Structure from Motion using a Single Camera on Embedded Systems

Bernd Zobl

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Embedded Systems Design

in Hagenberg

im Juni 2015

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 17, 2015

Bernd Zobl

Contents

Preface

Although this thesis is an individual work, many people helped to make it happen. I would like to take this opportunity to mention them here. First of all I want to thank my parents and family who supported me during my whole time as a student, as did Jasmin Walzinger. She was on my side all the time during this work and encouraged me to do my best.

I also want to thank Florian Eibensteiner who supervised my work. He was a great help and guided me to the completion of this thesis. My thanks also go to Klaudia Steinkellner, the programme administrator for Hardware Software Design and Embedded Systems Design, who in all my years in Hagenberg was supporting us students in every possible way. I also appreciate the work of the lectors and professors teaching at the university. Last of all I want to thank my friends and colleagues who always supported me, gave useful input, constructive criticism and let me forget about my work for a while, if that was necessary.

During my time in Hagenberg, my interest in embedded systems was awoken. This thesis combines my enthusiasm for Linux and programming with that for embedded systems. Although the field of computer vision was new to me at the start of my work, I quickly acquired some knowledge on it, not least because my supervisor guided me to the right literature.

Kurzfassung

Structure from Motion (SfM) ist ein Ansatz um eine dreidimensionale Karte aus zwei oder mehreren Bildern zu erstellen. Im Gegensatz zu Stereo Vision, wo zwei Kameras genutzt werden, werden bei SfM die Bilder nacheinander aufgenommen. Indem die Kamera bewegt wird, werden zwei unterschiedliche Ansichten der gleichen Szene generiert, aus der die 3D Punkte trianguliert werden können. Diese Arbeit beschäftigt sich zunächst mit den Grundlagen maschinellen Sehens und der Epipolargeometrie. Anschließend werden die drei Komponenten von SfM vorgestellt: Bildmerkmale erkennen und vergleichen, Berechnung der Positionen der Kamera und Triangulation. Die vorgestellten Algorithmen werden in MATLAB verifiziert und eine Implementierung für das NVIDIA Jetson TK1 Entwicklungsboard wird mithilfe der OpenCV Bibliothek realisiert. Darauf basierend werden einige Optimierungen vorgeschlagen, die die GPU des NVIDIA Tegra K1 verwenden. Der Vergleich der Implementierungen zeigt, dass der optimierte Algorithmus, der für die Triangulation verwendet wurde, schneller ist als der originale, solange die Anzahl der Punkte groß genug ist. Obwohl gezeigt wird, dass das Testprogramm mit manuell extrahierten Bildmerkmalen funktioniert, kann es keine korrekte 3D Karte aus dem Livestream der Kamera erstellen. Die OpenCV Bibliothek bietet sowohl eine CPU als auch eine GPU Variante des SURF Algorithmus zur Bildmerkmalerkennung an. Die in dieser Arbeit durchgeführten Tests zeigen, dass letztere weder geeignete Paare findet, noch einen Speedup erzielen kann. Die CPU Variante generiert hingegen verlässlichere Merkmale und ist zudem noch schneller. Passende Parameter für den Algorithmus, bzw. ein anderer Erkennungsalgorithmus, sind allerdings noch zu finden um einen Livestream verlässlich zu analysieren.

Abstract

Structure from Motion (SfM) is an approach to generate a three dimensional point cloud from two or more images. Unlike stereo vision, which uses two cameras, SfM takes images sequentially using a single camera. By moving the camera, different views of the same scene are generated, from which the points in 3D space can be triangulated. Next to introducing the fundamentals of computer vision and epipolar geometry, this thesis presents the three main components of SfM, i. e., feature detection, pose estimation, and triangulation. The algorithms are verified using MATLAB and an implementation for the NVIDIA Jetson TK1 development board using the OpenCV library is disclosed. Additionally, several optimizations using the GPU of the NVIDIA Tegra K1 are proposed and then compared to the baseline implementation. The comparisons show that the optimization of the iterative linear least-squares method, that is proposed for solving the triangulation problem, performs better than its CPU counterpart if the number of points to be triangulated is large enough. Although the test application is shown to be working with manually extracted features, using the OpenCV implementation of the SURF feature detector yields no usable point cloud of the captured scene. The library offers both a CPU and a GPU version of the algorithm. The tests conducted in this thesis show that the former performs better. Not only does the GPU alternative find less correct matches but also it takes more time. Nevertheless, fitting parameters for the SURF implementation or another feature detection algorithm have yet to be found.

List of Abbreviations

AC	auto-correlation function
API	application programming interface
BRIEF	binary robust independent elementary features
CAT	computerized axial tomography
CPU	central processing unit
CUBLAS	CUDA basic linear algorithm subprograms
CUDA	compute unified device architecture
CV	computer vision
DLT	direct linear transformation
eMMC	embedded MultiMediaCard
ePnP	efficient Perspective-n-Point
FAST	features from accelerated segment test
FLANN	fast library for approximate nearest neighbors
GCC	GNU compiler collection
GLOH	gradient location-orientation histogram
GNU	GNU's Not Unix!
GPU	graphics processing unit
I2C	inter-integrated circuit
L4T	Linux for Tegra
LiDAR	light detection and ranging
LMedS	least median of squares
LS	least-squares
LSP	Liskov substitution principle

MOPS	multi-scale oriented patches
NVCC	NVIDIA CUDA compiler
OpenCL	open computing language
ORB	oriented FAST and rotated BRIEF
PCB	printed circuit board
PCIe	peripheral component interconnect express
PCL	point cloud library
PnP	Perspective-n-Point
RAM	random access memory
RANSAC	random sample consensus
SATA	serial ATA
SfM	Structure from Motion
SFU	special function unit
SIFT	scale invariant feature transform
SIMD	Single-Instruction, Multiple-Data
SIMT	Single-Instruction, Multiple-Thread
SM	streaming multiprocessor
SoC	system on a chip
SSE	streaming SIMD extensions
SURF	speeded up robust features
SVD	singular value decomposition
UART	universal asynchronous receiver/transmitter
UAV	unmanned aerial vehicle
USB	universal serial bus
WSSD	weighted sum of squared differences

Chapter 1

Introduction

An embedded system is a computer system with a dedicated function within a larger system. Therefore it can be found in a wide range of applications. Embedded systems are present in virtually any electronic device ranging from a coffee machine over smart phones to cars and aircraft. The core of such a system is its processor. In recent years ordinary microprocessors were superseded by the more powerful, yet energy efficient, ARM processors. With the release of the Raspberry Pi [18] in 2012, embedded systems became very popular. This system features, as do many other embedded systems, a system on a chip (SoC) design and is running Linux. That makes it usable like a conventional desktop system with the difference that all components of the computer system, i. e., processors, memories, clock generators, peripherals, etc., are integrated into one single chip, making it possible to build powerful but small systems, e. g., the size of the Raspberry Pi's printed circuit board (PCB) is 8.6 by 5.7 centimeters. Many modern SoCs contain a graphics processing unit (GPU) as well, making it possible to connect displays and providing user interfaces.

The GPU also offers the possibility to perform complex computational problems in a very efficient way. This introduces the field of computer vision (CV), i. e., the acquiring, processing and analyzing of images, to embedded systems. CV problems generally are solved mathematically using matrix and vector operations. GPUs have the ability to perform many operations in parallel, and therefore, are suited for solving these problems. Programming the GPU in the past was quite inconvenient: the matrices had to be specially prepared and copied to the texture memory, which was the only memory accessible. A shader could then be used to compute the result using the previously copied data. In recent years two programming models, compute unified device architecture (CUDA) and open computing language (OpenCL), have emerged that provide a way to program GPUs using an extended form of the C programming language, making the development of software easier.

Systems that process real-time image data of the environment can be found in many areas, e. g., in cars they detect objects to avoid collisions, in manufacturing they are used for finding foreign matter on conveyor belts, and in medical applications they are used to create computerized axial tomography (CAT) images. Another application of such systems is the creation of three dimensional maps, which can then be used for navigation. With embedded systems becoming more autonomous, as they are included in unmanned aerial vehicles (UAVs) or robots, the navigation task becomes very important. To create a 3D map, multiple sensors that can measure ranges exist, the simplest are ultra-sonic sensors the more complex are radar or light detection and ranging (LiDAR) systems.

Another approach is to calculate the distances from two images of the same scene, which is known as stereo vision. Since size and weight matters, especially in small UAVs like quadcopters, the imaging system has to be as small and as lightweight as possible, which rules out the use of dedicated range sensors. Also the use of two cameras, as it is needed for stereo vision, increases both, weight and price of the system. An approach similar to stereo vision but using only one camera is known as Structure from Motion (SfM). For this approach two images are taken consecutively, where the camera is moved in between. Knowing the motion of the system the two images can be treated like a stereo vision problem. The motion can be learned from the system's sensors, e. g., the accelerometer and gyroscope of a quadcopter. If no such sensors are available the rotation and translation of the camera has to be estimated from the taken images.

Latter approach is discussed in this thesis to allow the creation of a 3D map on any embedded system without the need of dedicated sensors, besides a camera. Next to discussing the SfM approach an implementation using C++ and the OpenCV library is presented. The application is then tested on an NVIDIA Jetson TK1 development board that features the NVIDIA Tegra K1 SoC. To take advantage of the GPU, optimizations using CUDA are suggested and then compared to the baseline implementation.

1.1 Organization

Chapter 2 gives an overview on the notation used throughout the thesis and introduces the concept of homogeneous coordinates. Then the theoretical foundations of CV required for SfM are disclosed. These include the concept of features, their extraction, description and matching, as well as epipolar geometry. Chapter 3 explains the approach to SfM and discusses the steps to reconstruct a point cloud from a series of images, i. e., camera calibration, pose estimation and triangulation, in detail. The proposed algorithms are implemented and verified using MATLAB. Chapter 4 presents the implementation of SfM for the NVIDIA Jetson TK1 platform. For that matter, a

test application using the OpenCV library and several optimizations using CUDA are implemented. The performance gain by using the GPU is then analyzed and discussed. Finally, Chapter 5 concludes the thesis and gives an outlook on future work.

Chapter 2

Theoretical Foundations

This chapter deals with the theoretical foundations of CV that are needed for SfM. At first the notation used in this thesis will be stated and the concept of homogeneous coordinates is explained. The chapter will introduce the pinhole camera model and the mathematical basics of projective geometry, as well as providing a way to correct distortions in non-ideal cameras. Then the concept of features, their extraction, description and matching is presented. Finally epipolar geometry and its importance to multi-view geometry is depicted. Special attention is given to the Fundamental and Essential matrices, which are crucial to Structure from Motion.

2.1 Notation

Following the notation in [14], the following conventions are used throughout this work. Vectors $\boldsymbol{v} = (v_1, v_2, v_3)$ are lower case bold and – unless otherwise noted – column vectors. Matrices

$$\boldsymbol{M} = \begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \\ m_5 & m_6 \end{bmatrix} \tag{2.1}$$

are upper case bold and matrices and vectors can be concatenated so that

$$\boldsymbol{K} = \begin{bmatrix} \boldsymbol{M} & \boldsymbol{v} \end{bmatrix} = \begin{bmatrix} m_1 & m_2 & v_1 \\ m_3 & m_4 & v_2 \\ m_5 & m_6 & v_3 \end{bmatrix}. \tag{2.2}$$

Scalars S or s are mixed case and multiplications $s\boldsymbol{M}$ omit the multiplication sign $\cdot$ between factors. If homogeneous coordinates for points in 2D or 3D spaces are used, they are denoted as $\tilde{\boldsymbol{p}} = (x, y, z, w)$.

2.2 Homogeneous Coordinates

Homogeneous coordinates are widely used in computer graphics. By allowing the use of matrix multiplications for the transformation and projection of points, they simplify calculations in the field of projective geometry [2].

A point in an n-dimensional space has $n + 1$ homogeneous coordinates. The $n + 1$th value of the 3D point

$$\tilde{\boldsymbol{p}}_{3D} = (x, y, z, w) \tag{2.3}$$

is the weight w of the coordinates. If $w = 0$ the point is said to be at infinity and can be treated as positionless vector [14]. Given $w \neq 0$ the Euclidean coordinates of a point $\boldsymbol{p}_{3D}$ can be calculated by

$$\boldsymbol{p}_{3D} = (\frac{x}{w}, \frac{y}{w}, \frac{z}{w}). \tag{2.4}$$

Therefore an arbitrary Euclidean point $\boldsymbol{p} = (x_e, y_e, z_e)$ can be converted to homogeneous coordinates by simply extending the coordinates with weight $w = 1$

$$\tilde{\boldsymbol{p}} = (x_e, y_e, z_e, 1). \tag{2.5}$$

As mentioned before the advantage of this representation is the ability to calculate transformations using matrices. To translate a two-dimensional Euclidean point the equation $\boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{t}$ is used. When using homogeneous coordinates the same operation can be expressed by

$$\tilde{\boldsymbol{x}}' = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{t} \\ \boldsymbol{0}^T & 1 \end{bmatrix} \tilde{\boldsymbol{x}} \tag{2.6}$$

where $\boldsymbol{I}$ is the $N \times N$ identity matrix, $\boldsymbol{0}^T$ is a $1 \times N$ zero row-vector and $\boldsymbol{t}$ the $N \times 1$ translation vector. To rotate $\tilde{\boldsymbol{x}}$ by θ radians counter-clockwise about the origin, $\boldsymbol{I}$ can be substituted by a rotation matrix

$$\boldsymbol{R} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}. \tag{2.7}$$

Note that all rotation matrices $\boldsymbol{R}$ are orthonormal, i. e., $\boldsymbol{R}^T\boldsymbol{R} = \boldsymbol{I}$, and therefore the inverse is $\boldsymbol{R}^{-1} = \boldsymbol{R}^T$ [24, 14]. Further, the determinant of a rotation matrix is $det(\boldsymbol{R}) = 1$, which can be used to verify that a square matrix is in fact a rotation matrix. To scale the point, matrix $\boldsymbol{R}$ can be multiplied by a scalar s [2, 14]. The resulting transformation equation is given as

$$\tilde{\boldsymbol{x}}' = \begin{bmatrix} s\boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}^T & 1 \end{bmatrix} \tilde{\boldsymbol{x}}. \tag{2.8}$$

Using homogeneous coordinates multiple transformations can be chained together, which reduces the number of calculations. Furthermore the equations become easily solvable in both directions, since the transformation matrix is invertible, given the determinant is not zero.

