.14: Visualization of the hierarchy of image iterators.
5 Implementation

FrameIteratorInterface
Defines the interface which the clients of iterators use. Specifically it declares the `bool getNext(cv::Mat &image, uint64_t *timestamp)` method returning the actual images.

ImageIterator
This iterator just enumerates all the images contained in a given directory in alphabetical order.

FrameIterator
Returns data captured by the ETHCamera app by combining the images from returned from ImageIterator with the corresponding timestamps.

CachedFrameIterator
Given a scenario where access latency matters, e.g., our ETHCamera simulator (see Section 5.6.3), the data has to reside in memory. Hence, this iterator reads all images from disk into memory upon initialization. Using a strategy pattern enables this iterator to work for different underlying iterator implementations.

5.6.3 ETHCamera simulator

This helper is used to offline simulate previously recorded images and sensor measurements captured by our ETHCamera Android application. It does so by reading stored image frames and gyroscope and accelerometer measurements from disk and delivering them at an appropriate point in time to clients. Fig. 5.15 shows an illustration how the simulator works. First a client implementing PhoneLogListenerInterface registers at a PhoneLogSimulator instance. Making use of our iterators, the simulator starts reading recorded data from the file system (or from memory if caching iterators are used). Next, the simulator starts pushing the images and sensor measurements to the registered client. It aims to push it in time intervals matching the ones produced by the camera capturing. Comparing the respective timestamps with the current system timestamp, the simulator tries to deliver the data just in time. Section 6.6 compares different implementations, including caching to non-caching variants and wait techniques.

5.7 Synthetic global shutter camera data

Feature detection and feature matching always have certain uncertainties, e.g., caused by motion blur, imposing an error on the resulting features and matches. Using natural images, it is hard to estimate the magnitude of this error, as it depends on a lot of different factors. For example, low lighting conditions or fast camera motion quickly cause motion blur artifacts on the captured images. Evaluating new algorithms using feature matches and gyroscope measurements is difficult when working with such features containing an unknown error. Further, debugging implementations of such algorithms becomes tedious. It is unclear whether unexpected results are caused by errors on the feature matches or bugs in the code. Further, it is hard to predict the expected results of such algorithms. Hence, there is the need for feature matches without imposed errors, facilitating predictable results. To get such feature matches, we implemented a
Matlab functionality simulating a rotating camera. The algorithm synthesizes feature matching data corresponding to these rotations. The camera’s motion is computed by either generating random gyroscope measurements having a distribution similar to real-world data or by using recorded gyroscope measurements.

The algorithm assumes about 30 frames per second. For each frame \(f_i\) it generates a 3D scene \(S_i\) consisting of randomly placed features. It computes the intrinsic matrix of a virtual global shutter camera which captures this 3D scene, hence producing the 3D features of each frame. By interpolating and integrating the gyroscope rotations between two frames, we project the virtual cameras to the next positions where they would capture \(f_{i+1}\). The projected virtual cameras capture again the original 3D scenes \(S_i\) which produces the 2D matches in frame \(f_{i+1}\) for the features computed in frame \(f_i\). Clients may specify the desired focal length \(f\), principal point coordinate \((c_x, c_y)\), timestamp shift \(t_d\), image dimensions, relative gyroscope-to-camera orientation and so on. Additionally, a desired error size \(e\) may also be specified. This value is used to impose a uniformly distributed error \(\in [-e/2, +e/2]\) to the x- and y-axis component of each synthesized feature and its corresponding match. Further, enabling translation adds a random translation between consecutive frames. Fig 5.16 visualizes the different building blocks of our simulator. Algorithms build for global shutter camera can also be used with global shutter camera images by assuming a rolling shutter skew of zero. Using a global shutter model for synthesizing the feature matches is less complex than simulating a rolling shutter camera. Yet, the generated feature matches are sufficient to analyze and debug our algorithms.
5.8 Partial calibration and synchronization using average pixel translation rates

Using average pixel translation rates allows to efficiently compute an estimate of the timestamp delay $t_d$ and focal length $f$. The algorithm described in Section 4.3 takes advantage of such pixel translation rates computed from feature matches and gyroscope measurements. It first computes an estimate of $t_d$ using grid search and function similarity. Afterwards, $f$ is estimated by using least squares. Both the grid search and the least squares are very efficient to compute. The partial calibration and synchronization algorithm suits very well for computing initial estimates of $t_d$ and $f$ which can then be refined by other algorithms. Further, it allows to verify the success of sensor synchronization. We provide both a Matlab and a C++ implementation of this algorithm. Fig. 5.17 shows a an overview of the C++ implementation. The algorithm involves computing average pixel translation rates for each input frame’s matched features and each gyroscope measurement. These translation rates are then interpolated on a “timestamp grid” with fixed intervals. The resulting translations are then stored as members of the `PixelTranslationRate` object. Calling `computeTimestampDiff(...)` estimates the timestamp delay $t_d$ using the stored, equally spaced pixel translation rates to compute function similarity (as explained...
5 Implementation

Figure 5.17: Diagram of the main building blocks of our pixel translation rate based partial synchronization and calibration algorithm.

in Section 4.3.1). Invoking `computeFocalLength(...)` computes a least squares estimate of the focal length using the stored translation rates. The least squares computation is done by using an OpenCV implementation\(^{25}\). Both calls directly adapt the corresponding values of the passed `pixelTranslationRate::Parameters` argument. The C++ code is contained in the `lib_parameterEstimation` project integrated as a library in our CMake framework. Written as platform independent code, it seamlessly integrates into either executable or Android projects. Further, we also added tests to verify the functionality of the code. The Matlab code is structured very similarly. Hence, we omit a specific description in this document.

Using the pixel translation rate only allows to compute an initial estimate of \( t_d \) and \( f \). Therefore, we implemented a coplanarity constraint based algorithm to compute more accurate results based on these initial estimates. This algorithm is presented in the following section.

5.9 Calibration and synchronization using coplanarity constraint and grid search

Our coplanarity constraint and grid search based calibration and synchronization algorithm allows to compute accurate estimations of focal length \( f \), timestamp delay \( t_d \) and principal coordinates \((c_x, c_y)\). The algorithm computes one parameter at once and proceeds in multiple phases, as described in Section 4.4. Fig. 5.18 shows an overview of the algorithm’s the C++ implementation. The `compute(...)` method of class `CoplanarityConstraintEstimator` is used to estimate one specific parameter. It takes feature matches, gyroscope measurements, a pointer to

\(^{25}\) See [http://docs.opencv.org/modules/core/doc/operations_on_arrays.html](http://docs.opencv.org/modules/core/doc/operations_on_arrays.html) for more details.
Figure 5.18: Diagram of the main building blocks of our coplanarity constraint and grid search based synchronization and calibration algorithm.

an instance of an AbstractGridSearchStrategy and a pixelTranslationRate::Parameters object as inputs. The strategy object defines which parameter to estimate (estimating $t_d$ is currently implemented as default if a NULL pointer is passed). First, groups of horizontally close and vertically distant features are prepared as described in Section 4.4.2. The grid search then computes the coplanarity constraint values for each group using different values of the parameter to be estimated. This computation is represented by compute determinants. It basically builds an $m \times n$ matrix, where $m$ is the number of groups and $n$ is the number of different parameter values, storing all the determinants resulting from the coplanarity constraint. The computation requires to interpolate rotations for each matched feature pair contained in a group. First sorting the groups according to their contained feature matches’ timestamps allows to just iterate over the gyroscope measurements and the groups for this computation. Section 4.4.1 gives a detailed description on how these determinants are computed for each group. Next, the values of problematic groups are rejected by evaluate determinants as described in Section 4.4.3. Compute(...) then determines the optimal parameter value based on the remaining coplanarity constraint values of the group. Further, it adapts the passed pixelTranslationRate::Parameters accordingly. The computeAlternatingParameter(...) method performs multiple iterations of alternating parameter estimation by repeatedly calling the compute(...) providing different strategies.
5 Implementation

Besides other debugging facilities, the CoplanarityConstraintEstimator class additionally offers an option to store and load specific group configurations. Hence, we are able to generate comparable results for invocations with different initial parameters. The implementation of this algorithm is also located in the lib_parameterEstimation project, hence making it available to all our CMake projects. Further, we provide tests for this implementation using our CMake framework. The code is written platform independently which enables usage in executable projects and android projects.