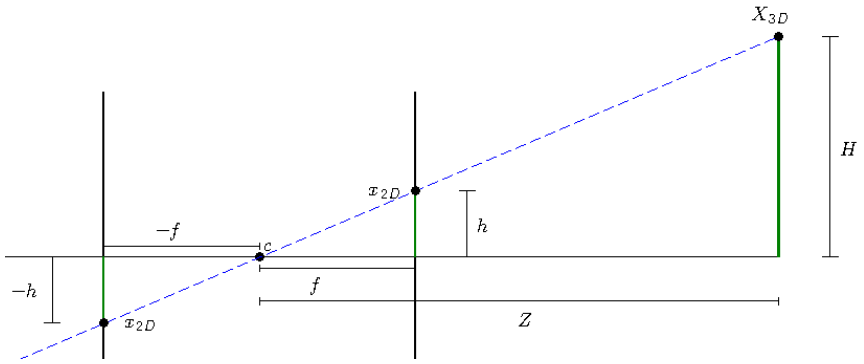


Figure 2.1: Pinhole camera model. A ray (blue line) emitted from a real world object X_{3D} travels through the center of projection c and then intersects with the image plane, creating the image x_{2D}. It can be seen that the proportions $h/f = H/Z$. The image plane can be drawn in front of the principal point or behind, the only change is the sign of h.

2.3 Camera Model

The knowledge of internal and external characteristics of a camera is essential to virtually any application of CV. The simplest model of a camera is the *pinhole camera*. This model consists of two planes, the image plane and the pinhole plane, which are separated by the focal length f. All rays emitted from a distant object travel through a hole, the center of projection c, in the center of the pinhole plane, and then are projected onto the image plane. The axis connecting the centers of the planes is called optical axis. Using basic geometry it can be shown that the proportion between the height of the image on the image plane h and the focal length f is equal to the height of the object in the real world H and its distance to the pinhole plane Z. The equation

$$-h = f\frac{H}{Z} \tag{2.9}$$

describes this relation. The negative sign can be eliminated by rearranging the image plane to be in front of the pinhole plane. As long as the focal length f is the same the two representations are equal, except that the image is not flipped [3]. Figure 2.1 illustrates the connection between the real world object and its image.

When considering a point $\boldsymbol{p} = (X, Y, Z)$ in a 3D space, the corresponding point on the image plane, given in pixels, is [3, 6]

$$\boldsymbol{p}_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} f_x \frac{X}{Z} \\ f_y \frac{Y}{Z} \end{bmatrix}. \tag{2.10}$$

Note that the focal lengths for x and y are not equal. This is based on the fact that pixels on an image sensor might not be square but rectangular

[3, 8]. The focal length f_x, measured in pixels, is then calculated by the product of the physical focal length, measured in millimeters, and the width of the individual sensor elements, measured in pixels per millimeter [3]. Using homogeneous coordinates the *perspective projection* can be written as

$$\tilde{\boldsymbol{p}}_i = \begin{bmatrix} x_i \\ y_i \\ w_i \end{bmatrix} = \begin{bmatrix} f_x & 0 & 0 & 0 \\ 0 & f_y & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}. \tag{2.11}$$

As can be seen in Equation (2.11), the depth information is lost in the 2D projection [14]. The Z value is converted to the weight w_i of the projection, therefore each 3D point on the line through the center of projection and $\tilde{\boldsymbol{p}}_i$ is mapped to this point.

To illustrate this, Figure 2.2 shows two homogeneous 2D points projected onto a 1D image plane. The ray through $\tilde{\boldsymbol{X}}_1 = (2, 4, 1)$ and the center of projection intersects the image plane at point $\tilde{\boldsymbol{x}}_{image} = (0.5, 1)$, since

$$\tilde{\boldsymbol{x}}_{image} = \begin{bmatrix} x \\ w \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 1 \end{bmatrix}, \tag{2.12}$$

where $f = 1$. Extending this ray, it can be seen that $\tilde{\boldsymbol{X}}_2 = (3, 6, 1)$ also lies on it. The projection can be written as

$$\tilde{\boldsymbol{x}}_{image} = \begin{bmatrix} x \\ w \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 1 \end{bmatrix}. \tag{2.13}$$

Therefore both 2D points are projected onto $\tilde{\boldsymbol{x}}_{image}$ and, based on the knowledge of this image alone, it can not be said which point on the ray is the original one, hence the depth information is lost.

Due to the manufacturing process the center of the image plane, i. e., the principal point, often is not the center of the image sensor. Therefore the two parameters c_x and c_y are introduced, describing the offset of the center of the chip to the optical axis in pixels. Extending Equation (2.11) with the offset leads to

$$\tilde{\boldsymbol{p}}_i = \begin{bmatrix} x_i \\ y_i \\ w_i \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}. \tag{2.14}$$

Additionally, a skew factor s can be added to the matrix. It encodes the possibility that the sensor's axis is not perpendicular to the optical axis [14].

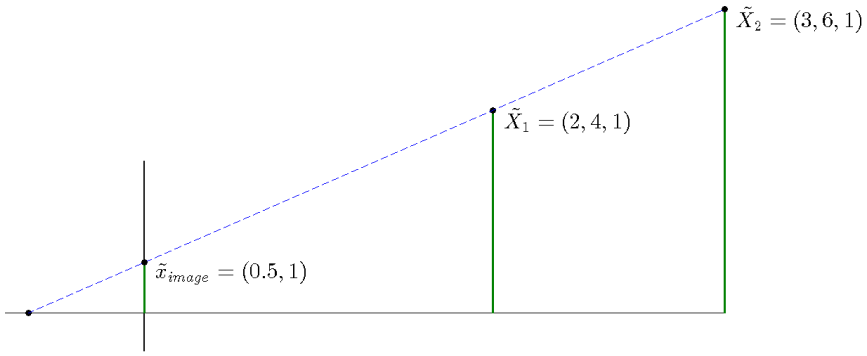


Figure 2.2: The homogeneous 2D points $\tilde{\boldsymbol{X}}_1$ and $\tilde{\boldsymbol{X}}_2$ are projected onto the image plane. The ray traveling through $\tilde{\boldsymbol{X}}_1$ also travels through $\tilde{\boldsymbol{X}}_2$. Therefore the image point $\tilde{\boldsymbol{x}}_{image}$ is not unique, hence the depth information of $\tilde{\boldsymbol{X}}_1$ and $\tilde{\boldsymbol{X}}_2$ is lost.

The final 3×4 matrix $\boldsymbol{P}$, i. e., the *projection matrix* or *camera matrix*, which projects a homogeneous 3D point onto a homogeneous 2D image point, can be written as

$$\boldsymbol{P} = \begin{bmatrix} f_x & s & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{2.15}$$

$\boldsymbol{P}$ in this form, however, can only be used if the camera's coordinate system and the world coordinate system are equal. To avoid this limitation $\boldsymbol{P}$ can be written as

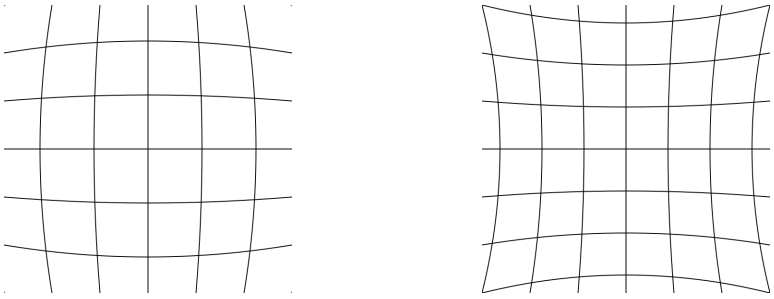
$$\boldsymbol{P} = \boldsymbol{K} \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \end{bmatrix}, \tag{2.16}$$

where

$$\boldsymbol{K} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.17}$$

is the *calibration matrix* describing the *camera intrinsics* and the concatenated matrix $\begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \end{bmatrix}$ describes the *camera extrinsics*. $\boldsymbol{R}$ and $\boldsymbol{t}$ are the rotation and translation of the camera's coordinate system to the world coordinate system, respectively.

Although the pinhole model is a useful model in theory, it has to be extended to adapt to actual cameras. To focus the beam of light going through the pinhole virtually all imaging devices use one or more lenses. Lenses introduce two types of distortions: *radial* and *tangential*. Former is based on properties of the lens, latter arises from inaccurate placement of the lens in relation to the image sensor.



(a) Barrel distortion. From [32]. **(b)** Pincushion distortion. From [33].

Figure 2.3: Two types of radial distortion. (a) Lenses with barrel distortion bend the rays further to the edge the greater the radius. (b) Lenses with pincushion distortion bend the rays further to the center the greater the radius.

2.3.1 Radial Distortion

Most lenses do not bend the light equally strong depending on the distance from the entry point of the ray to the center of the lens, subsequently called radius. Generally radial distortions can be classified into two types [15]: *barrel* and *pincushion* distortion. As can be seen in Figure 2.3, rays entering with a greater radius are bent further to the edge of the image when talking about barrel distortion. For pincushion distortion the rays are bent further to the center of the image the bigger the radius gets. Additionally to these two, *fish-eye* and *complex* distortions are common types of radial distortions. A *fish-eye* lens exhibits an extreme barrel distortion, and therefore, is suitable if a large field of view is desired, e. g., in surveillance or panoramic photography. A combination of barrel and pincushion distortion is called *complex* or *moustache* distortion.

Radial distortion can be approximated by low-order polynomials [3, 14]

$$\begin{aligned} \hat{x} &= x(1 + \kappa_1 r^2 + \kappa_2 r^4 + \kappa_3 r^6) \\ \hat{y} &= y(1 + \kappa_1 r^2 + \kappa_2 r^4 + \kappa_3 r^6), \end{aligned} \tag{2.18}$$

where $\hat{x}$ and $\hat{y}$ are the image coordinates after applying the distortion, x and y are the undistorted coordinates of the image after projection and $r^2 = x^2 + y^2$. The distortion coefficients κ_1, κ_2 and κ_3 have to be estimated. This is done by capturing and analyzing images of known patterns, usually checkerboards. The third coefficient often is $\kappa_3 = 0$, except for highly distortive lenses such as fish-eyes.

2.3.2 Tangential Distortion

In contrary to radial distortion, which is a property of the lens itself, tangential distortion results from an inaccurate placement of the lens during the manufacturing process. Image sensor and lens have to be in parallel, otherwise the points are displaced elliptically as a function of location and radius [3]. The distortion can then be approximated by [9]

$$\begin{aligned} d_x &= 2\kappa_4 xy + \kappa_5(r^2 + 2x^2) \\ d_y &= \kappa_4(r^2 + 2y^2) + 2\kappa_5 xy, \end{aligned} \tag{2.19}$$

where κ_4 and κ_5 are the coefficients describing the severity of the distortion.

Summing up, considering the two most common distortions, an image can be corrected by

$$\begin{bmatrix} \hat{x} \\ \hat{y} \end{bmatrix} = \underbrace{(1 + \kappa_1 r^2 + \kappa_2 r^4 + \kappa_3 r^6) \begin{bmatrix} x \\ y \end{bmatrix}}_{\text{radial distortion}} + \underbrace{\begin{bmatrix} 2\kappa_4 xy + \kappa_5(r^2 + 2x^2) \\ \kappa_4(r^2 + 2y^2) + 2\kappa_5 xy \end{bmatrix}}_{\text{tangential distortion}}. \tag{2.20}$$

The identification of the distortion coefficients $\kappa_1, \kappa_2, \kappa_3, \kappa_4, \kappa_5$ as well as the focal length f_x, f_y, the skew s and the offset c_x, c_y is called camera calibration and will be discussed in Section 3.1.

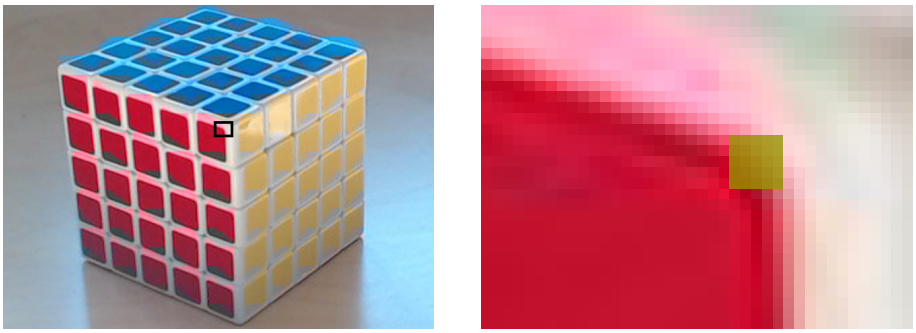
2.4 Feature Detection

Feature detection in CV is essential to compare two images. The applications using features range from object detection, over panoramic mosaics to 3D reconstruction. There are two kinds of features to be found in an image. The first kind is called *keypoint feature*, *feature point* or *corner* and describes a characteristic or outstanding area in the image. The other class are *edges*, which can be utilized as indicators for object boundaries and combined to form *curves* or *straight lines*. In respect to the application discussed in this thesis, feature points are the more important features.

Feature detection can be split into three stages [14]. At first the images have to be analyzed, to find distinctive points; this is called feature extraction. The feature description stage is responsible for assigning a mathematical expression to each feature, so that the third stage, i. e., feature matching, can compare feature points from different images. If the descriptors of two points match, a keypoint pair, that can be used to apply CV algorithms, is found.

2.4.1 Feature Extraction

When searching for distinctive points, the image is not analyzed pixel by pixel but rather using patches. An image patch is a window with the analyzed



(a) Rubik's cube **(b)** A 5×5 image patch

Figure 2.4: Using image patches for detection of gradients. (a) Rubik's cube. The black square indicates the section of the image shown in (b). (b) A 5×5 image patch is used to find distinctive feature points.

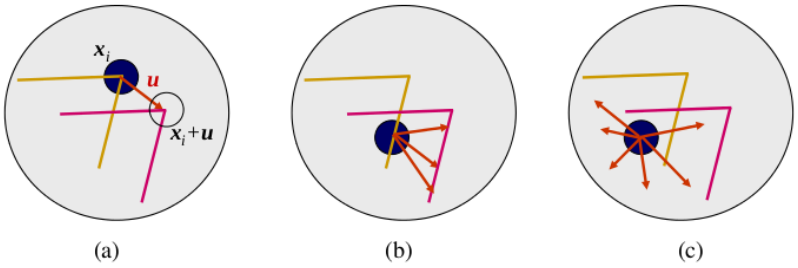


Figure 2.5: Distinctiveness of feature points in regard to the number of gradients. (a) two gradients; (b) one gradient; (c) no gradient. From [14].

pixel in the center. When talking about a 3×3 patch, it contains 9 pixel: the pixel of interest and every adjacent pixel. Figure 2.4b shows a 5×5 patch. It can clearly be seen that it is virtually impossible to find a distinctive feature using only one pixel. By using a patch, however, a gradient, i. e., changes in color or brightness, can be detected. A distinctive feature generally is distinguished by gradients in at least two different directions. When looking at one of the squares on the Rubik's cube in Figure 2.4a, the change in color between the sticker, e. g., the upper right red square, and the white cube can be detected. Every feature point lying on the edge of the sticker possesses one gradient, therefore the exact location of a feature point on this edge cannot be determined. A feature point at the corner of the sticker, on the other hand, can clearly be distinguished, since it shows two gradients. Any patch from the center of the sticker will have no gradients, and therefore, is not distinctive. These three types of distinctiveness are shown in Figure 2.5

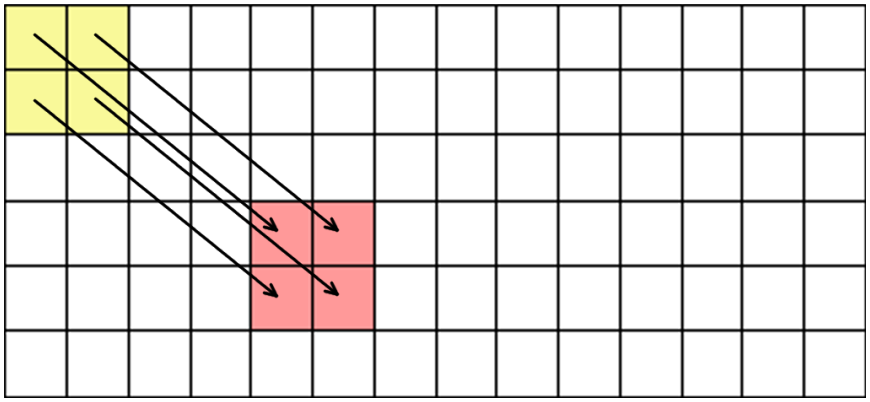


Figure 2.6: To a 2×2 patch P_0, marked yellow, the displacement vector $\boldsymbol{u} = (3, 4)$ is added, resulting in patch P_1, marked red. Each pixel of P_0 is moved 3 pixel down and 4 pixels to the right. The result is patch P_1.

To find an $N \times M$ image patch P_0, defined in image I_0, in another image I_1, the weighted sum of squared differences (WSSD) can be used. It is defined by

$$E_{WSSD}(\boldsymbol{u}) = \sum_{n=1}^{N}\sum_{m=1}^{M} w(\boldsymbol{x}_{n,m}) \left[I_1(\boldsymbol{x}_{n,m} + \boldsymbol{u}) - I_0(\boldsymbol{x}_{n,m})\right]^2, \tag{2.21}$$

where the summation is over all pixels $\boldsymbol{x}_{n,m}$ in patch P_0 and $w(\boldsymbol{x})$ is a spatially varying weighting function [13, 14]. The argument of E_{WSSD}, the displacement vector $\boldsymbol{u} = (u, v)$ describes the offset which is added to each pixel location in I_1.

Figure 2.6 shows $\boldsymbol{u} = (3, 4)$ and the 2×2 patch P_0 containing the pixels

$$P_0 = \left[\begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right]. \tag{2.22}$$

After adding the displacement vector to P_0, patch P_1 contains the pixels

$$P_1 = \left[\begin{bmatrix} 3 \\ 4 \end{bmatrix} \quad \begin{bmatrix} 3 \\ 5 \end{bmatrix} \quad \begin{bmatrix} 4 \\ 4 \end{bmatrix} \quad \begin{bmatrix} 4 \\ 5 \end{bmatrix} \right]. \tag{2.23}$$

The minimum of $E_{WSSD}(\boldsymbol{u})$, considering all possible $\boldsymbol{u}$, describes the most likely displacement vector and thereby the position of P_0 in I_1.

To measure the distinctiveness of a feature point, i. e., a patch, its WSSD, where $I_0 = I_1$, can be used. This is called auto-correlation function (AC) [13] or *surface* [14] and is defined as

$$E_{AC}(\Delta\boldsymbol{u}) = \sum_{n=1}^{N}\sum_{m=1}^{M} w(\boldsymbol{x}_{n,m}) \left[I_0(\boldsymbol{x}_{n,m} + \Delta\boldsymbol{u}) - I_0(\boldsymbol{x}_{n,m})\right]^2, \tag{2.24}$$

where $\Delta \boldsymbol{u}$ is a small displacement vector. If $\Delta \boldsymbol{u}$ exhibits a strong minimum at $\Delta \boldsymbol{u} = (0, 0)$, the feature point can be well localized. Equation (2.24) can be approximated by [13]

$$E_{AC}(\Delta \boldsymbol{u}) \approx \Delta \boldsymbol{u}^T \boldsymbol{A} \Delta \boldsymbol{u}, \tag{2.25}$$

where $\boldsymbol{A}$ is the auto-correlation matrix and can be written as

$$\boldsymbol{A} = w * \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}. \tag{2.26}$$

I_x and I_y describe the horizontal and vertical derivatives of the image, respectively. Each element is convoluted by a windowing function w. Because of the symmetry of the matrix, only 3 convolutions have to be calculated. The rank of $\boldsymbol{A}$ describes the type of features detected:

- $rank(\boldsymbol{A}) = 2$: interest point,
- $rank(\boldsymbol{A}) = 1$: edge,
- $rank(\boldsymbol{A}) = 0$: homogeneous region (no interest point).

When a Gaussian is used as window function instead of a simple sum, the detector becomes insensitive to rotations of the patch [14].

2.4.2 Feature Description

When describing a feature, a tradeoff between distinctiveness and generality must be made. On one hand the feature detection should be insensitive to translation, rotation and perspective changes, on the other hand the descriptor must be distinguishable from other features' descriptors. A more detailed discussion of some widely used descriptors can be found in [14]. These include multi-scale oriented patches (MOPS), scale invariant feature transform (SIFT), speeded up robust features (SURF), gradient location-orientation histogram (GLOH) and others.

2.4.3 Feature Matching

When matching features, their descriptors are compared and if they are equal or are within a given margin a match is found. This, however, can lead to false positives if the whole image is searched for matching features. The simplest strategy to avoid this is to use a threshold using an Euclidean distance metric, i. e., the maximal distance from the original feature, to exclude potentially matching features too far away from the original feature [14].

2.5 Epipolar Geometry

For the calculation of 3D points and camera pose from multiple views epipolar geometry is essential. It defines constraints between 2D points in two

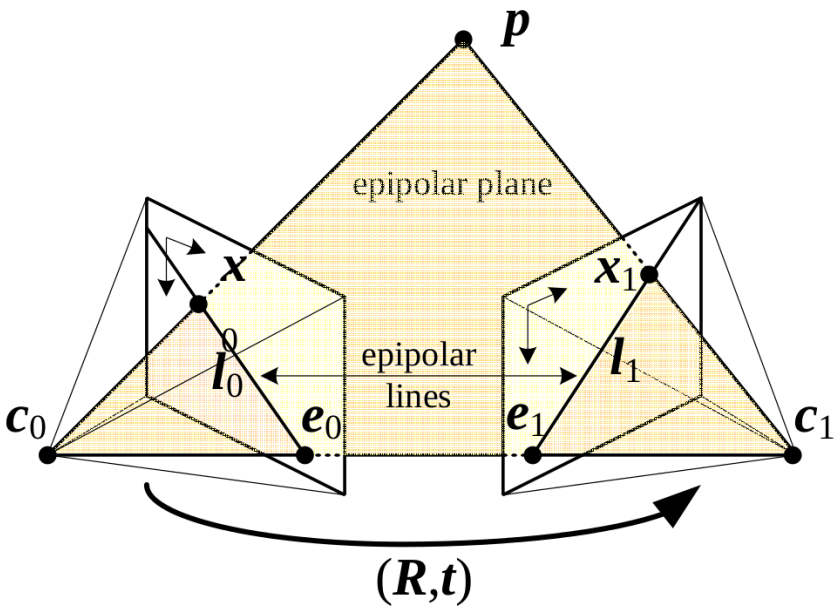


Figure 2.7: Epipolar geometry of two views. The camera centers $\boldsymbol{c}_0$ and $\boldsymbol{c}_1$ and the point in 3D space $\boldsymbol{p}$ span the epipolar plane. The intersection of image plane and epipolar plane gives the epipolar lines $\boldsymbol{l}_0$ and $\boldsymbol{l}_1$. The projections of point $\boldsymbol{p}$, i. e., $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$, are known to lie on $\boldsymbol{l}_0$ and $\boldsymbol{l}_1$, respectively. From [14].

views and their 3D correspondence. A point $\boldsymbol{p}$ in 3D space is seen in images I_0 and I_1 as point $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$, respectively. The ray through the camera center $\boldsymbol{c}_0$ and the point on the image plane $\boldsymbol{x}_0$ will also go through $\boldsymbol{p}$, the same holds for the ray through $\boldsymbol{c}_1$ and $\boldsymbol{x}_1$. Given the images are taken with pinhole cameras and no noise, both rays will intersect at point $\boldsymbol{p}$ as shown in Figure 2.7. The epipoles $\boldsymbol{e}_0$ and $\boldsymbol{e}_1$ are the intersections of line connecting the camera centers with the left and right camera's image plane, respectively. The epipoles, the centers of the cameras, the points on the image planes, and the point in 3D space are coplanar, spanning the epipolar plane. The line connecting $\boldsymbol{e}_0$ and $\boldsymbol{x}_0$, the epipolar line, is the projection of the ray through $\boldsymbol{x}_1$ and $\boldsymbol{p}$.

The *epipolar constraint* is defined as [8, 23]

$$\tilde{\boldsymbol{x}}_1^T \boldsymbol{F} \tilde{\boldsymbol{x}}_0 = 0, \tag{2.27}$$

where $\boldsymbol{F}$ is the 3×3 *Fundamental matrix*. This can be expanded to

$$\begin{aligned} f_{11}x_1x_0 + f_{12}x_1y_0 + f_{13}x_1 + \\ f_{21}y_1x_0 + f_{22}y_1y_0 + f_{23}y_1 + \\ f_{31}x_0 + f_{32}y_0 + f_{33} = 0, \end{aligned} \tag{2.28}$$

where f_{11} through f_{33} are entries in $\boldsymbol{F}$ and x_0, x_1 and y_0, y_1 are the coordinates of the points $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$, respectively. The resulting equation is

$$\begin{bmatrix} x_1x_0 & y_1x_0 & x_0 & x_1y_0 & y_1y_0 & y_0 & x_1 & y_1 & 1 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{bmatrix} = 0. \tag{2.29}$$

The equation can be solved using standard approaches if multiple 2D point pairs are known [19]:

$$\boldsymbol{A} \begin{bmatrix} f_{11} & f_{12} & f_{13} & f_{21} & f_{22} & f_{23} & f_{31} & f_{32} & f_{33} \end{bmatrix}' = \boldsymbol{A}\boldsymbol{f} = 0, \tag{2.30}$$

where $\boldsymbol{A}$ is a $N \times 9$ matrix

$$\boldsymbol{A} = \begin{bmatrix} x_{1,1}x_{1,0} & y_{1,1}x_{1,0} & x_{1,0} & x_{1,1}y_{1,0} & y_{1,1}y_{1,0} & y_{1,0} & x_{1,1} & y_{1,1} & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{N,1}x_{N,0} & y_{N,1}x_{N,0} & x_{N,0} & x_{N,1}y_{N,0} & y_{N,1}y_{N,0} & y_{N,0} & x_{N,1} & y_{N,1} & 1 \end{bmatrix} \tag{2.31}$$

containing N known 2D point pairs $x_{n,0} \leftrightarrow x_{n,1}$.

The Fundamental matrix has 9 elements but is only defined up to scale, meaning 8 point pairs are sufficient to find $\boldsymbol{F}$. However, the matrix also is singular, which can be used to eliminate another degree of freedom by setting the constraint $det(\boldsymbol{F}) = 0$. Therefore 7 point pairs are sufficient in general [8, 19]. The most common algorithms to approximate the Fundamental matrix, described in particular by [8], are the (normalized) *8-point* algorithm, the *Gold Standard* algorithm and the random sample consensus (RANSAC) method.

2.5.1 Essential Matrix

If the cameras, i. e., the intrinsics of each camera, are known, the epipolar constraint is given through the *Essential* matrix $\boldsymbol{E}$ as [14]

$$\tilde{\boldsymbol{x}}_{1C}^T \boldsymbol{E} \tilde{\boldsymbol{x}}_{0C} = 0, \tag{2.32}$$

where $\tilde{\boldsymbol{x}}_{0C}$ and $\tilde{\boldsymbol{x}}_{1C}$ are normalized image coordinates, given by [12]

$$\tilde{\boldsymbol{x}}_C = \boldsymbol{K}^{-1} \tilde{\boldsymbol{x}}. \tag{2.33}$$

The equation can be rewritten like Equation (2.27), resulting in a solvable linear system analog to Equation (2.30). If the Fundamental matrix already is known, the Essential matrix is [8]

$$\boldsymbol{E} = \boldsymbol{K}'\boldsymbol{F}\boldsymbol{K}. \tag{2.34}$$

From the Essential matrix rotation $\boldsymbol{R}$ and translation $\boldsymbol{t}$ between views can be extracted, since the matrix is composed as [8, 12, 14]

$$\boldsymbol{E} = [\boldsymbol{t}]_\times \boldsymbol{R}, \tag{2.35}$$

where $[\boldsymbol{t}]_\times$ denotes the cross product matrix, which is given as

$$\boldsymbol{t} \times \boldsymbol{n} = [\boldsymbol{t}]_\times \boldsymbol{n} = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \boldsymbol{n}. \tag{2.36}$$

An algorithm for finding $\boldsymbol{R}$ and $\boldsymbol{t}$ will be presented in Section 3.2.

Chapter 3

Structure from Motion

SfM describes the process of recovering depth information from two or more images taken successively from the same camera. The approach is similar to stereo vision with the difference that only one image at a time can be taken. For the successful estimation of a point in a 3D coordinate system two images from different angles, showing the same 3D point, are necessary. When using two cameras – as in stereo vision – the images can be captured at the exact same moment. A moving object would therefore result in a blurred image at most. SfM, however, has to capture the first image, then move the camera, then capture the second image. If an object in the captured scene moves during the transition of the camera, its 3D coordinates have changed. Therefore it is essential for SfM that the captured scene is rigid. Stereo vision does not have this limitation.

This chapter describes the algorithms and approaches to achieve SfM that are then implemented in MATLAB[1] for verification. At first the camera calibration procedure, i. e., the estimation of the camera intrinsics, will be explained. Section 3.2 will provide a way to estimate the rotation and translation of the second view in relation to the first view using keypoints found in both images and epipolar geometry. An approach for triangulation using an iterative method will be discussed in Section 3.3. Finally the approach will be verified using MATLAB and the results will be presented and compared to ground truth data.

3.1 Camera Calibration

Camera calibration is a crucial part in CV. As described in Section 2.3, the knowledge of the *camera intrinsics* is necessary to project a point in a 3D coordinate system onto an image point. The parameters estimated by the calibration are: the focal lengths f_x and f_y, the offset of the image center to the optical axis c_x and c_y, the skew factor s, the radial distortion

[1] The MATLAB version used is `R2012a Student Version`.

coefficients κ_1, κ_2 and κ_3 and the tangential distortion coefficients κ_4 and κ_5. Since the intrinsics are constant, the calibration has only to be performed once. Therefore, the calibration can be done beforehand and does not have to be implemented on the embedded system.