Additionally, we provide a Matlab implementation of this algorithm, capable of computing estimates of $t_d$. This code includes nested loops which are known to be inefficiently handled by Matlab. Therefore, we also provide a Matlab interface delegating the all the computations to a more efficient C++ executable.

5.10 Calibration and synchronization using pixel translation rates, coplanarity constraint and grid search

The computation time of the coplanarity constraint and grid search implementation presented in the previous section is directly dependent on the width of the search radius used by the grid search. Therefore, keeping this search radius as small as possible is advantageous for our application. Having a small search radius is only possible if the initial parameter estimations are sufficiently accurate (otherwise we might miss the optimal parameter values). A reasonable estimate of the principal point’s coordinates $(c_x, c_y)$ is the image’s center, but $f$ and especially $t_d$ may differ significantly between different experiments. Hence, we use the implementation of partial calibration and synchronization algorithm using average pixel translation rates (presented in Section 5.8) to initialize the estimations of $f$ and $t_d$.

The C++ implementation runAlternatingParameterGridSearch(...) declared in ParameterIdentificationParameters.hpp of lib_parameterEstimation first runs the pixel translation and grid search algorithm described in Section 5.8 to compute an initial estimate of $t_d$ and $f$. Next, it invokes the coplanarity constraint and (alternating parameter) grid search implementation presented in Section 5.9. Clients call this function by passing a parameterIdentification::Parameters object which then gets adapted according to the estimated parameters. Our algorithms can be configured using many parameters. Further, some of the parameters are even shared among the implementations (e.g., focal length or principal point). We used a parameter object pattern requiring a user to set the parameters of the algorithms using fields of a special objects. The client then passes the prepared object to the algorithm along with other arguments. Besides avoiding a large number of arguments in the algorithms’ method signatures, this pattern allows reusing the definitions of parameters by inheritance. Further, it offers a clean way of providing default values and in-code documentation. Fig. 5.19 shows the inheritance hierarchy of our parameter objects. parameterIdentification::BaseParameters contains the parameters required by both the pixel translation rate and the coplanarity constraint based algorithms. Hence, both the pixelTranslationRate::Parameters and the coplanarityConstraint::Parameters classes inherit from this basic parameter class and add algorithm specific parameters. The methods declared in ParameterIdentificationParameters.hpp
5 Implementation

![Diagram showing the C++ inheritance hierarchy of parameter objects for our algorithms.](image)

**Figure 5.19:** C++ inheritance hierarchy of parameter objects for our algorithms. We only show a selection of the contained parameters.

call both the pixel translation rate and the coplanarity constraint based algorithms. Hence, they expect an object of type `coplanarityConstraint::Parameters` which inherits all the necessary parameters. The inheritance structure shown by Fig. 5.19 is a classical instance of the diamond inheritance problem. While some programming languages prohibit such constructs, C++ enables this elegant solution using virtual inheritance.

5.11 Image enhancement code base

This thesis aims to provide a platform for performing sensor synchronization and camera calibration as well as gyroscope-aided image enhancement. Severin Münger in his bachelor thesis [39] analyzed different gyroscope-aided motion blur removal and rolling shutter correction algorithms. He also provided C++ implementations for these algorithms embedded in a Qt [50] GUI application. We factored this code into our CMake framework by separating it into a platform independent library implementation (`lib_blurRemoval`) and the Qt based graphical user interface (`exec_blurRemoval`) shown by Fig. 5.20. Further, we replaced all the previously used buffers with the implementations provided by this thesis (see Section 5.6.1) to enable compatibility with all the other code in our framework. The algorithm performing rolling shutter compensation is based on the code provided by Karpenko et at. [12] and requires OpenGL
ES. Hence, we also provide a customized integration of the Qualcomm Adreno Framework\textsuperscript{26} (along with GLFW \textsuperscript{27} required for building executables) in our CMake framework. The Adreno framework is a cross-platform tool for developing OpenGL ES based applications. Hence, we use it to maintain the necessary OpenGL ES state for the rolling shutter compensation code. Further, this thesis also provides tests ensuring equivalent results for the refactored and the original code.

### 5.12 Image enhancement Android application

As described in the previous section, we had the opportunity to include image deblurring and rolling shutter correction code provided by a thesis by Sevein Münger [39] into our thesis. Combining the resulting image enhancement library with our sensor synchronization and camera calibration library, we developed an image enhancement Android application as proof of concept. The development of this application was even more facilitated by our CMake based framework for Android application development. The application has three states:

- **Idle**
  The application has just been started and waits for user interaction. Fig. 5.21 shows a screenshot of the application in the idle state.

- **Capturing**
  The user pressed the “START CAPTURING” button. Now the application records all camera images and gyroscope sensor measurements. The user is supposed to move the camera constantly as the recorded data is used for calibration later on.

- **Calibrated**
  The user pressed the “EST” button after recording data for about ten seconds. Next, the

\textsuperscript{26} See https://developer.qualcomm.com/mobile-development/maximize-hardware/mobile-gaming-graphics-adreno/tools-and-resources.

\textsuperscript{27} See http://www.glfw.org/
application runs feature tracking using ORB features [52] (we use the ORB detector as it did run very fast in our performance evaluation, see Section 6.4). After finishing the feature tracking, camera calibration and sensor synchronization is performed as described in Section 5.9. Upon finishing the calibration, the application is ready for deblurring captured images.

Figure 5.21: Screenshot of our camera calibration, sensor synchronization and image enhancement Android application in the idle state.

If the application is in the capturing or calibrated state, the user can press the red “CAPTURE” button to store an image for deblurring. Pressing the “SWITCH” button toggles between camera

Figure 5.22: Screenshot of the image enhancement view of our camera calibration, sensor synchronization and image enhancement Android application. The left image shows a blurry camera image while the right one displays sharpened version.
preview and image enhancement mode. The image enhancement mode shown by Fig. 5.22 consists of additional “DEBLUR” and “RSCORRECT” buttons at the top. The former is used to trigger image deblurring, while the latter performs rolling shutter correction. Unfortunately the rolling shutter correction does not yet work as OpenGL ES state initialization is more involved on Android than on an ordinary computer. Further, the image enhancement view also contains two surfaces for images, as shown by Fig. 5.22. The left one always shows the image most recently captured by pressing the “CAPTURE” button. The right one displays the output of the deblurring algorithm.
6 Experiments

We have conducted several experiments to test the performance of the implemented camera calibration and sensor synchronization algorithms. In this chapter we summarize our findings. First, we show how the camera is calibrated in OpenCV, then we show the strengths and weaknesses of each algorithm. Further, we propose and analyze enhancements for the algorithms. All the images and sensor measurements were recorded using our ETHCamera Android application. The images were captured with a rate of approximately 30 frames per second and a resolution of 720x480 pixels.