3.1.1 Camera

The camera used in this thesis is a *Logitech HD Pro Webcam C920* connected via Hi-Speed USB 2.0 [26]. It supports[2] videos up to 1920×1080 pixels, i. e., Full HD, using either uncompressed YUV, compressed MJPG or compressed H264 streams. When using the uncompressed format, 30 frames per second can only be achieved for resolutions smaller or equal 800×448 pixels. When capturing in Full HD, only 5 frames can be transferred per second. The compressed formats, however, can take images at 33 millisecond intervals, which equals 30 frames per second, even when using a Full HD resolution.

The camera is equipped with an autofocus, which has to be turned off manually. When using the *video4linux* driver this can be done with the command `v4l2-ctl -c focus_auto=0`. This is necessary, since the autofocus would change the focal length constantly, which would render the calibration of the camera useless. The resolution of the images matters as well, since some of the calibration parameters, e. g., focal length and center offset, are measured in pixels. Other automatically changing properties of the camera, e. g., auto-exposure and auto-white-balance, might interfere with feature detection. A change in light between two images could trigger the camera's firmware to adjust the brightness gain and, as a consequence, feature points could drastically change, hence no matching keypoints are detected. Within the scope of this thesis, however, the settings will be left in automatic mode, in spite of the potential problems, because the disadvantages of automatic adjustment are negligible in the static scenes discussed here.

3.1.2 Calibration Toolbox

To estimate the camera intrinsics the *Calibration Toolbox for* MATLAB by Jean-Yves Bouguet from the California Institute of Technology [16] is used. This toolbox provides scripts and functions to conduct a straight-forward calibration. For that matter a pattern, i. e., a checkerboard, has to be produced and images of it have to be taken from multiple angles. Figure 3.1 shows the 21 images, taken with a resolution of 1280×720 pixels, that were used for the calibration.

The corners of the pattern have to be extracted semi-automatically. For each image the outer four corners have to be clicked, the first point being

[2] The supported modes were queried using the tool `v4l2-ctl` from the *video4linux utilities* package `v4l-utils` in Debian wheezy.

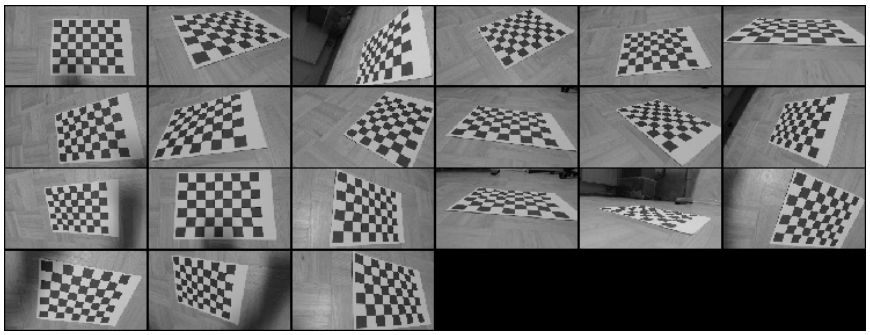


Figure 3.1: Images of the calibration pattern from 21 different angles. Each image has a resolution of 1280×720 pixels. The camera intrinsics can be estimated from them using the calibration toolbox.

the origin of the pattern's coordinate system, the toolbox then can detect the remaining corners. At first the corner-finder counts the number $n_x \times n_y$ of squares in x and y directions, respectively. The rectangle spanned by the four outer corners can then be divided into $n_x \times n_y$ squares and the corner detection algorithm will detect the change between black and white at the estimated corners of these squares. The resulting image can be seen in Figure 3.2, where the corners are marked with blue squares. The origin of the pattern's coordinate system in the top left is marked with O. The origin has to be the same corner in each image, since it is later used to calculate the position of the camera. When calibrating fish-eye lenses, the corner-finder might not work. In that case a pre-estimated distortion factor has to be entered to detect the corners correctly.

Once all corners in all images are extracted, the calibration can be started. The camera intrinsics to be estimated can be configured; by default the focal lengths f_x and f_y, the offset of the camera center c_x and c_y and the distortion coefficients κ_1, κ_2, κ_4 and κ_5 are estimated. To avoid confusion it should be noted that the 5×1 vector $\boldsymbol{kc}$, that is created by the toolbox, contains the distortion parameters in the following order $\boldsymbol{kc} = \begin{bmatrix} \kappa_1 & \kappa_2 & \kappa_4 & \kappa_5 & \kappa_3 \end{bmatrix}^T$, the fifth element being the coefficient of the 6^{th} order term of radial distortion.

The calibration yields the camera intrinsics and, as a byproduct, the camera extrinsics, that can be seen in Figure 3.3, of the analyzed images. For the first calibration run the estimation of all parameters was enabled. The result can be seen in Table 3.1. The values of the center offset, c_x and c_y, show that the chip's center is close to but not exactly at the optical axis.

It should be noted that the deviation of the distortion coefficients κ_3 and κ_4 are greater than the values itself and the deviation of κ_5 is nearly the same as the value. This jitter negatively affects the estimation process of the other parameters, especially the deviation of the center offset. Thus the

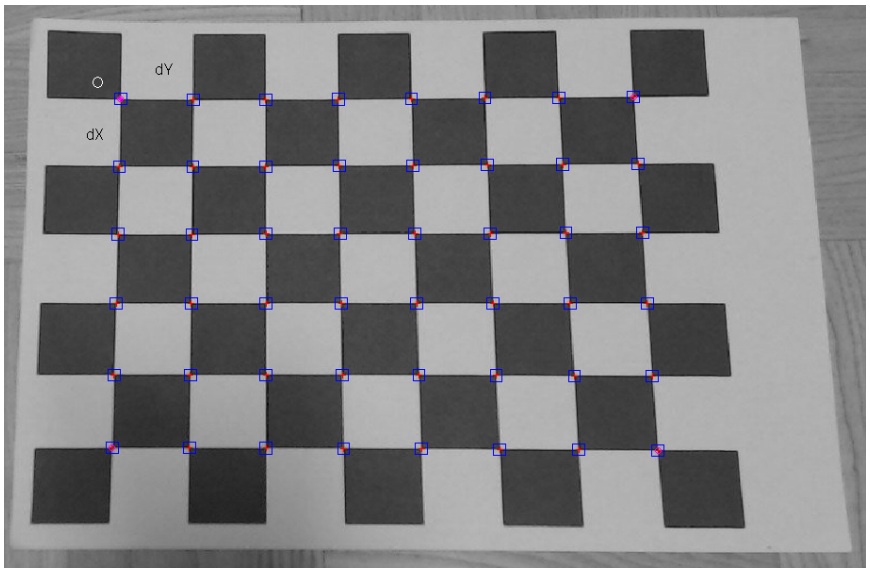


Figure 3.2: After clicking the four outer corners, the first click being the origin in the top left marked with O, the corner finder of the calibration toolbox finds the inner corners, marked with blue squares.

estimation of the coefficients for the tangential distortion and the 6^{th} order term of the radial distortion are disabled for the final calibration. Therefore

Table 3.1: Results of calibration with all estimations enabled: The values of the distortion factors κ_3 through κ_5 are smaller than, or equal, their deviation. This negatively affects the estimation of the other parameters, especially the center offset.

Description	Variable	Value	Deviation
Focal Length	f_x	922.33603	$\pm$2.30656
	f_y	920.15413	$\pm$2.31955
Center Offset	c_x	631.04752	$\pm$4.46891
	c_y	353.96906	$\pm$3.36721
Skew	s	0.00150	$\pm$0.00099
Radial Distortion	κ_1	0.11081	$\pm$0.01880
	κ_2	-0.23776	$\pm$0.12605
	κ_3	0.07775	$\pm$0.24177
Tangential Distortion	κ_4	-0.00018	$\pm$0.00140
	κ_5	-0.00203	$\pm$0.00201

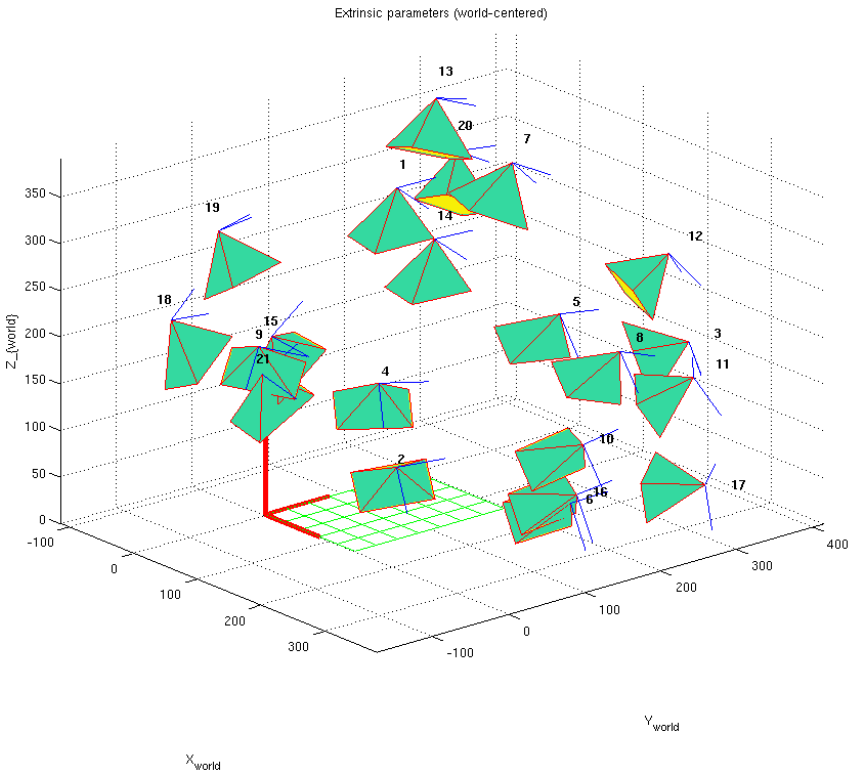


Figure 3.3: The camera extrinsics as estimated by the calibration toolbox. The cameras are drawn in green with the yellow face indicating the direction of view. The calibration pattern's upper left corner is the origin.

the camera's distortion can be modeled using only radial distortion from Equation (2.18) with $\kappa_3 = 0$.

It can also be seen that the skew factor is very small. Converted to degrees, the pixels are at 89.91419 ± 0.05673 degrees to the optical axis, which shows that the assumption that the sensor is mounted perpendicular is quite feasible. However, other than the distortion parameters, disabling the skew does not have an impact on the accuracy of the remaining parameters, and therefore, it will be estimated in the final calibration.

The results of the final calibration in Table 3.2 show that the deviation of the center offset has decreased by about 3.2 and 1.4 pixels for the x and y axis, respectively. The deviations of the two remaining distortion parameters κ_1 and κ_2 have been improved as well.

Table 3.2: Results of final calibration: Tangential distortion and last coefficient of radial distortion are not estimated. This leads to a more accurate estimation of the remaining parameters and simplifies the distortion model

Description	Variable	Value	Deviation
Focal Length	f_x	922.45051	±2.19234
	f_y	920.41192	±2.22917
Center Offset	c_x	635.39454	±1.25605
	c_y	354.20846	±1.93584
Skew	s	0.00147	±0.00098
Radial Distortion	κ_1	0.10620	±0.00873
	κ_2	−0.20464	±0.02669
	κ_3	not estimated	
Tangential Distortion	κ_4	not estimated	
	κ_5	not estimated	

3.2 Pose Estimation

The movement of the camera, i. e., rotation $\boldsymbol{R}$ and translation $\boldsymbol{t}$, has to be determined from the keypoints. This is achieved using epipolar geometry and the Essential matrix, as discussed in Section 2.5. The Essential matrix, however, only describes the relative movement between two views. Therefore the first view is assumed to be at the origin, hence the first camera matrix is

$$\boldsymbol{P}_0 = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{3.1}$$

The second camera matrix $\boldsymbol{P}_1$ can then be deduced using the Fundamental and Essential matrices.

The Fundamental matrix is calculated by the Matlab function `fundmatrix` from [25], taking two corresponding vectors of keypoints and returning the Fundamental matrix $\boldsymbol{F}$ and the epipoles $\boldsymbol{e}_0$ and $\boldsymbol{e}_1$. The algorithm applied is the normalized 8-point algorithm described in [8]. The Essential matrix is then derived using Equation (2.34), where $\boldsymbol{K}_0 = \boldsymbol{K}_1$ is the calibration matrix retrieved in Section 3.1.2. The Essential matrix can be decomposed using singular value decomposition (SVD), i. e.,

$$\boldsymbol{E} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T. \tag{3.2}$$

Using Equation (2.35) and the ansatz from [8] and [14], the decomposition

can be written as

$$\boldsymbol{E} = [\boldsymbol{t}]_\times \boldsymbol{R} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T = \boldsymbol{S}\boldsymbol{Z}\boldsymbol{R}_{90}\boldsymbol{S}^T\boldsymbol{R}, \tag{3.3}$$

where the left handed side can be deduced as followed.

According to [8, p. 257]

> A 3×3 matrix is an Essential matrix if and only if two of its singular values are equal, and the third is zero.

Therefore – and since the Essential matrix is only defined up to scale, singular values of one can be assumed – the SVD yields

$$\boldsymbol{\Sigma} = \boldsymbol{Z} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{3.4}$$

Further, the cross product matrix $[\boldsymbol{t}]_\times$ can be interpreted as an operation that projects a vector onto a set of orthogonal vectors rotated by 90 degrees, while the vector itself is eliminated [14]. Considering the operations for the expression

$$[\boldsymbol{t}]_\times = \boldsymbol{S}\boldsymbol{Z}\boldsymbol{R}_{90}\boldsymbol{S}^T, \tag{3.5}$$

with the rotation expressed as

$$\boldsymbol{R}_{90} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.6}$$

it can be shown that the rightmost column vector of $\boldsymbol{S}$ is eliminated, and therefore, has to be the translation $\boldsymbol{t}$ [14]:

$$[\boldsymbol{t}]_\times = \boldsymbol{S}\boldsymbol{Z}\boldsymbol{R}_{90}\boldsymbol{S}^T = \begin{bmatrix} \boldsymbol{s}_0 & \boldsymbol{s}_1 & \boldsymbol{t} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{s}_0^T \\ \boldsymbol{s}_1^T \\ \boldsymbol{t}^T \end{bmatrix}. \tag{3.7}$$

Comparing the result of the SVD, $\boldsymbol{E} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$, with the assumption $\boldsymbol{E} = \boldsymbol{S}\boldsymbol{Z}\boldsymbol{R}_{90}\boldsymbol{S}^T\boldsymbol{R}$ from Equation (3.3) shows that

$$\boldsymbol{S} = \boldsymbol{U}, \tag{3.8}$$

$$\boldsymbol{Z} = \boldsymbol{\Sigma} \tag{3.9}$$

and

$$\boldsymbol{R}_{90}\boldsymbol{S}^T\boldsymbol{R} = \boldsymbol{V}^T, \tag{3.10}$$

hence

$$\boldsymbol{R} = \boldsymbol{S}\boldsymbol{R}_{90}^T\boldsymbol{V}^T. \tag{3.11}$$

Inverting $\boldsymbol{S}$ and $\boldsymbol{R}_{90}$ is not necessary, since both are orthogonal matrices for which $\boldsymbol{M}^{-1} = \boldsymbol{M}^T$ applies. However, $\boldsymbol{S}$ or $\boldsymbol{V}^T$ may not be rotation matrices, leading to $\boldsymbol{R}$ not being a rotation matrix. This can be checked by analyzing the matrix: for a rotation matrix applies $det(\boldsymbol{M}) = 1$ [31]. If $\boldsymbol{R}$ is not a rotation matrix, the sign of $\boldsymbol{U}$ is flipped, which still is valid, since $-\boldsymbol{E} = -\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ and $\boldsymbol{E}$ is defined only up to scale. This also leads to $\boldsymbol{t}$ being only defined up to scale, meaning the sign of the translation is unknown, as is the direction of the rotation $\boldsymbol{R}_{90}$, leading to four possible combinations for the second *camera matrix* $\boldsymbol{P}_1 = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \end{bmatrix}$, namely [8, 12, 14]

$$\begin{aligned} \boldsymbol{P}_1 = &\begin{bmatrix} \boldsymbol{U}\boldsymbol{R}_{90}\boldsymbol{V}^T & \boldsymbol{u}_3 \end{bmatrix} \\ \text{or } &\begin{bmatrix} \boldsymbol{U}\boldsymbol{R}_{90}^T\boldsymbol{V}^T & \boldsymbol{u}_3 \end{bmatrix} \\ \text{or } &\begin{bmatrix} \boldsymbol{U}\boldsymbol{R}_{90}\boldsymbol{V}^T & -\boldsymbol{u}_3 \end{bmatrix} \\ \text{or } &\begin{bmatrix} \boldsymbol{U}\boldsymbol{R}_{90}^T\boldsymbol{V}^T & -\boldsymbol{u}_3 \end{bmatrix}, \end{aligned} \tag{3.12}$$

where $\boldsymbol{u}_3$ is the rightmost column of $\boldsymbol{U}$.

To obtain the correct camera matrix, all four possible solutions are tested. For that purpose a set of keypoints has to be triangulated, as described in Section 3.3. Figure 3.4 shows the four combinations for $\boldsymbol{P}_1$ and a triangulated point $\boldsymbol{p}$. The point is calculated by intersecting the two rays through the centers and the feature point on the image plane of each view. The point's position, i. e., its *chirality* [14], decides whether the combination is valid. Note that only in Figure 3.4a a solution for $\boldsymbol{P}_1$ is found, since it is the only combination for which $\boldsymbol{p}$ lies in front of both views.

Additionally, the 3D points are reprojected using $\boldsymbol{P}_0$ and $\boldsymbol{P}_1$ and their reprojection error is determined. Note that $\boldsymbol{P}_0$ and $\boldsymbol{P}_1$ have to be multiplied by $\boldsymbol{K}_0$ and $\boldsymbol{K}_1$, respectively. Otherwise only normalized image coordinates $\boldsymbol{x}_C$ can be computed. If the majority of the points is in front of both cameras and the mean reprojection error is below a specified threshold, the camera matrix $\boldsymbol{P}_1$ is assumed to be correct. In the discussed implementation the threshold is 75 percent of the points are in front of both cameras with a mean reprojection error of less than 50 pixels.

3.2.1 Adding Views

While the initial pose estimation has to be based on a set of matching feature points alone, additional views can be added using the knowledge of previously triangulated points in the 3D space. The problem of finding the camera's rotation and translation from a set of known 3D-to-2D point correspondences is known as Perspective-n-Point (PnP). To find such correspondences, the features of the additional view have to be matched with features from previous views that already have been triangulated. The matched keypoints can then be attributed to points in the 3D space.

The correspondence between 2D and 3D points is expressed by

$$\tilde{\boldsymbol{x}} = \boldsymbol{P}\tilde{\boldsymbol{p}}, \tag{3.13}$$

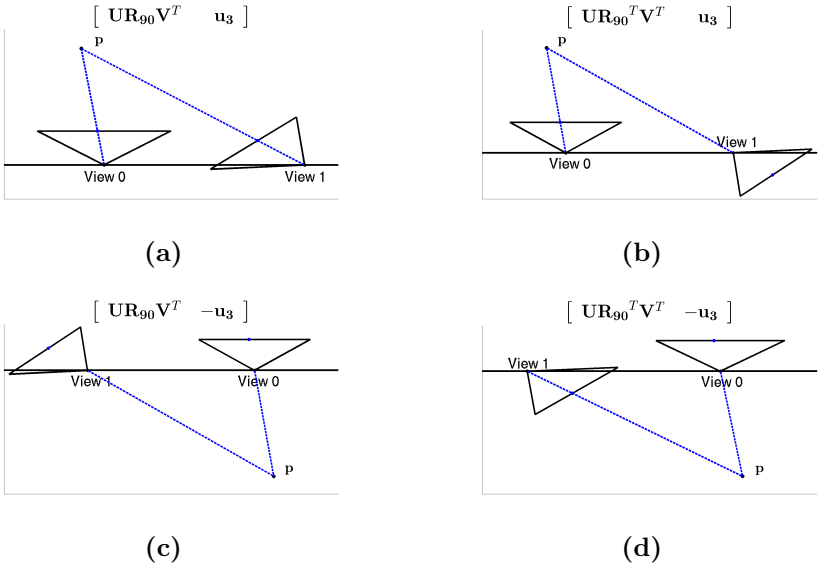


Figure 3.4: Images (a) through (d) show the four possible combinations for $\boldsymbol{P}_1$. Note that only (a) is a valid solution. The point $\boldsymbol{p}$ is found by intersecting the rays through the camera's center and the feature point on the image plane, marked in blue. In (b), (c) and (d) point $\boldsymbol{p}$ is not in front of both views.

i. e., perspective projection, where $\tilde{\boldsymbol{p}} = (X, Y, Z, 1)$ is a 3D-point and $\tilde{\boldsymbol{x}} = (u, v, 1)$ is the corresponding image point. The 3×4 projection matrix $\boldsymbol{P}$ has twelve degrees of freedom. Each 3D-to-2D correspondence provides 2 expressions, therefore 6 points suffice for the calculation of $\boldsymbol{P}$. Equation 3.13 can be expanded and rearranged as linear system

$$\boldsymbol{A}\boldsymbol{q} = 0, \tag{3.14}$$

where $\boldsymbol{q}$ is a 12×1 vector containing the elements of $\boldsymbol{P}$ and $\boldsymbol{A}$ is a $2n$×12 matrix, where n is the number of correspondences, i. e.,

$$\boldsymbol{A} = \begin{bmatrix} X_1 & Y_1 & Z_1 & 1 & 0 & 0 & 0 & 0 & -X_1u_1 & -Y_1u_1 & -Z_1u_1 & -u_1 \\ 0 & 0 & 0 & 0 & X_1 & Y_1 & Z_1 & 1 & -X_1v_1 & -Y_1v_1 & -Z_1v_1 & -v_1 \\ X_2 & Y_2 & Z_2 & 1 & 0 & 0 & 0 & 0 & -X_2u_2 & -Y_2u_2 & -Z_2u_2 & -u_2 \\ 0 & 0 & 0 & 0 & X_2 & Y_2 & Z_2 & 1 & -X_2v_2 & -Y_2v_2 & -Z_2v_2 & -v_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ X_n & Y_n & Z_n & 1 & 0 & 0 & 0 & 0 & -X_nu_n & -Y_nu_n & -Z_nu_n & -u_n \\ 0 & 0 & 0 & 0 & X_n & Y_n & Z_n & 1 & -X_nv_n & -Y_nv_n & -Z_nv_n & -v_n \end{bmatrix}, \tag{3.15}$$

where $\boldsymbol{p}_i = (X_i, Y_i, Z_i)$ and $\boldsymbol{x}_i = (u_i, v_i)$ form a 2D-to-3D correspondence.

The solution to this equation cannot be determined exactly if the correspondences contain noise. In that case [8] suggests the direct linear transformation (DLT) algorithm, whereas [10] provides the efficient Perspective-n-Point (ePnP) algorithm.

3.3 Triangulation

The problem of finding a point $\boldsymbol{p}$ in a 3D coordinate system from its projection onto two points $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$ in different images is known as triangulation. According to epipolar geometry, as discussed in Section 2.5, it is known that the epipolar line in the left view is the projection of the ray through $\boldsymbol{p}$ and $\boldsymbol{x}_1$. The same applies to the epipolar line in the right image and the ray through $\boldsymbol{p}$ and $\boldsymbol{x}_0$. Therefore the position of $\boldsymbol{p}$ can be determined by intersecting the two rays. Without noise this is a trivial problem. If, however, noise is present in the images, the rays are not guaranteed to cross, and therefore, the best possible solution has to be found. In [7] several methods to solve the triangulation problem are discussed. *Linear triangulation*, the most common method, is based on Equation (2.11), i. e., perspective projection. It is known that

$$\tilde{\boldsymbol{x}}_0 = \boldsymbol{P}_0 \tilde{\boldsymbol{p}} \tag{3.16}$$

and

$$\tilde{\boldsymbol{x}}_1 = \boldsymbol{P}_1 \tilde{\boldsymbol{p}}, \tag{3.17}$$

where $\tilde{\boldsymbol{p}} = (X, Y, Z, W)$ is the 3D point to be found. Assuming $\tilde{\boldsymbol{x}}_0 = (u_0, v_0, w_0)$, Equation (3.16) can be expanded to

$$\begin{aligned} p_{11}X + p_{12}Y + p_{13}Z + p_{14} &= w_0 u_0 \\ p_{21}X + p_{22}Y + p_{23}Z + p_{24} &= w_0 v_0 \\ p_{31}X + p_{32}Y + p_{33}Z + p_{34} &= w_0, \end{aligned} \tag{3.18}$$

where p_{ij} are elements of $\boldsymbol{P}_0$. Using the third equation to express w_0 in the other two gives

$$\begin{aligned} (p_{11} - u_0 p_{31})X + (p_{12} - u_0 p_{32})Y + (p_{13} - u_0 p_{33})Z &= u_0 p_{34} - p_{14} \\ (p_{21} - v_0 p_{31})X + (p_{22} - v_0 p_{32})Y + (p_{23} - v_0 p_{33})Z &= v_0 p_{34} - p_{24}. \end{aligned} \tag{3.19}$$

Equation (3.17) can be expanded similarly and leads to

$$\begin{aligned} (q_{11} - u_1 q_{31})X + (q_{12} - u_1 q_{32})Y + (q_{13} - u_1 q_{33})Z &= u_1 q_{34} - q_{14} \\ (q_{21} - v_1 q_{31})X + (q_{22} - v_1 q_{32})Y + (q_{23} - v_1 q_{33})Z &= v_1 q_{34} - q_{24}, \end{aligned} \tag{3.20}$$

where q_{ij} are elements of $\boldsymbol{P}_1$. Equations (3.19) and (3.20) can be combined to give an overdetermined linear system in the form $\boldsymbol{Ax} = \boldsymbol{b}$ [1]

$$\begin{bmatrix} p_{11} - u_0 p_{31} & p_{12} - u_0 p_{32} & p_{13} - u_0 p_{33} \\ p_{21} - v_0 p_{31} & p_{22} - v_0 p_{32} & p_{23} - v_0 p_{33} \\ q_{11} - u_1 q_{31} & q_{12} - u_1 q_{32} & q_{13} - u_1 q_{33} \\ q_{21} - v_1 q_{31} & q_{22} - v_1 q_{32} & q_{23} - v_1 q_{33} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} u_0 p_{34} - p_{14} \\ v_0 p_{34} - p_{24} \\ u_1 q_{34} - q_{14} \\ v_1 q_{34} - q_{24} \end{bmatrix}, \tag{3.21}$$

that can be solved using standard approaches.

Since the expression being minimized, i. e., $||\boldsymbol{Ax} - \boldsymbol{b}||$, has no geometric meaning, [7] suggests weighting the equations by multiplying $1/w_0$ to the first two rows of $\boldsymbol{A}$ and $\boldsymbol{b}$, and $1/w_1$ to the other rows, i. e., the rows that originate from the second image. The weights are, according to the third line in Equation (3.18),

$$w_0 = p_{31}X + p_{32}Y + p_{33}Z + p_{34} \tag{3.22}$$

and

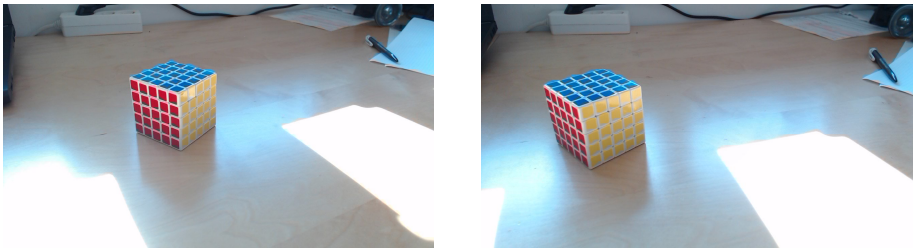
$$w_1 = q_{31}X + q_{32}Y + q_{33}Z + q_{34}. \tag{3.23}$$

The problem is then solved iteratively until the change in the weights is below a predefined threshold. As stated in [7], the threshold can be chosen freely and is 10^{-9} for the implementation discussed here. Point $\boldsymbol{p}$ will usually converge within a few iterations, which is why [7] suggests ten iterations at most, before a fallback method is used. Since the weights depend on $\boldsymbol{p}$, they are initialized with $w_0 = w_1 = 1$ for the first iteration.

3.4 Verification

To verify the approach to SfM presented in this chapter, the discussed algorithms are implemented in MATLAB and tested with images taken by the camera introduced in Section 3.1.1 that is later used on the embedded system. To test the approach, the images shown in Figure 3.5 are used. They are imported into MATLAB using the functionality of the calibration toolbox [16]. To simplify the visualization of the cube, the keypoints needed for pose estimation and triangulation have been extracted manually. By doing so, each keypoint can be identified and a color can be assigned to it, leading to a colored visualization of the calculated point cloud. The extracted keypoints can be seen in Figure 3.6.

As described in Section 3.2, the Fundamental matrix can be extracted from the keypoints and can then be converted to the Essential matrix using the known camera intrinsics from Section 3.1. The camera matrices $\boldsymbol{P}_0$ and $\boldsymbol{P}_1$ can then be derived from $\boldsymbol{E}$ using the SVD method. Note that $\boldsymbol{P}_0$ is the origin and the position of $\boldsymbol{P}_1$ is relative to it. Knowing the camera matrices, each keypoint can be triangulated and the reprojection error can be



(a) View 1 **(b)** View 2

(c) View 3 **(d)** View 4

Figure 3.5: The different views of a Rubik's cube that are used to verify the SfM approach. Note that the scene is rigid and the camera moved to the right between (a) and (b). Images (c) and (d) were captured after the scene was altered. However, since the feature detection is done manually and the object of interest, i. e., the Rubik's cube, itself has not changed, all images can be used together.