6.1 Camera calibration using OpenCV

We performed the OpenCV camera calibration described in Section 4.1 multiple times for a LG Nexus 5 running Android Lollipop (21). Fig. 6.1 shows the results of the OpenCV based camera calibration. The input images of all the experiments conducted have a width and height of 720 and 480 pixels respectively. The principal point is usually located at the image center. Hence, we assume 

\[(c_x, c_y) = (360, 240)\]

The results listed in Fig. 6.1 support this assumption. The average principal point coordinates are at (358.2, 240.7). Still, the listing also shows some variation between the different measurements. This algorithm assumes a global shutter camera. We used a rolling shutter camera to capture the images. Hence, slight rolling shutter distortions could cause these errors. Another reason could be small pattern detection errors.

<table>
<thead>
<tr>
<th>Measurement Nr.</th>
<th>#Images</th>
<th>Reprojection error</th>
<th>(c_x)</th>
<th>(c_y)</th>
<th>(f_x)</th>
<th>(f_y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>0.198</td>
<td>359.3</td>
<td>244.8</td>
<td>642.0</td>
<td>640.3</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>0.177</td>
<td>361.7</td>
<td>234.6</td>
<td>642.3</td>
<td>641.3</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>0.173</td>
<td>357.5</td>
<td>241.2</td>
<td>638.7</td>
<td>639.3</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>0.211</td>
<td>362.2</td>
<td>250.0</td>
<td>645.6</td>
<td>644.2</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>0.193</td>
<td>353.6</td>
<td>237.0</td>
<td>651.3</td>
<td>650.0</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>0.163</td>
<td>354.8</td>
<td>236.1</td>
<td>650.4</td>
<td>651.9</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>0.186</td>
<td>358.2</td>
<td>240.7</td>
<td>645.1</td>
<td>644.5</td>
</tr>
</tbody>
</table>

**Figure 6.1:** Results of the OpenCV camera calibration algorithm. We performed six measurements, each consisting of 16 input images capturing an asymmetric point grid. The input images have a resolution of 720x480 pixels.
6.2 Extended Kalman filtering based online calibration and synchronization

This algorithm has several advantages, as it estimates a lot of parameters at once. Further, it is able to run online and theoretically also capable to dynamically adapt to changing ground-truth values. Another interesting property is the fact of the algorithm not being restricted to a certain scene or motion pattern. Because of these desirable properties, we decided to evaluate this approach for performing camera calibration and timestamp synchronization. Fortunately, Jia’s first paper describing the EKF-based approach from 2013 [42] includes a Matlab implementation and one test set. The provided testing data incorporates gyroscope measurements, frame timestamps and matched feature descriptions for each frame. Fig. 6.2 shows how the Matlab implementation performs on this data set. The algorithm starts with the feature matchings of the very first frame at iteration zero using default values, e.g., \((c_x, c_y) = (360, 240)\). It then successively incorporates feature matchings of the next frames in each iteration. The plots for the different parameter estimations show the values converging towards the actual ground truth. All the analysis and findings of the following sections are based on the aforementioned Matlab code. Please refer to Section 4.2 for more details concerning the algorithm’s internals.

6.2.1 Verifying the relative orientations

As explained in Section 2.4, we account for the different coordinate systems used by Android’s API and the image data recorded in a landscape orientation by applying a transformation

\[
\omega(t) = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} \omega_x^0(t) \\ \omega_y^0(t) \\ \omega_z^0(t) \end{bmatrix}
\]

to all rotations velocities \(\omega^0(t)\) returned by the Sensor API. We evaluate the performance of Jia’s Matlab code by applying images and sensor measurements recorded by our ETHCamera app. Hence, we have to make sure our data sets’ orientations match the one of the sample data provided by Jia in order to get correct (i.e., comparable) results. We verify the correctness by showing that the gyroscope rotations of the sample data match the motion described by the included feature matches when applying our proposed transformation.

First, we used the average pixel translation rate (see Section 2.11 for more details) technique to ensure the correctness of the x- and y-axis transformations. I.e., we applied our proposed transformations to the gyroscope measurements and analyzed the resulting correspondence of image and gyroscope pixel translation rates. Fig. 6.3 shows the corresponding figure for both axes. It exhibits a clear alignment of the pixel translation rates and hence establishes the correctness of the x- and y-axis transformations. Unfortunately, the pixel translation rate is not suitable to evaluate z-axis rotations. Given a uniform distribution of matched image features, the effect of a z-axis rotation cancels out because z-rotation causes pixel movements on circles which are averaged to zero. Therefore, we use the previously described x- and y-axis transformations and perform a grid search for \(t_d\) using the coplanarity constraint’s average
determinants (see Section 4.4) with $\omega_z(t) = -\omega_0^z(t)$ and $\omega_z(t) = \omega_0^z(t)$ to show the correctness of our z-axis transformation. As described in Section 2.10, its determinant values become smaller the better the image feature and the gyroscope measurements are matching. Fig. 6.4 visualizes the coplanarity constraint’s average determinant value for different $t_d$ shifts based on the sample data set. As $\omega_z(t) = -\omega_0^z(t)$ results in smaller determinant values, we conclude that our z-axis transformation is also chosen correctly.

The extended Kalman filtering algorithm performs very well with the provided data set, as shown by Fig. 6.2. Unfortunately, it did not perform equally well for data recorded by our ETHCamera application. The following sections will analyze different problems of the EKF-based approach and suggest an alternative algorithm.
Conclusion

Our experiments have shown that the transformation we apply to all rotations is also valid for the sample data provided by Jia [42]. Hence, we can use images and gyroscope measurements recorded with our ETHCamera as valid input of Jia’s extended Kalman filtering algorithm.

6.2.2 State initialization

The EKF’s state vector needs to be initialized with care, as the filter will not converge otherwise. Thus, we have to be able to guess all of the parameters to a reasonable degree of
Figure 6.4: Visualization of the coplanarity constraint’s average determinant values subject to grid search using different transformations for $\omega_0^z$. Further, x- and y-axis transformations are applied as described in Section 2.4.

accuracy. The sample data set provided along with the Matlab code uses the following values for state initialization:

\[
\begin{align*}
t_s &= 0.025 \\
t_d &= 1.175 \\
f &= 700 \\
c_x &= 360 \\
c_y &= 240
\end{align*}
\]

\[
\text{r}_{\text{cam}} = [\pi/\sqrt{2}; -\pi/\sqrt{2}; 0];
\]

Fig 6.5 shows how the filter evolves when using different values for the initial state. It compares the results using a modified guess for the relative camera rotation $r_{\text{cam}}$ and the timestamp shift $t_d$. Depending on the modification’s nature and magnitude, the algorithm may fail by returning completely unrealistic values for certain parameters as e.g., negative principal point coordinates. Further, it may also result in wrong parameter estimations. Hence, we need a sufficiently accurate initial guess of the state in order to obtain practical results.
Jia’s Kalman filtering algorithm needs accurate enough initial guesses. Otherwise the filter does not converge to the true values. Hence, these defaults either have to be guessed (e.g., \((c_x, c_y)\) at the image center) or computed.

### 6.2.3 Performance for camera motion around specific camera axes

We analyzed Jia’s EKF Matlab code using images and gyroscope measurements recorded with our ETHCamera Android application. The first scene we used consists of a planar surface,
as shown by Fig. 6.6. A second one (see Fig. 6.7) is made of a panorama view consisting of far away objects. Both data sets were captured using roll, pitch and yaw motion patterns. All the recorded images and gyroscope measurements can be found in the supplemental material. Fig. 6.8, Fig. 6.9 and Fig. 6.10 show how some of the parameters estimated by the algorithm evolve. The results are discussed in the following sections.