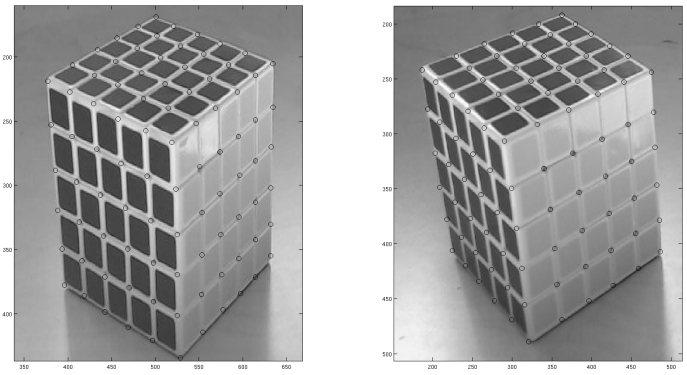


Figure 3.6: The features used to test the SfM approach. Each keypoint is extracted manually and assigned to a color.

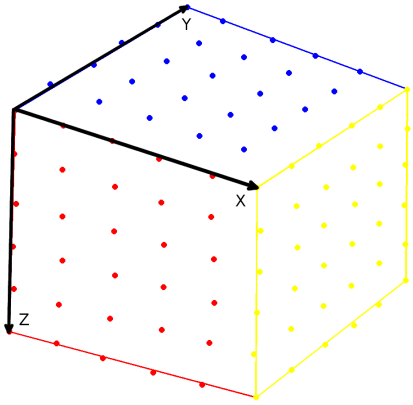


Figure 3.7: The created point cloud for the two images of the Rubik's cube. The points are colored corresponding to their keypoints. For easier perception, the edges of the cube are inserted.

determined. For this implementation the reprojection error has to be below 15 pixels to add a 3D point to the created point cloud. The mean reprojection error after applying the algorithms to the views in Figure 3.5a and 3.5b, is 5.8 pixels and all 91 processed keypoints are added to the point cloud, illustrated in Figure 3.7. The Rubik's cube is known to have a side length of 63.7 millimeters. To compare the size of the reconstructed cube with the original, a reference length in the images has to be known, since the model is only defined up to scale. To find a scale factor, the average horizontal and vertical distance between keypoints is calculated. Since it is known that each keypoint is the corner of one of the small colored cubes, subsequently called subcubes, of the Rubik's cube, the distance of the keypoints should be the side length of one subcube. Since it is a 5×5 Rubik's cube, a subcube's side has to have a length of 12.74 millimeters. For the calculation of the average side length of the reconstructed subcubes, only 3D points with a reprojection error less than five pixels are taken into account. This leads to a side length of 0.1120 units, and therefore, a scale factor of $s = \frac{12.74}{0.1120} = 113.75$. When multiplying the coordinates of each point with s, the reconstructed cube has side lengths of 78.3503×66.6941×54.8164 millimeters. Where the x axis is the red/blue edge, the y axis is the blue/backside edge and the z axis is the red/backside edge.

The same procedure is applied to different pairs of images. Table 3.3 shows the result of the SfM approach, when moving the camera from view i to view j, according to the first column. The mean reprojection error during triangulation is referred to in the second column. The results show that the deviations vary greatly. Especially the two combinations of views 2 and 3 result in a high inaccuracy, since the motion of the camera between these

Table 3.3: Measured dimensions (in millimeters) of reconstructed cubes and their deviations from the original. For dimensions marked with *, the maximal reprojection error of a 3D point considered for averaging is adopted to ten pixels. The last row shows the dimensions measured after filtering the views and taking the point-wise average.

Views	Reprojection Error	Dimensions	Deviation		
			X	Y	Z
$1 \rightarrow 2$	5.8258	78.35×54.82×66.69	14.65	−8.88	2.99
$1 \rightarrow 3$	1.0859	56.57×65.92×65.34	−7.13	2.22	1.64
$1 \rightarrow 4$	11.2161	52.89×61.39×60.52*	−10.81	−2.31	−3.18
$2 \rightarrow 1$	5.7965	79.10×54.95×66.63	15.40	−8.75	2.93
$2 \rightarrow 3$	2.4762	3.48×34.71×42.58	−60.22	−28.99	−21.12
$2 \rightarrow 4$	3.8028	65.15×69.26×52.82	1.45	5.56	−10.88
$3 \rightarrow 1$	0.7431	58.69×64.59×64.52	−5.01	0.89	0.82
$3 \rightarrow 2$	2.4108	18.11×56.43×91.83	−45.59	−7.27	28.13
$3 \rightarrow 4$	2.5225	51.69×59.55×65.93	−12.01	−4.15	2.23
$4 \rightarrow 1$	11.5657	56.74×64.29×62.95*	−6.96	0.59	−0.75
$4 \rightarrow 2$	3.7070	66.67×69.76×53.25	2.97	6.06	−10.45
$4 \rightarrow 3$	2.5417	51.28×59.89×66.97	−12.42	−3.81	3.27
Average after filtering	-	62.77×63.34×62.76	−0.93	−0.36	−0.94

views is minimal, and therefore, the rays connecting the image points with the 3D point are in parallel. This can lead to the inaccurate or incorrect triangulation of the 3D point.

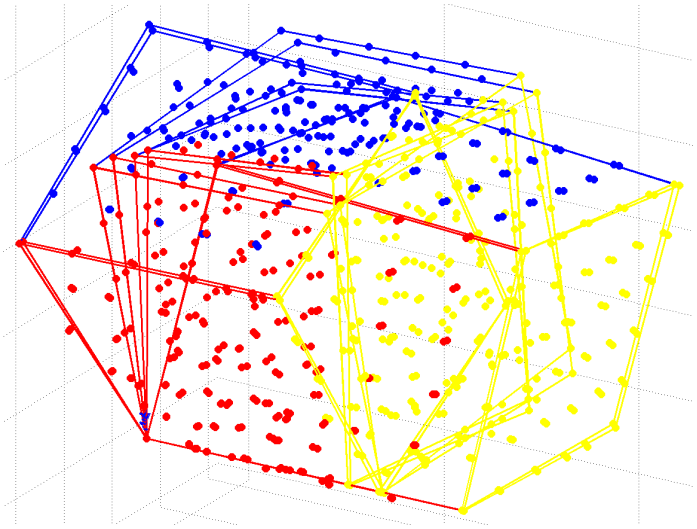
To compare the reconstructed cubes from different views, they have to be translated, scaled and rotated. The Rubik's cube of each view combination is moved to the origin, then rotated, so that the lower left corner of the red side is the origin and the axis are in parallel to the corresponding sides of the cube. Finally it is scaled, so that the bottom edge of the red side is the size of the real world object. To achieve this, each point in the point cloud $\tilde{\boldsymbol{X}}$ is transformed using

$$\tilde{\boldsymbol{X}}_o = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{R}\boldsymbol{t} \\ \boldsymbol{0}_3^T & 0 \end{bmatrix} \tilde{\boldsymbol{X}}, \tag{3.24}$$

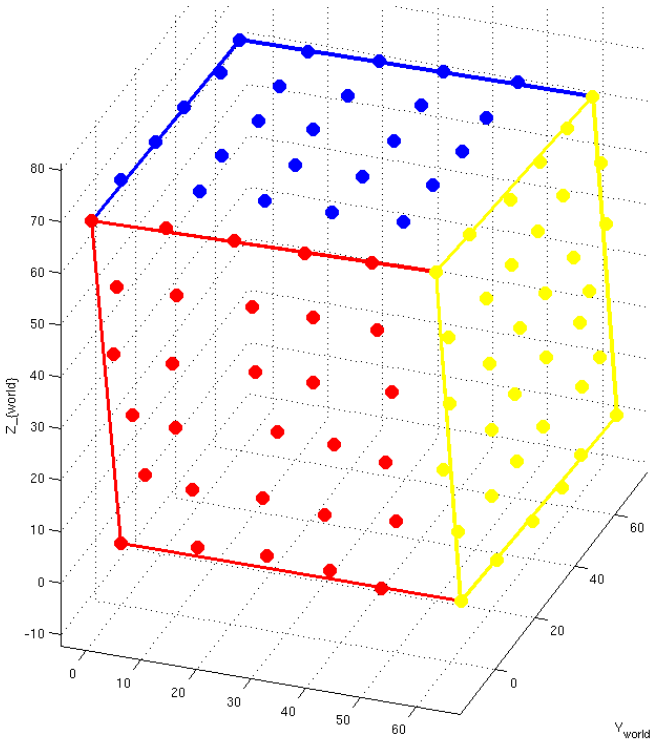
where $\boldsymbol{R} = s\boldsymbol{R}_z\boldsymbol{R}_x$ is the rotation to align the bottom edge of the cube with the x axis, then align the red/backside edge with the z axis and finally scale the point by a factor s. The translation $\boldsymbol{t}$ is the vector connecting the origin of the coordinate system with the lower left corner of the cube.

To filter the reconstructed cubes, the reprojection errors of their points are used. Points greater than a specified threshold are rejected and if a view combination exceeds a given number of rejected points, the model will not be used in averaging. Figure 3.8a shows the reconstructed models from the eight views that are considered valid, i. e., the reprojection error is below

ten pixels and less than 20 rejected points were found. The average for each point from the eight models is taken, which leads to the model seen in Figure 3.8b. When measuring the dimensions of the averaged model it can be seen that the accuracy has improved drastically. The dimensions are $62.77 \times 63.34 \times 62.76$ millimeters, leading to deviations of -0.93, -0.36 and -0.94 millimeters in x, y and z, respectively.



(a) Cubes of the eight view combinations considered for averaging.



(b) Average taken from eight view combinations.

Figure 3.8: (a) shows the remaining models after filtering. The models are then point-wise averaged, which leads to the model shown in (b).

Chapter 4

Implementation for Embedded Systems

In this chapter an implementation of the previously discussed approach to SfM for embedded systems using C++ is discussed. The test system and a baseline implementation using the OpenCV library are presented. Based on that optimizations of the feature detection and triangulation implementations are disclosed. The results of the implementation and the performance gain are then discussed in Section 4.4.

4.1 NVIDIA Jetson TK1

To test the approach to SfM on embedded systems, the NVIDIA Jetson TK1 development board is used. It is designed for evaluating the NVIDIA Tegra K1 SoC and provides peripherals such as USB 3.0, Gigabit Ethernet, mini-PCIe, SATA, I2C, UART and others that are commonly used by embedded as well as desktop systems. It is equipped with 2 GB of RAM and 16 GB eMMC flash memory with Ubuntu Linux 14.04 preinstalled [28]. The NVIDIA Tegra K1 processor is an ARM Cortex-A15 with NVIDIA's 4-Plus-1 technology. The CPU has five cores: a Quad Cortex-A15 and another single Cortex-A15. The quadcore CPU is only enabled when needed. If that is not the case, e. g., in idle mode, the low-power single core takes over [11, 29]. This encourages the use of the SoC in mobile, battery powered systems, since its power consumption rarely exceeds four watts [28].

The main feature of the NVIDIA Tegra K1, however, is that it features a high-performance NVIDIA Kepler-based GPU. It fully supports DirectX 11, PhysX, OpenGL 4.3, and OpenGL ES 3.0, which makes it a comparable equivalent to discrete NVIDIA GPUs used in desktop computers [11]. The GPU is supplied with 192 CUDA cores for arithmetic single-precision floating point operations and 32 special function units (SFUs) for special operations, e. g., logarithm, sine or square root. Each core, or SFU, can execute

one operation per clock cycle, e. g., a CUDA core computes a multiplication in one cycle. This allows for up to 192 multiplications per clock cycle. With the CUDA the GPU can be programmed directly using e. g., C++ code, thereby allowing programs to utilize the massively parallel hardware. Exactly this feature makes the NVIDIA Tegra K1, and therefore, the NVIDIA Jetson TK1 suitable for CV applications such as SfM.

4.2 OpenCV Implementation

The OpenCV library [20] is a collection of more than 2500 CV and machine learning algorithms. The library was initially developed by Intel employees beginning in 1998 [21] and presented first at the IEEE Conference on Computer Vision and Pattern Recognition in 2000. The first official release followed in 2006. It is published under BSD license and actively maintained by both, an open-source community and companies. The OpenCV core is implemented in C++ and additionally provides C, Python, Java, and MATLAB interfaces. All versions of the library are available for Windows, Linux, and Mac. Since 2011, i. e., version 2.3.0, the cross-platform compatibility was extended to include Android and starting with version 2.4.9, released in 2014, an iOS version of OpenCV is available.

The continuously growing collection of algorithms range from basic image processing, i. e., color space transformation, filtering, geometric transformations, etc., over feature detecting and matching, camera calibration and object detection to machine learning [21]. The functionality is not only optimized algorithmically but also by taking advantage of the architectural features of the CPU, mainly streaming SIMD extensions (SSE). Additionally, many algorithms, especially in the domains of video encoding/decoding, feature detection, optical flow analysis, and matrix operations, are implemented using the CUDA API, and thereby, using the parallel computing capabilities of the GPU.

For the implementation on the NVIDIA Jetson TK1 the OpenCV library is compiled from source using version 3.0.0-beta. CUDA support was enabled and compiled with the CUDA 6.5 Toolkit for Linux for Tegra (L4T) release 21.3 [27]. To use the patented SURF feature detector, the non-free module `xfeatures2d`, that is available from the OpenCV extra modules repository [22], was included in the build.

The test application was designed with the interchangeability of the SfM components, i. e., feature detection, extraction and matching, pose estimation and triangulation in mind. Each implements an interface `IFeatureDetect`, `IPoseEstimation` and `ITriangulation`, respectively. A feature detection class has to implement two functions: `findFeatures()`, which searches for distinguishable features in an image and calculates their descriptors, and `matchFrames()`, which compares the descriptors of two

frames and finds matching keypoints. The pose estimation interface defines two versions of a function `estimatePose()` to estimate the position of a frame. The first takes two frames and estimates the rotation and translation of the second frame in relation to the first using the approach described in Section 3.2. The second version takes one frame and a list of corresponding 2D and 3D points. The position is then estimated using the solution to the PnP problem described in Section 3.2.1. Classes implementing the triangulation interface have to have a function `triangulate()`, taking a `ViewCombination` object that contains references to two views, i. e., a pointer to a `Frame` object, their keypoints, and their projection matrices. The resulting 3D points are estimated using the iterative linear least-squares (LS) algorithm discussed in Section 3.3. The flow chart of the test application is shown in Figure 4.1. In the following sections the tasks and implementations of each step are presented.