**Figure 6.6:** Images of the planar scene. Pieces of a jigsaw were scattered on the floor and captured using our **ETHCamera** Android application. The above images are ordered from left to right and show a sequence produced by a roll (y-axis rotation) camera motion from left to right.

**Figure 6.7:** Images of the panorama view. The view from the Zürichberg towards Wallisellen was captured using our **ETHCamera** Android application. The above images are ordered from left to right and show a sequence produced by a roll (y-axis rotation) camera motion from left to right.

**Rolling shutter skew \( t_r \)**

Fig. 6.8 shows how the rolling shutter skew \( t_r \) estimation produced by Jia’s EKF algorithm evolves. For each scene, the illumination intensity was the same for all three capturings. Hence, we expected the rolling shutter skew to be constant. In fact, Android’s **Camera2** API provides the actual rolling shutter skew value which was constantly reported as \( 0.031083360 \) s for all capturings. However, the estimation of \( t_r \) using the EKF algorithm shows significant differences for both data sets depending on the motion pattern.

**Focal length \( f \)**

Fig. 6.9 shows how the focal length \( f \) estimation produced by Jia’s EKF algorithm evolves. For each scene, the focus was fixed for all three capturings. Hence, we expected the estimated values to be about the same. Other experiments have proven the focal length estimation to be dependent on the features’ quality. This explains, the differences between roll and pitch shown by Fig. 6.9. However, the estimations produced by a yaw motion around the z-axis differ significantly from the ones produced by roll and pitch motion patterns.
6 Experiments

Figure 6.8: Rolling shutter skew $t_r$ estimated by Jia’s EKF algorithm for roll, pitch and yaw camera motion. The figures show the estimation for different scenes captured by our ETHCamera Android application.

Figure 6.9: Focal length $f$ estimated by Jia’s EKF algorithm for roll, pitch and yaw camera motion. The figures show the estimation for different scenes captured by our ETHCamera Android application.

Principal point $(c_x, c_y)$

Fig. 6.10 shows how the principal point estimation $(c_x, c_y)$ produced by Jia’s EKF algorithm evolves. The principal point describes the optical center of the camera. Usually it is located in the middle of the picture. Nevertheless, inaccuracies of the lens’s quality or the optical system can lead to a deviation of this value. The principal point’s actual coordinate is independent of focal length and exposure time. Hence, we expect the estimations of different capturings using the same camera to match. The results of Jia’s algorithm shown by Fig. 6.10 clearly
Figure 6.10: Principal points $(c_x, c_y)$ estimated by Jia’s EKF algorithm for roll, pitch and yaw camera motion. The figures show the estimation for different scenes captured by our ETHCamera Android application.

indicate different outcomes for $(c_x, c_y)$ considering both the planar (Fig. 6.10a) and the panorama (Fig. 6.10b) data set.

Conclusion

We fixed all the camera settings, therefore we expected the output values for all parameters, except the timestamp delay $t_d$, to be constant for all data sets. However, the experiments show different results for different camera motion. Further, some of the estimations even converge to completely unexpected values.
6.2.4 Conclusions

Section 6.2.2 demonstrated the need for accurate initial guesses of the state in order to obtain satisfactory results. Certain parameters allow to use simple, yet accurate, guesses. E.g., the principal point will usually be located somewhere near the image’s center. Other parameters are more difficult to guess, e.g., $t_r$, $t_d$, $f$ or the transformation between gyroscope and camera coordinate systems have to be measured or somehow deducted from the input data.

Analyzing different real-world data sets in Section 6.2.3 revealed other difficulties. We expected the outcome of the different runs for rolling shutter skew $t_r$, focal length $f$ and principal point $c_x, c_y$ to be constant. However, depending on the captured scene and the motion pattern, the estimation resulted in different values contradicting our expectation. Therefore, we decided to try other calibration and synchronization approaches, as the extended Kalman filtering approach did not turn out to be as robust as we expected.

6.3 Synchronization and calibration using coplanarity constraint

Our coplanarity constraint based synchronization and calibration algorithm estimates one parameter at once. The optimal value is found by computing the constraint’s outcome for different values of the parameter (i.e., by computing grid search). After estimating one parameter it proceeds by the next one. First, the algorithm builds different groups of features used for the constraint computation. Next, it computes coplanarity constraint values for each group using each value of the current parameter. The algorithm then combines the different estimations by averaging over the groups’ values. Section 4.4 explains the algorithm in more detail.

The coplanarity constraint usually works well for camera motion that combines rotations around multiple axes. The better our parameter estimation, the smaller the constraint’s value should become. Hence, we expect to find a local minimum in our search window. Fig. 6.11 shows the results of one iteration of coplanarity constraint and grid search. Each graph shows the average coplanarity constraint values resulting from performing grid search with a particular parameter. All the graphs show a local minimum, as expected. However, running grid search experiments using images and gyroscope measurements captured by strong rotation motion around one specific axis returned surprising results. Fig. 6.12 also shows the results of one iteration of coplanarity constraint and grid search with alternating parameters. The input images and gyroscope rotations captured strong camera motion around a specific axis. The graphs of focal length $f$ and principal point component $c_y$ still show a local minimum in the grid search interval. However, Fig. 6.12b contradicts our expectation by showing a local maximum. Hence, the minimum value of this interval is reported at its boundary leading to a wrong result for $c_x$. Hence, we conducted some analysis for the causes of the described behavior which will be discussed in the following sections.
Figure 6.11: Calibration results after one round of coplanarity constraint and grid search with alternating parameters. The data was captured by performing a pitch motion (i.e., rotation around the x-axis). Further, the image dimensions are 720x480 pixels in x and y direction respectively. The calibration results are: \( f = 653 \), \( c_x = 390 \) and \( c_y = 243 \). This example meets our expectations. The grid search produced a local minimum for \( f \), \( c_x \) and \( c_y \).

Figure 6.12: Calibration results after one round of coplanarity constraint and grid search with alternating parameters. The data was captured by performing a strong pitch motion (i.e., rotation around the x-axis). Further, the image dimensions are 720x480 pixels in x and y direction respectively. The calibration results are: \( f = 659 \), \( c_x = 560 \) and \( c_y = 247 \). This example does not meet our expectations. The estimations of focal length \( f \) and principal point component \( c_y \) are very similar to the ones of Fig. 6.11. Nevertheless Fig. 6.12b does not show a local minimum in the grid search interval. Hence, the minimum value is detected at the interval’s boundary which results in a wrong result of \( c_x = 560 \) (compared to \( c_x = 390 \) in Fig. 6.11c).
6.3.1 Impact of feature detection error

Feature detection errors are inevitable in our application. Motion blur or other detection inaccuracies will certainly introduce such imprecision. Hence, we conducted research on the leverage such errors have on the coplanarity constraint. To do so, we used our synthetic data tool (see Section 5.7) and gyroscope measurements for roll, pitch and yaw motion patterns. Further, we added uniformly distributed errors of varying magnitude on the synthesized features. Fig. 6.13 shows the parameter value producing a minimal average coplanarity constraint value for principal point coordinate \((c_x, c_y)\) and focal length \(f\). We clearly see the parameter estimation
degrading as the imposed error is chosen bigger. Further, there seems to be a correlation between the motion’s main rotation axis and the parameter influenced the most by the error. Namely, the estimation for the parameter on the rotation axis degrades faster than the other ones.

### 6.3.2 Impact of feature detection error on one coplanarity constraint group

We analyzed the coplanarity constraint value of one exemplary feature group to get an idea of what causes the maximum instead of an expected minimum shown by Fig. ???. The actual effect produced by imposing an error on feature matches highly depends on how the features are positioned relatively to each other and the shape of the error vectors. Generally, the imposed error shifts the minimum coplanarity constraint value by some amount as shown by Fig. 6.14.

The coplanarity constraint is explained in more detail in Section 2.10. An imposed error first alters the directions of a matched feature pair \((\mathbf{p}_i', \mathbf{p}_i')\) to \((\mathbf{p}_i', \mathbf{p}_i') = ((\mathbf{p}_i' + \mathbf{e}_i'), (\mathbf{p}_i' + \mathbf{e}_i'))\) where \(e_i'\) and \(e_i\) are feature specific error vectors. The computation of the constraint’s value works with normalized vectors. Hence, it first scales \(\mathbf{p}_i'\) and \(\mathbf{p}_i\) to unit length. Next, some of the modified feature vectors of each group are rotated. Given a global shutter camera\(^1\), only the features \(\mathbf{p}_i^2\) are affected by such a rotation. Hence, the imposed error alters the epipolar planes’ directions, thus degrading the coplanarity of the respective normal vectors for the (actually) optimal parameters. Tampering with one component of the feature vectors also introduces a special kind of error. While a regular error has a different vector for each feature, this artificial one has the same vector for all features. Further, this vector has only one nonzero component. Such an error vector may strengthen or weaken the influence of one specific component of each feature vector direction by making the component (absolutely) bigger or smaller respectively. Hence, by adding our artificial error, we first either pull the feature vectors towards a certain axis or push them away from it before applying normalization and rotations. Therefore, performing grid search with this special kind of error introduces a motion of the epipolar plane’s orientations. These motion paths potentially differ for each epipolar plane\(^2\), causing the epipolar plane normals either to converge towards a plane or to diverge.

When performing grid search with features not suffering from a regular error, applying our artificial error certainly leads to a worse result as the corresponding epipolar plane normals already are coplanar. But when given features actually suffering from a detection error, the introduced motion among the epipolar plane orientations may actually result in a local minimum of the coplanarity constraint which is shifted with respect to the minimum shift induced by the ground truth value.

---

\(^1\) As it is the case with our synthetic data.

\(^2\) Given a rolling shutter camera, all features except \(\mathbf{p}_0'\) get rotated.

\(^3\) Depending on the shape of the actual feature vectors defining the plane’s orientation and the rotation applied to part of the feature vectors.
6 Experiments

Figure 6.14: Coplanarity constraint values of one exemplary (synthetic) feature group. The y-axis indicates the magnitude of the error imposed on the feature group. Further, the x-axis show the values for different guesses of principal point coordinates \((c_x, c_y)\) and focal lengths \(f\). The red line indicates the minimal value for each error. We thresholded the data to the smallest 40\% for increased visibility. The coplanarity constraint values were generated using \((c_x, c_y) = (360, 240)\) and \(f = 700\).

6.3.3 Impact of feature detection error on combined coplanarity constraint groups

Analyzing the coplanarity constraint values of lots of different feature groups based on feature matches with varying error supports the findings of the previous section. The added error shifts the locations of the coplanarity constraint’s minimal values of the groups. Figures 6.15a, 6.15b and 6.15c visualize the impact of feature detection errors on the position of the coplanarity constraint’s minimum value for the focal length \(f\) produced by grid search. Each of the figures shows the results for a different camera motion pattern. The features were synthesized, hence we have a known ground-truth focal length \(f\). Each column of a figure shows the offsets of the coplanarity constraint’s minimal value to the expected focal length of \(f\) as colors. Green
Figure 6.15: Figures 6.15a, 6.15b and 6.15c show the shift of the coplanarity constraint’s minimal value in a search window for focal length $f$. Each column contains the value of one feature group. The error grows bigger as we go vertically. Green indicates a small shift, while fully saturated red represents a shift of 200 (i.e., we reached the border of our search window). Fig. 6.15d summarizes the mean shift of all features for the different errors.

denotes an offset of zero (i.e., a focal length of $f$ lead to the minimal value) while red represents an offset of at most 200 (i.e., the minimum is located at the search interval’s border). Each column starts with an imposed error of zero at the bottom which grows towards the top. We clearly see the bottom row constantly colored in green, indicating a shift of zero for all groups. Going upwards, the colors generally turn more and more towards red, indicating a tendency of the minimum value being shifted towards the search interval’s border. This tendency is also reflected in Fig. 6.15d. This figure shows the mean values of the previous graphs’ rows, hence indicating the average shift resulting from the different errors. All three motions (i.e., roll, pitch and yaw) show a clear tendency of shifting the minimum value away from the optimal one.
These shifted minimum values start dominating the combined coplanarity constraint’s outcome at some point. Hence, we end up having a maximum value as illustrated by Fig. ??.

Moreover, all of the graphs of Fig. 6.15d match the observation of Fig. 6.13c. The yaw motion degrades the fastest with increasing error, followed by the pitch and the roll motion patterns. As already explained, we move the feature vectors towards a certain axis by imposing our artificial error. Given a rotation around the same axis, we hide a certain portion of the error contained in the feature vector’s direction. Having such a correlation of rotation and error axes leads to a smaller difference between $R_i \overrightarrow{p_i}$ and $\overrightarrow{p_i}'$ than compared with the effect of other rotations. Hence, the epipolar plane’s normal direction is less affected by the error. Thus, grid search for parameters on the rotation axis degrades faster than the other ones as observed in Section 6.3.1.

6.3.4 Non-minimum interval rejection

The previous sections explained the effect of shifted minimum value positions produced by feature detection errors. As explained in Section 4.4.3, we use non-minimum interval rejection to cope with the problem of such dominating shifted minimum values. Fig 6.16 shows the results of parameter estimation using coplanarity constraint combined with non-minimum interval rejection for different feature errors. Comparing with the results generated by the base implementation, shown by Fig. 6.13 reveals the increased robustness of the rejection heuristic. The results of Fig. 6.13 tend to degrade quite fast with increasing errors. In contrast, the results of Fig. 6.16a and Fig. 6.16b almost do not show any degradations at all. Also, the values of Fig. 6.16c degrade way slower than the ones without the rejection heuristic.

6.3.5 Dominant groups

The previous subsections discussed the negative effects caused by the combination of multiple problematic feature groups. Our experiments have also shown cases where a small number of feature groups numerically dominates the whole result negatively. Fig. 6.17 illustrates the problem caused by such groups. The coplanarity constraint computation proceeds by first computing the coplanarity constraint values (i.e., the resulting determinants) for all feature groups and different values of the parameter (in this case $c_y$) to be estimated. The result of this computation is shown by the surf plot at the top of Fig. 6.17. Afterwards, we combine all these estimations by computing the mean of all groups for each value of $c_y$ as shown by the plots at the bottom of Fig. 6.17. We expect the result of $c_y$ around 240px as the source images had a height of 480px. Fig. 6.17b shows how two feature groups with large coplanarity constraint values dominate the final result leading to a wrong estimation of $c_y = 175$. Fig. 6.17b shows the same set of groups after removing the smallest and largest 1% of all groups. The computed average is no longer dominated by few values leading to a significantly better estimation of $c_y = 239$. 

89
6 Experiments

Figure 6.16: Plots showing the improved outputs of the calibration algorithm for imposed errors of varying maximal lengths on synthesized data. Each plot shows the values for a roll, pitch and yaw motion. The synthetic feature matches were generated using \((c_x, c_y) = (360, 240)\) and \(f = 720\). Using non-minimum interval rejection leads to a significant improvement compared to the results shown in Fig. 6.13. There is less deviation from the ground truth even though we increase the artificial errors in the feature matching.