4.2.1 Initialization and Taking Images

At first the components are initialized and the first image is taken. The images can either be loaded from files or directly taken by the connected camera using the OpenCV `VideoCapture` class. Unfortunately the library does not return the current image but the next image when reading the camera, i. e., if the time between captures is longer than the camera's frame rate, the frame is not skipped but the old frame is returned. This leads to a delay in the image stream, which is not desirable. To solve this problem a polling thread is started. This thread captures an image every 33 milliseconds to ensure that the main program gets the current frame. The main program therefore does not query the frame directly from the library but rather from the polling thread, which is implemented in the class `LiveStream`. Each taken image is saved as `Frame`, which is a class encapsulating all information of a view.

Since most feature detection algorithms only work on images with one channel, the image is converted to grayscale when a frame is created. For visualization purposes the colored image is kept as well. The frame also includes data structures, i. e., a vector of OpenCV `KeyPoints` and an OpenCV `Mat` to save the detected features and their descriptors, respectively. Since the position of each frame has to be known for triangulation, the view's projection matrix is saved as OpenCV `Mat`.

4.2.2 Feature Detection, Extraction, and Matching

The OpenCV library offers a wide range of feature detection classes. All of them are derived from a class `Feature2D` and have to implement, amongst others, the functions `detect()` and `compute()`. The former searches the image for distinct features and returns a vector of `KeyPoints`. Those can then be handed to the latter function, which calculates the descriptors. Ac-

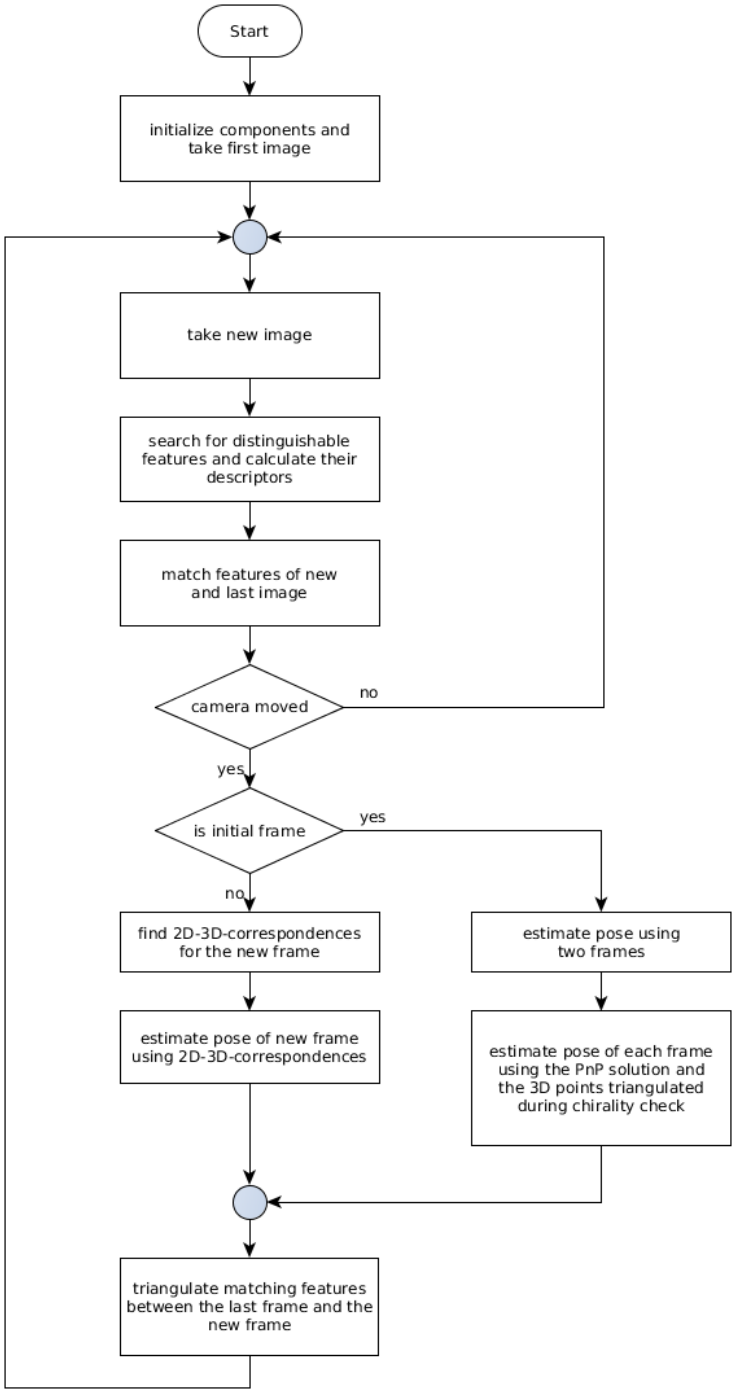


Figure 4.1: Flow chart showing the SfM approach implemented in the test application.

cording to the documentation of version 3.0.0-beta, 15 classes implement the `Feature2D` interface and six of those are only available via the non-free `xfeatures2d` module. However, some of those classes implementing the interface do not provide both methods, e. g., the `SimpleBlobDetector` only implements the detection method. If the missing function is called, a not-implemented-exception is thrown, which violates the Liskov substitution principle (LSP) that states that a derived class has to implement the full functionality of the base class.

The class `OpenCVFeatureDetector` that was implemented for this application, is derived from `IFeatureDetect` and utilizes the common base class of feature detection algorithms. For easy testing of the various algorithms the class implements three create methods for the feature detector, the feature extractor, and the descriptor matcher. This allows the change of the algorithm without touching the classes' code. The algorithms offered by the library can be configured using their constructor. Since the creation of the object is encapsulated in the create methods, variadic templates, i. e., a C++11 feature, are used to propagate the arguments. The implementation of the methods can be seen in the following code snippet.

```
1 template <typename TDetector, typename ...TArgs>
2 void createDetector(TArgs const &...args)
3 {
4   mDetector = TDetector::create(args...);
5 }
```

The ellipses, i. e., the `...`, describe the variadic template. The `args` are an arbitrary number of values of potentially different types `TArgs`, given to the method as constant references. Those are forwarded to the `create` method of the type `TDetector`. Using this concept any class implementing a static `create` method and derived from `Feature2D` can be created.

When the function `findFeatures()` is called, the frame's keypoints are found using the `detect()` function of the previously created feature detector. The descriptors are then computed by calling `compute()` of the created feature extractor. Keypoints and descriptors are saved in the frame object. The `matchFrames()` function can therefore access the descriptors for each frame and apply the matching algorithm from the previously created descriptor matcher by calling `match()`. The library offers two classes derived from `DescriptorMatcher`: `BFMatcher` and `FlannBasedMatcher`. The former is a brute-force approach, the latter uses the fast library for approximate nearest neighbors (FLANN) [17]. For the implementation discussed here the `BFMatcher` is used. The algorithm checks for each descriptor in the first frame whether any descriptor of the second frame matches. The closest matching keypoint, based on a configurable norm, is the correct match. Optionally, cross checking can be enabled: a match is only found if the first keypoint is the second keypoint's nearest match and vice versa.

The combination of SURF detector, SURF extractor, and brute-force

(a) The original image. **(b)** The slightly transformed image.

Figure 4.2: The combination of SURF feature detector, SURF feature extractor, and brute-force matcher using L2-norm and cross checking. (a) and (b) show 30 of the 564 matches in the images. Each match consists of a keypoint pair that is drawn in the same color. Dots in (a) and (b), which are of the same color, mark the same feature in each image, i. e., the detection is very accurate.

matcher, with L2-norm and cross-checking enabled, proved to be the most promising. Figure 4.2 shows the matches found by the mentioned combination. The algorithm was applied to two identical images with the second slightly transformed to imitate an other view. The matches are then drawn as dots of the same color. Other combinations did not yield satisfying results, e. g., the combination of the features from accelerated segment test (FAST) detector, the oriented FAST and rotated BRIEF (ORB) feature extractor, and the same brute-force matcher as before. Compared to the SURF combination it found almost four times more features but the matches are not as accurate. It can clearly be seen in Figure 4.3 that some of the 30 randomly selected matches are not correct.

4.2.3 Pose Estimation

To find the rotation and translation, i. e., the projection matrix, of each frame the class `OpenCVPoseEstimation` was implemented for the test application. It derives from the `IPoseEstimation` interface previously mentioned. The object of the class `ViewCombination`, which is returned by the feature detection component, holds references to two frames, which indirectly gives access to their keypoints, and a vector of OpenCV `DMatch` objects. Each match includes two indices defining the matching keypoints of the first and second frame. The class provides convenience functions to easily access the

(a) The original image. **(b)** The slightly transformed image.

Figure 4.3: The combination of FAST feature detector, ORB feature extractor, and brute-force matcher using L2-norm and cross checking. (a) and (b) show 30 of the 1928 matches in the images. Each match consists of one dot in (a) and one dot in (b), where both are of the same color. It can be seen that many same colored dots are at different image positions, i. e., different feature points, leading to incorrect matches.

matching keypoints without knowing the semantics of the `DMatch` object.

This combination of views is given to the `estimatePose()` method that returns the projection matrices of the two frames. As stated in Section 3.2 the position of the first frame's camera is the origin, therefore the motion of the second frame's camera is relative to it. To compare the triangulated points of all views, the pose estimation using two frames is only performed once to get a set of corresponding 2D-to-3D points. For every following frame the solution to the PnP problem is found.

4.2.4 Initial Pose Estimation

For the initial pose estimation the Essential matrix is needed. OpenCV provides a function `findFundamentalMat()` that returns $\boldsymbol{F}$ based on a set of matching points. The method to compute $\boldsymbol{F}$ can be configured and is either the 7-point, 8-point, RANSAC or least median of squares (LMedS) algorithm. The Essential matrix can then be calculated using the known camera intrinsics and the Fundamental matrix. To compute the SVD of $\boldsymbol{E}$ the OpenCV class `SVD` is used. Using the result of the SVD the four combinations for $\boldsymbol{P}_1$ can be assembled. For each combination the keypoints of both views are triangulated and their chirality is tested. For the triangulation a pointer to an object implementing the `ITriangulation` interface is given to the `OpenCVPoseEstimation` constructor. To test the chirality all points are projected using the two projection matrices. Since the `operator*()` method

for OpenCV `Mat` objects is overloaded, the matrix multiplication needed for reprojection is very simple.

After $\boldsymbol{P}_1$ was computed, the positions of the cameras for both frames is re-computed based on the 3D points found during the initial pose estimation. This is necessary, since the computed projection matrices are relative to each other, and therefore, not directly comparable to the views later computed using 2D-to-3D correspondences. The 3D points used for the computation are filtered based on their reprojection error. The corresponding 2D points are known implicitly. To calculate the new projection matrices the `estimatePose()` method is called for both views.

4.2.5 2D-to-3D Correspondences

To generally find 2D-to-3D correspondences for a new frame its features have to be detected and their descriptors computed. The descriptors can then be matched to previous frames that have already been processed. If a matching feature is found, a triangulated 3D point for the previous frame's keypoint is searched. For that matter, the class `PointCloud`, that holds all triangulated 3D points, keeps a list of corresponding 2D points: for each triangulated point the two frames and indices of the corresponding keypoints are saved. This collection can then be searched.

If enough correspondences were found, the projection matrix of the new frame can be computed using the OpenCV function `solvePnPRansac()`. The library provides multiple algorithms to solve the problem. For the test application the ePnP algorithm from [10] is used. The function returns a rotation and a translation vector. Former can be converted using Rodriguez's formula [14, p. 42], which is implemented in OpenCV as `Rodrigues()`. The rotation matrix and translation vector can then be concatenated to form the projection matrix $\boldsymbol{P}$ of the view.

4.2.6 Triangulation

Knowing matching keypoints and the projection matrices of two frames the 3D points can be triangulated. This is done using the perspective projection

$$\tilde{\boldsymbol{x}} = \boldsymbol{P}\tilde{\boldsymbol{p}}, \tag{4.1}$$

where $\boldsymbol{P} = \boldsymbol{K}\begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \end{bmatrix}$ is the projection matrix, $\tilde{\boldsymbol{p}}$ is the point in 3D space and $\tilde{\boldsymbol{x}}$ is the 2D point on the image. Since the projection matrices of the views have been computed without considering the calibration matrix $\boldsymbol{K}$, the keypoints have to be normalized, so that

$$\tilde{\boldsymbol{x}}_C = \boldsymbol{K}^{-1}\tilde{\boldsymbol{x}}. \tag{4.2}$$

This leads to the equations

$$\tilde{\boldsymbol{x}}_{C0} = \boldsymbol{P}_0\tilde{\boldsymbol{p}} \tag{4.3}$$

and

$$\tilde{\boldsymbol{x}}_{C1} = \boldsymbol{P}_1 \tilde{\boldsymbol{p}} \tag{4.4}$$

that can be solved using the iterative linear LS algorithm described in Section 3.3. The algorithm is implemented in the class `IterativeLinearLS` using functions provided by the OpenCV library. Matrix operations, e. g., matrix multiplications, matrix inversions, and scalar multiplications, are easily performed using the OpenCV class `Mat`. This class is used for all vectors and matrices during the computation, i. e., the keypoints, the calibration matrix, the projection matrix, the created $\boldsymbol{A}$ and $\boldsymbol{b}$ of the linear system for the LS, as well as the resulting 3D point. The linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ from Equation (3.21) can be solved using the OpenCV function `solve()`. The function is configured to use an SVD approach to solve the overdetermined system. As mentioned before, ten iterations at most are repeated. If no solution is found the 3D point is marked as invalid.

When a point was successfully triangulated, its reprojection error is calculated. That is done by applying Equation (4.1) to the triangulated point for both projection matrices. For that matter, $\boldsymbol{P} = \boldsymbol{K}\boldsymbol{P}_0$ and $\boldsymbol{P} = \boldsymbol{K}\boldsymbol{P}_1$ have to be calculated. The reprojected point does not include the distortion introduced by the lens, thus the distorted points have to be calculated using Equation (2.20). Note that the distortion equation expects the coordinates to be between -1 and 1, the center of the image being at $(0, 0)$. The image points can be converted to the correct range with

```
1 double x = (pt.x * 2 - cam.getWidth()) / cam.getWidth();
2 double y = (pt.y * 2 - cam.getHeight()) / cam.getHeight();
```

where `pt` is the projected point. After applying the distortion the coordinates can be recovered using

```
1 pt.x = (x + 1) * cam.getWidth() / 2;
2 pt.y = (y + 1) * cam.getHeight() / 2;
```

so that `pt` will be the distorted point in image coordinates. The reprojection error is then computed by determining the Euclidean distance between the original keypoint and the distorted point.