6.3.6 Conclusion

Defective feature detection and matching leads to feature groups where parameter estimation results are shifted with respect to the ground truth. Further, a small number of problematic feature groups with very small or very large values can dominate the whole estimation. Hence, we suggest non-minimum interval rejection and the refusal of very small or large groups. Experiments have shown the positive effect of both approaches.
6 Experiments

Figure 6.17: Influence of dominating groups on combined coplanarity constraint estimation. Both sub-figures show the determinants of the coplanarity constraint for different feature groups and values of principal point coordinate $c_y$. Fig. 6.17a shows two feature groups dominating the final result computed by averaging over all groups. In contrast, Fig. 6.17b shows the successful result after removing these dominant groups.

6.4 Feature matching performance

The goal of this thesis is to develop an application running on smartphones. Hence, feature detection and matching has to be efficient. There exists a broad range of feature detectors but not all are suitable for mobile devices due to their computational demands. We evaluated the performance of four popular feature detection and matching techniques. The data sets used are the same as in Section 6.5. $n_1$ to $n_9$ capture well illuminated natural images of varying scene depth. $b_2$ to $b_4$ capture scenes inside a building where illumination is not optimal. Having to process too many matched features slows down the application. Therefore, we configure the feature detectors to return a low number of strong features. All the feature detectors
were configured to return between 100 and 150 features. The experiment was performed on a computer equipped with an Intel Core i7-4600U processor.

Figure 6.18: Runtimes of different feature detecting and matching techniques. The data sets were captured with our ETHCamera Android application. Each point represents the average run time per frame of a specific detector. The vertical bars indicate the respective standard deviations. Given a frame-rate of 30 images per second, the detection and matching desirably requires less than $\frac{1}{30}$ s. The data sets $n_1$ to $n_9$ capture a well illuminated scene, while lighting conditions of $b_2$ to $b_4$ were not optimal. The green horizontal line indicates this boundary. Feature detection and matching was done using OpenCV on an Intel Core i7-4600U processor.

Fig. 6.18 shows the average run times per frame required by feature detection and matching. Fig. 6.18a uses optical flow (based on image intensity variations) while Fig. 6.18b uses descriptors (descriptions of image regions) to match features of successive image frames. Feature detection and matching preferably does not take too long as it otherwise slows down the whole application. The SIFT (scale-invariant feature transform [20]) detector on average requires constantly between 150 and 200 milliseconds per frame. Given a capturing rate of 30 frames per second, this detector is too slow for our application. The Adaptive feature detector internally uses FAST (features from accelerated segment test [17]) features. It iteratively adapts thresholds for these FAST features until it returns the desired number of features. The well illuminated scenes potentially return more features due to their better contrast. Hence, the Adaptive feature detector requires more iterations to return the desired (low) number of features. Therefore, the execution times show a large difference between the well lit scenes $n_1$ to $n_9$ and the other ones. Both the GFTT (good features to track [18]) and the ORB (oriented FAST and rotated BRIEF [52, 53]) run on average under 33 ms per frame for each data set. Except for the SIFT detector, the runtimes do not differ significantly between optical flow based matching (Fig. 6.18a) and descriptor matching (6.18b). Fig 6.19 shows the average number for matched features for the same data sets and detectors as in Fig. 6.18. The number of matched features is reduced for data sets $b_2$ to $b_4$ compared to the other ones. The SIFT and GFTT detectors generate more matches with descriptor matching than with optical flow.
6.4.1 Conclusion

Due to potentially limited processing capabilities on mobile devices, we favor fast feature detection and matching techniques. Therefore, SIFT feature detection is not suitable. GFTT and ORB both are very fast and hence interesting candidates for our application. Adaptive feature tracking requires accurate configuration, otherwise it is almost as expensive as SIFT.

6.5 Coplanarity constraint grid search

The previous section analyzed the robustness of the coplanarity constraint. Even small errors on feature matches may cause substantial problems. We try to cope with this problem by rejecting problematic groups of feature matches. Using synthetic data this approach performed well even for large errors. This section additionally evaluates the improved algorithm’s performance using images and gyroscope measurements captured with our ETHCamera application. First, we will compare results produced by different feature detectors and matching techniques for a combined roll (x-axis rotation) and pitch (y-axis rotation) motion. Second, the performance of our algorithm on the roll, pitch and yaw (z-axis rotation) data sets of Section 6.2.3 is analyzed.

6.5.1 Combined roll and pitch motion

Testing different camera motion patterns, we could achieve the best results using a combined roll (y-axis) and pitch (x-axis) camera rotation. Fig. 6.20 shows the respective rotation for
experiments

Figure 6.20: Camera rotation yielding the best results with our coplanarity constraint algorithm. The plots show the rotation velocities $\omega_x$, $\omega_y$, $\omega_z$ around the x-, y- and z-axis respectively. The rotation velocities have been low-pass filtered using averaging in order to remove high-frequency noise. We captured the sensor measurements (along with the images) using our ETHCamera app. The camera performed a circular motion consisting combined of roll (y-axis) and pitch (x-axis) rotations.

We evaluated our algorithm using different scenes. $n_1$ to $n_2$ capture natural images of varying scene depths. The scenes are fully exposed to sun light, hence well illuminated. Scenes $b_2$ to $b_4$ capture images inside a building where illumination is not optimal. The values of timestamp difference $t_d$ and focal length $f$ are initialized using the pixel translation rate algorithm (see Section 4.3). Principal point coordinates $(c_x, c_y)$ are initialized as the image center. The algorithm performs multiple runs per iteration. In each run it first fixes all parameter except one and then estimates the value of this parameter using grid search. Hence, we successively estimate $t_d$, $f$, $c_x$ and $c_y$ in one run. We conducted each of the following experiments by performing three iterations. Fig. 6.21 shows the results for the estimated parameters after each iteration for scene $n_7$. It compares the outcomes using different feature detection and matching techniques. The estimations for $t_d$ only vary little (i.e., $\pm 1ms$). The other parameters’ estimations are within a acceptable interval of approximately $\pm 10px$. We performed the same analysis for the twelve data sets described in the beginning of this section. Fig. 6.22 shows the resulting estimations of the different parameters for all data sets after the third iteration. The estimations for $t_d$ do only show a variation of at most $\pm 2ms$ for the well lit scenes $n_1$ to $n_9$. The scenes $b_2$ to $b_4$ with suboptimal illumination show a much larger variation of $t_d$ when using different feature detectors. The other parameters estimations for $f$, $c_x$ and $c_y$ are in acceptable boundaries for the scenes $n_1$ to $n_9$. The other scenes show bigger variations which could be problematic for our application. Both the descriptor matching and the optical flow matching show similar magnitudes of differences between the results produced by different feature detectors. Our application requires an as accurate as possible estimation of the timestamp difference $t_d$. Hence,
we compared the maximum difference of timestamp estimations of the different feature detectors for each sample data set in Fig. 6.23. The differences of optical flow based feature matching are smaller, indicating more accurate feature matching.

### 6.5.2 Performance for camera motion around specific camera axes

Section 6.2.3 showed the performance of the extended Kalman filtering algorithm for camera motion around specific camera axes. The results were not satisfactory, as the outcome was not suitable for our application for some data sets. Hence, Section 6.3 analyzed the robustness of the coplanarity constraint in presence of feature detection and matching errors. Further, we suggested possible enhancements for our grid search parameter estimation algorithm. This section shows the performance of our algorithm on these problematic data sets. As in Section 6.5.1, we performed three iterations of our alternating parameter grid search algorithm. Fig. 6.24 compares the results produced by different feature matching techniques after the third iteration and the results obtained by the extended Kalman filtering. Using optical flow for matching Fig. 6.24a, the estimations produced by our algorithm and different feature detectors are very close compared to (Fig. 6.24b). Hence, optical flow seems to generate more consistent feature matches, as we already expected in Section 6.5.1. The extended Kalman filtering leads in most cases to similar results. Nevertheless, the timestamp differences vary significantly between both algorithms. Further, our algorithm successfully prevented estimations from overshooting (see
Figure 6.22: Results of our coplanarity constraint based alternating parameter gradient descent for different data sets. We performed three iterations in which we estimated \( t_d \), \( f \) and \((c_x, c_y)\) successively. The plots show the result of each parameter at the end of the third iteration. Further, the outcome of using different feature detectors and matchers are compared. The data sets \( n_1 \) to \( n_9 \) capture a well illuminated scene, while lighting conditions of \( b_2 \) to \( b_4 \) were not optimal.

the Kalman filtering focal length estimation of the yaw camera motion capturing the planar scene shown by Fig. 6.9a).

### 6.5.3 Conclusions

Given suitable camera motion and scene illumination, our coplanarity constraint based grid search synchronization algorithm performs very well. I.e., the estimated values are close to the true values, and the calibration is done in a reasonable time of five to ten seconds. The timestamp delay \( t_d \) is estimated accurately by each of the tested feature detectors for the well illuminated data sets. Yet, the estimations if \( t_d \) for the data sets captured indoors with a weaker illumination are not that similar for different feature detectors. This indicates that the accuracy
6 Experiments

Figure 6.23: Maximum difference of $t_d$ estimations generated by different feature detectors. The optical flow line shows the differences for the $t_d$ estimations of Fig. 6.22a while the descriptor matching line shows the respective values for Fig. 6.22b. The data sets $n_1$ to $n_9$ capture a well illuminated scene, while lighting conditions of $b_2$ to $b_4$ were not optimal.

of feature detection and matching is still problematic if the contrast of the captured scene is not optimal. The camera calibration estimations for $f$, $c_x$ and $c_y$ also shows variations for different feature matching techniques. The ranges (i.e., the variations between different feature detectors as well as between different scenes) of estimated values are similar for both kinds of scene illumination. We would expect the well illuminated scenes to also result in better estimations. Hence, improved feature detection and matching could also enhance the result of the parameter estimation.

Our calibration and synchronization algorithm also performs well on the data sets capturing camera motion around specific camera axes. The results obtained by using different feature detectors are quite similar despite the non-optimal camera motion. Wrong estimations caused by problematic groups (as discussed in Section 6.3) could be successfully avoided. Further, all the estimations produced by our algorithm are in acceptable tolerances.

6.6 ETHCamera simulator

As described in Section 5.6.3, we evaluated different strategies for injecting camera or sensor events for our ETHCamera simulator offline, running on a PC. The simulator aims to reproduce the behavior of the Android application as close as possible using recorded images and sensor measurements. Hence, it has to deliver it in intervals as similar as possible to the ones described by the data’s timestamps. We compared three approaches, described by the following sections. Fig. 6.25 shows the mean gyroscope data delivery delays of different implementations of our ETHCamera simulator. This delay compares the expected to the actual delivery time. Hence, the smaller this delay the better. Further, the bars indicate the respective standard deviations. The graph shows three different implementations. The first one is our base implementation not using any caching and performing pure spinning. The second one improves on the first one by loading the input images and sensor measurements into memory (i.e., by caching) and the third one by combining the spinning with sleeping. Comparing the base with the caching
Figure 6.24: Results of our coplanarity constraint based alternating parameter gradient descent the same data sets we used to evaluate the extended Kalman filtering algorithm (see Section 6.2.3). We performed three iterations in which we estimated $t_d$, $f$ and $(c_x, c_y)$ successively. The plots show the result of each parameter at the end of the third iteration. Further, the results of using different feature detectors and matchers are compared. The line labeled with “EKF” shows the resulting value of the extended Kalman filtering. The data sets labeled with $Pl$ and $Pa$ captured the planar and the panorama scene respectively. Further, data sets labeled with subscripts $r$, $p$ and $y$ while the camera performed a roll (y-axis rotation), pitch (x-axis rotation) or yaw (z-axis rotation) motion respectively.

implementation shows an improvement from a high mean delay of 0.6 ms to 0.009 ms. Further, the standard deviation of the delay is reduced by a factor of 41, clearly backing the benefits of caching. Spinning blocks a processor while not performing any work most of the time. Hence we improved our simulator with a combination of spinning and sleeping. Using such a heuristic reduces the CPU load from 100% to about 20%. While reducing the system’s load, this approach potentially also reduces the responsiveness of the algorithm. Nevertheless, comparing the delay plots for our caching & spin and caching & sleep implementations shows the mean values and
Figure 6.25: Mean delay of gyroscope measurement delivery using our \textit{ETHCamera} simulator. The plot compares the base implementation to caching and sleep extensions. Further, the bars indicate the standard deviation of each data set.

standard deviations not being affected negatively. The graphs included in Section 8.9 show the results summarized in Fig. 6.25.
7 Conclusion and Future Work

7.1 Conclusion

In this thesis different techniques for camera calibration and sensor synchronization were evaluated. Further, we also provide platform-independent C++ source code for performing calibration, synchronization and image enhancement. Integrating this code with the CMake based framework developed by this thesis facilitates the development of locally running executables or Android projects. Three different algorithms for camera calibration and sensor synchronization were evaluated in more detail. The extended Kalman filtering using the coplanarity constraint proposed by Jia is a promising online algorithm for camera calibration and synchronization. Average pixel translation rates offers fast estimation of the camera’s focal length and sensor synchronization. Coplanarity constraint and grid search with alternating parameters is computationally more demanding, but also yields more accurate results. Further, using the coplanarity constraint incorporates camera translation which is often ignored by other sensor synchronization techniques.

This thesis conducted different experiments to analyze the problems of coplanarity based calibration and synchronization algorithms. Using synthesized feature matches, the impact of feature detection and matching errors on the results of such algorithms was evaluated. Based on the gained insights, suggestions for possible enhancements for our algorithm based on coplanarity constraint and grid search with alternating parameters were made. The evaluation of these enhancements on synthesized feature matches shows a significant improvement of estimation accuracy also for large feature detection errors. Experiments on real data confirm the robustness of the calibration with the coplanarity constraint. Using the coplanarity constraint and grid search is slow for large search spaces. Especially focal length estimation and sensor synchronization require to test a lot of possible parameters if an accurate initial value is missing. Hence, we use the efficient average pixel translation rates to reduce these search spaces drastically.

The proposed algorithm provides reliable estimations for the task of image enhancement. The sensor synchronization produces very similar results for different feature detection techniques, hence indicating a suitable robustness. Nevertheless, the estimations of other parameter (e.g., the principal point of the camera) still show some variation. Our experiments indicated a non-negligible correlation between image quality and feature detection and matching accuracy. Hence, computing accurate feature matches despite non-optimal image quality is in important requirement for robust and exact camera calibration and sensor synchronization.
The experiments conducted have shown that feature detection and matching accuracy is an important issue and has significant impact on the performance of the calibration algorithms. This thesis uses feature detection and matching techniques offered by OpenCV. RANSAC and with fundamental matrices currently is the only refinement mechanism in place. Hence, other refinement techniques (e.g., track-retrack) could potentially improve the accuracy of detected features. Further, using a known calibration pattern could also significantly improve the detected feature’s quality.

Gyroscope measurements usually include small measurement errors. Our algorithms do integrate these measurement only using very small time intervals. Hence, the drift caused by the errors is relatively small. Nevertheless, the quality of gyroscope measurements could be significantly improved by combining accelerometer and gyroscope data using Kalman filtering.

The algorithms proposed by this thesis do only consider camera rotation measured by a gyroscope. Translation is factored in by the assumption of uniform translation directions between a small number of successive camera pictures. Nevertheless, a significantly accelerating translation motion could actually have a varying direction between successive camera frames. Especially when capturing images with a low frame-rate, such non-uniform translations would degrade our results. Second, we completely ignore the information given by the accelerometer. Hence, including the translation information into the camera calibration and sensor synchronization algorithm potentially increases the accuracy of the resulting values.

Computational efficiency is of major concern for calibration and synchronization algorithms intended for mobile platforms. There is still room for improvement for the proposed coplanarity constraint and grid search with alternating parameters with respect to its runtime. The expensive grid search could be replaced by the more efficient Brent’s method [38]. Further, it remains to be evaluated whether accurate principal point estimation is even necessary for successful image enhancement. The initial value of the image center is likely to be sufficiently accurate for devices without severe defects in the lens system.

The coplanarity constraint algorithm proposed by this thesis does only estimate a subset of all possible parameters. Jia proposed an extension for lens distortion estimation for his coplanarity constraint and Kalman filtering algorithm. For example in presence of a wide-angle camera lens, ignoring the introduced image distortion would degrade the algorithm’s results significantly. Further, this thesis assumed well aligned camera and gyroscope axes. In reality, such a matching alignment typically can not be guaranteed. Hence, factoring in possible displacements of the camera and gyroscope coordinate systems may improve the calibration and synchronization accuracy.

In summary, we proposed a new method that brings us one step closer to robust automatic gyroscope-camera calibration, but there are still a couple of issues that need further analysis. We believe our method will be widely applicable in practice as it requires no special hardware and makes no strong assumptions. The proposed method works out of the box given that good quality correspondences can be found across a small number of subsequent camera frames. We
plan to make the source code of our cross-platform C++ implementation publicly available to facilitate further research on the crucial problem of gyroscope and camera calibration.
8 Appendix

8.1 CMake based framework

The framework keeps a flat hierarchy of configuration files and actual projects, as indicated by Fig. 8.1. The CMakeLists.txt files configure each project. Sections 8.3 to 8.5 explain the usage of the different project types in more detail. As explained in Section 5.5, the framework automatically searches for all relevant files. Hence, the user does not have to list the project’s actual contents. Thus, the user is free to choose any suitable internal project structure.

![Diagram]

**Figure 8.1:** General CMake framework directory structure showing configuration files and different project directories.

8.2 Project configuration files

The CMakeLists.txt configuration files are used to name and declare the projects, to specify dependencies and defining the project type. Actually the configurations look very similar for
the different project types. Hence, we do not cover the project file’s structure for each type individually. Fig. 8.2 shows a listing of an exemplary configuration file:

```cmake
project("foo")
#OR: project("lib_foo")
include(${CMAKE_CURRENT_SOURCE_DIR}/../globals.cmake)
dependency_opencv()
dependency_boost()
dependency_lib(foo)
define_library()
#OR: define_executable()
#OR: define_cvgCameraModule()
```

**Figure 8.2:** Listing of an exemplary project CMakeLists.txt configuration file.

1-2 Defines the project’s name. Given an executable project that depends on a library having the same name, certain IDE’s do not show the library. Thus, CMake configurations of such libraries may prefix the name with `lib_` avoiding name clashes.

4 Includes our framework.

6-8 Declare dependencies. See Section 8.6 for more details.

10 Used to declare a library project.

11 Used to declare an executable project.

12 Used to declare an Android module.

### 8.3 Library project structure

Fig. 8.3 shows the directory structure of a library project using our CMake based framework. The project’s top-level directory has to be prefixed with `lib_`, otherwise the framework will not be able to include the library into other projects. Libraries usually expose headers to be used by clients. Therefore a library project is obliged to contain an `include` directory containing such headers.

### 8.4 Executable project structure

Fig. 8.4 shows the directory structure of an executable project using our CMake based framework. In contrast to a library project, the top-level directory naming actually does not matter. Anyway we prefixed executable project directories with `exec_` for the sake of clarity.


8 Appendix

Figure 8.3: Structure of a library project using our CMake based framework.

Figure 8.4: Structure of an executable project using our CMake based framework.

8.5 Android module project structure

Fig. 8.5 shows the directory structure of an Android module project using our CMake based framework. Internally, the project type is derived from a library type, hence the same constraints apply as described in Section 8.3. Due to the fact that these projects are not intended to be used as regular dependencies, the include folder is expected to be empty.

8.6 Declaring dependencies

Fig. 8.2 shows the contents of an exemplary cmake configuration file which also specifies dependencies. Using OpenCV and Boost is straightforward by including dependency_opencv() or dependency_boost() respectively. Dependencies on user defined libraries are specified using dependency_lib(library_name). Using Qt basically only requires specifying dependency_qt(). Qt resource files have to be specified using qt5_add_resources(EXEC_SRCS "ressource1.qrc ressource2.qrc") after the dependency declaration.
8.7 Testing

The framework internally uses the Google C++ testing framework [46]. Hence, the C++ code has to be written as described by the documentation\(^1\). As soon as the code is stored in a file having a `.test.cpp` suffix, it is automatically included in the tests. Further, the project’s `CMakeLists.txt` file has to contain a `dependency_gTest()` dependency in order to enable testing. The testing library depends on the local helpers per default making it possible to access the filesystem. In order to disable this dependency use `dependency_gTest(TRUE)`. CMake generates an additional executable for each (sub-) project containing tests. All these tests are run at once by executing `ctest` in the root project’s build directory. Fig. 8.6b shows an example of such an execution.

8.8 Generate projects

Executing

```
cmake /path/to/project
```

generates an out-of-source build for the current platform’s default build system\(^2\). The `-G` argument may be used to specify the generator to be used. The following command shows how to generate an Eclipse CDT project under Linux:

```
cmake -G"Eclipse,CDT4.4\_Unix,Makefiles" /path/to/project
```

For Android modules we may either use the `-DBUILD_TYPE=android` in order to cross-compile an android shared library or `-DBUILD_TYPE=local` for building a local executable.

---

\(^1\) See [https://code.google.com/p/googletest/wiki/V1_7_Primer](https://code.google.com/p/googletest/wiki/V1_7_Primer).

\(^2\) I.e., Make for Linux platforms.


```cpp
#include "gtest/gtest.h"
#include <string>

using namespace std;

TEST(Example, hello_world) {
    EXPECT_EQ(string("Hello World!"), string("Hello World!"));
}
```

(a) The code.

```sh
humairl@t440s:/tmp/foo$ ctest
Test project /tmp/foo
Start 1: Example.hello_world
1/1 Test #1: Example.hello_world .............. Passed 0.00 sec

100% tests passed, 0 tests failed out of 1
Total Test time (real) = 0.00 sec
```

(b) The execution.

Figure 8.6: Hello world example using the Google C++ testing framework [46].

8.9 ETHCamera simulator delivery delays

![Graph showing delivery delays](image)

Figure 8.7: ETHCamera simulator delivery delays not using any cache. Further the thread always spins between the deliveries.
Figure 8.8: ETHCamera simulator delivery using a cache. Further the thread spins between the deliveries.

Figure 8.9: ETHCamera simulator delivery delays using a cache. Further the thread sleeps between the deliveries.
Bibliography


Bibliography