4.3 Optimizations

Especially on embedded systems, where processor power is relatively low and battery consumption is critical, it is essential to optimize the application. Most of the time it is not enough to just enable the optimization switch of the compiler but the algorithms itself have to be optimized. If that is not possible or the optimizations are not sufficient, hardware features of the used processor can be utilized. Since the algorithms implemented in OpenCV already are optimized, this section attends the optimization for specific hardware features. The NVIDIA Tegra K1 offers a GPU that can be programmed using CUDA. The following sections will show how advantage of this powerful feature is taken.

4.3.1 CUDA

A GPU is a processor using a Single-Instruction, Multiple-Data (SIMD) architecture and was originally intended for rendering images before sending them to a display. In 2006 NVIDIA introduced the CUDA platform and programming model [5]. This created the possibility to program parallel hardware for solving complex computational problems in a very efficient way using C as a high-level programming language. The programming model abstracts the hardware in a way that the developer does not need any knowledge of the underlying system. The supported CUDA features are defined by the compute capability of the GPU. This version number comprises a major and a minor number, where the former describes the architecture of the GPU, and therefore, a set of supported features. The minor version number corresponds to an incremental improvement of the core architecture [5]. The NVIDIA Tegra K1 has compute capability 3.2, i. e., it is based on the Kepler architecture. In general an NVIDIA GPU consists of one or more streaming multiprocessors (SMs). A SM employs a Single-Instruction, Multiple-Thread (SIMT) architecture and is designed to execute hundreds of GPU threads – comparable to threads on a CPU – in parallel. Threads are arranged in warps, i. e., groups of 32, that always start together, even if less threads are needed. Each thread has its own program counter and registers, enabling it to work independently. Common CPU architectural optimizations, such as branch prediction or speculative execution, do not exist on GPUs and all threads within a warp execute one common instruction at a time. If a thread's implementation contains a data-dependent conditional branch, both paths are executed sequentially for all threads of the warp, while disabling the threads that are not on the current path. That is why branching and looping within warps is generally discouraged [5].

The developer implements only one thread, i. e., an algorithm that will run on the GPU. The implementation of a thread, i. e., the kernel, is called from regular C, or in the case of this thesis C++, code using a CUDA-specific language extension. A kernel call

```
1 CudaKernelFunction<<<numBlocks, threadsPerBlock>>>(args);
```

defines how many threads should be started. The started threads are arranged in blocks, where the maximum number of threads per block is limited by the underlying architecture. However, since the hardware should be exchangeable, the limits usually are queried at run time, thus the number of threads per block and the number of blocks emerge from the queried limit and the number of threads needed, which is known. The variables `numBlocks` and `threadsPerBlock` are of type `dim3` and have three dimensions. This hints the original purpose of the GPU as an image processor. Blocks, as well as threads within a block, can be arranged in three dimensions. Figure 4.4 shows a 2×3 block grid, with 3×4 threads for each block. Using more than

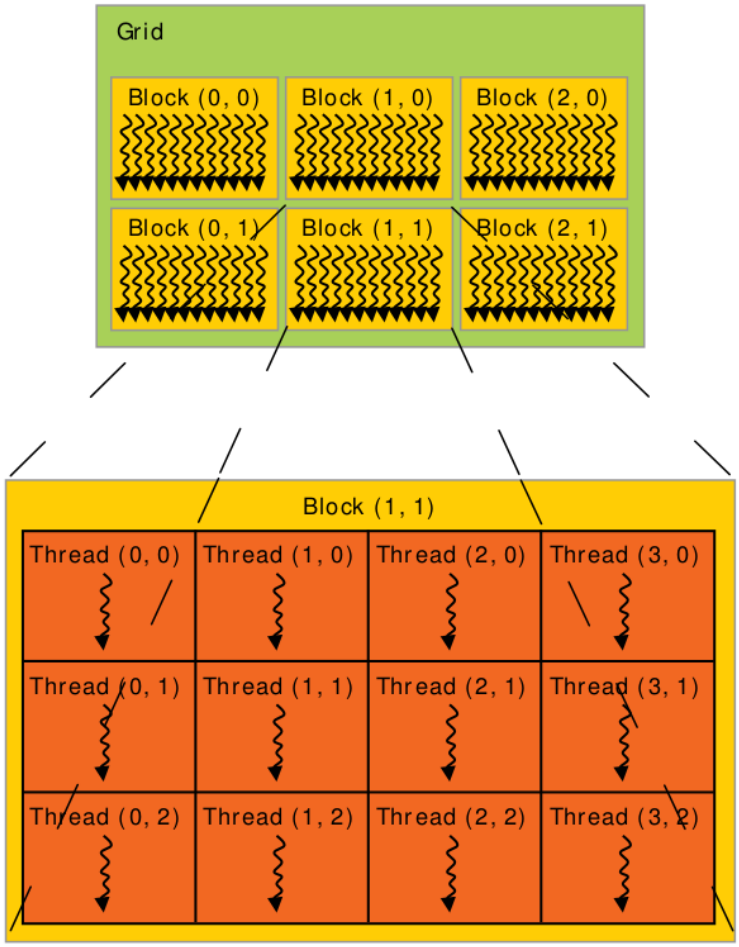


Figure 4.4: CUDA blocks and threads can be arranged in up to three dimensions. This image shows a 2×3 block grid, with 3×4 threads for each block. From [5].

one dimension does not increase the maximal number of threads usable per block. It does, however, make the processing of data, e. g., a two dimensional array or image, easier, since each thread is assigned a unique index. This index can be queried within the context of the thread using the external variables `threadIdx` and `blockIdx` provided by the CUDA API.

When the kernel is called the CUDA-runtime library executes blocks on the available SMs. If a system has more SMs available, more blocks can be executed in parallel, making the overall processing time shorter. If a block is assigned to a SM it is divided into warps, which then can run in parallel or sequentially depending on the available resources.

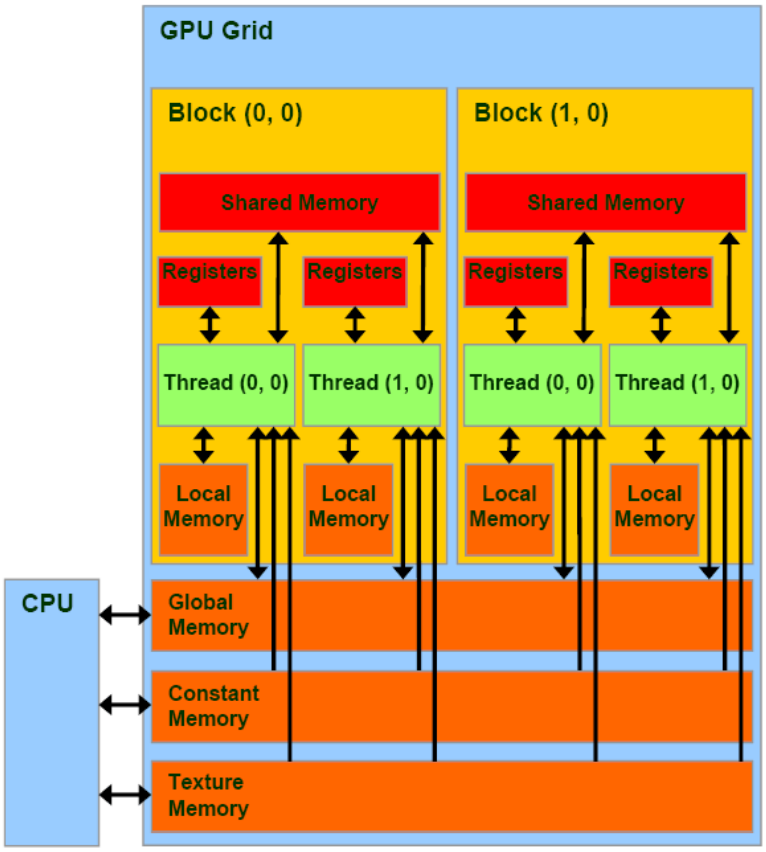


Figure 4.5: The memory hierarchy used in the CUDA programming model. The most important memories are: Registers: fast but small; Shared memory: fast and reasonably large, although shared between threads; Global memory: slow but very large. From [30].

The CUDA programming model introduces a quite complex memory hierarchy [5, 30]. There are seven different memories and two levels of caches available for a thread that differ in size, latency and scope. An overview of this model can be seen in Figure 4.5. The efficient usage of the different memories is essential to the performance of an application. Since the computation itself can be done in parallel, transferring the data from and to the GPU usually is the bottleneck. The key is to take advantage of the properties of the memories.

The smallest but fastest type of memories are registers that exist for each thread. The NVIDIA Tegra K1 provides up to 32768 32-bit registers per block, i. e., 32 registers per thread, if all 1024 threads available per block are used. The registers are comparable to local variables in a function as their value is only valid as long as the thread is running. All scalar variables that are declared within the context of the thread are assigned to registers.

If more local variables than registers, or variables that use large structures, are declared they are saved to the local memory of a thread. This memory has the same latency as global memory, i. e., a very slow memory, and thus, should be avoided.

For a larger amount of data that should be accessed with low latency the shared memory is used. This memory has a latency comparable to registers but is shared between all threads of a block and therefore has to be synchronized. That can either be done by design, i. e., each threads only accesses a unique element, or by using the CUDA-specific language extension `__syncthreads` that creates a sync barrier. A thread reaching this barrier will wait until all threads of the block have reached it, before the execution continues. The shared memory is often used to cache parts of other, slower memories in order to increase performance. The NVIDIA Tegra K1 provides 48 KB of shared memory per block. The shared memory can either be used as user managed cache, by declaring variables with the CUDA-specific `__shared__` modifier, or as L1 cache for other memories [5].

The most important memory when transferring data to and from the GPU is global memory. Accessing the global memory is about 100 times slower than accessing registers or shared memory but, in return, provides a large capacity. The GPU of the NVIDIA Tegra K1 shares its RAM with the CPU, and therefore, provides 1.7 GB of usable global memory. Threads cannot access host memory, i. e., memory allocated for the CPU, directly. As a result data has to be copied to the GPU's global memory, i. e., device memory. The memory on the GPU has to be allocated and the data copied using `cudaMalloc()` and `cudaMemcpy()`, respectively. The pointers to the newly allocated memory can then be passed to the kernel. The global memory is cached using the available L1 cache, and since CUDA compute capability 2.0, global memory is cached using a dedicated L2 cache as well. Writing to the memory, however, will invalidate and flush the cache line. As it is unnecessary to copy data on the same physical memory, CUDA provides a way to map data from host memory to device memory. This so called zero-copy memory reduces the data to be copied, thus increasing performance.

If the GPU should only have read capabilities on data, constant memory is an alternative to global memory. Although both of them share the same physical memory, the access times for constant memory usually are considerably faster [30]. Both memories are cached but since constant memory cannot be written, its cache lines will not be flushed. Variables can be allocated in constant memory by declaring them at global scope with the `__constant__` modifier. The variable can be initialized at declaration or set from host code using `cudaMemcpyToSymbol()`. The NVIDIA Tegra K1 provides 64 KB of constant memory.

The CUDA programming model provides two more memories: texture memory and surface memory. These are specialized memories that bring several performance benefits when working with textures. A more detailed description of those can be found in [5].

Applications containing CUDA code have to be compiled with the NVIDIA CUDA compiler (NVCC). This is necessary, since the CUDA programming model extends the C programming language by introducing new keywords that are unknown to regular compilers, e. g., the previously mentioned modifiers for memory, variables to query the thread's index or the specifiers `__global__`, `__device__`, and `__host__`. The latter let the compiler know for which processor, CPU or GPU, the function should be compiled. The NVCC uses a regular C or C++ compiler, in this case the GNU compiler collection (GCC) C++ compiler, for compiling regular code and its specialized GPU compiler for device code only. All functions marked with `__global__` receive special treatment. Only such a function can be called from the host and run on the device, i. e., the entry point for a kernel [5, p. 89].

4.3.2 Feature Detection

The feature detection using SURF, discussed in Section 4.2.2, is done using the CPU. The OpenCV library, however, provides a class `SURF_CUDA` that implements the same algorithm using a CUDA-compatible GPU. Unfortunately the class does not implement the same interface `Feature2D` as the CPU classes, therefore a wrapper class was built, implementing the `IFeatureDetect` interface previously mentioned. Since the GPU cannot, in general, access the host memory, `SURF_CUDA` expects the data to be transferred to the device memory using the OpenCV class `GpuMat`. This class abstracts memory allocation and data management and thus does not allow the developer to decide which memory to use, which can in some cases have an impact on performance.

The GPU accelerated implementation of the SURF algorithm does not implement the `detect()` and `compute()` methods but overloads the `operator()()` method for detection and computation of keypoints and descriptors, respectively. It also provides a function to download, i. e., copy to host memory, and convert the detected keypoints to the same format used in the CPU implementations. These can then be saved in the `Frame` object, without changing the used data types.

To match the descriptors the class `BFMatcher_CUDA` is used. It implements the same algorithm as the brute-force matcher described before but takes `GpuMat` objects as arguments. The usage of the GPU promises a vast improvement, since the descriptors can be matched in parallel. As the SURF implementation, the matcher class provides a function to convert the `GpuMat` matches to the same format, i. e., a vector of OpenCV `DMatch`, as used before.

All memory allocated during feature detection, that is for the image, keypoints, descriptors and matches, will be freed immediately. This is necessary, since when processing many images the GPU will eventually run out

of memory, if all images are kept. The high number of allocations and deallocations, however, has a large impact on the performance of the application. So as an alternative, the memory allocated for the current and last image could be reused, therefore avoiding reallocation and minimizing the overhead to copying the new image to the device memory. Unfortunately the `GpuMat` class provides, next to the constructor, only the function `upload()` to copy data to device memory. This function, however, calls `release()` and then reallocates memory before actually copying the data. The (de-)allocation is done using an allocator object that can be supplied when constructing the `GpuMat`. To optimize the feature detection performance discussed here, a custom allocator is implemented, that on one hand complies with the OpenCV `GpuMat` semantics, and on the other hand, holds an already allocated memory block that can be used by the `GpuMat` object, thus avoiding reallocation.

4.3.3 Triangulation

Next to the feature detection, the triangulation promises the most performance gain, when optimized. Since all 3D points can be processed independently, the iterative linear LS triangulation method from Section 3.3 is easily ported to the GPU. The implementation is encapsulated in the class `GPUIterativeLinearLS`, which consists of a regular C++ part, that prepares the memory needed for triangulation and processes its results, and a CUDA part, that implements the actual triangulation algorithm. For solving the LS equation $\boldsymbol{Ax} = \boldsymbol{b}$ the CUDA basic linear algorithm subprograms (CUBLAS) library, that is included in the CUDA 6.5 Toolkit, is used. The library provides the function `cublasSgelsBatched()` [4], which solves many LS problems in parallel but has to be called by the host. This limitation leads to the approach shown in Figure 4.6. To take advantage of the GPU, kernel calls are made before each call of the CUBLAS function to prepare the LS matrices in parallel. After the LS solution is computed, kernels to calculate the weights and the reprojection errors are called. The following presents the implemented steps in detail.

Preparing Memory: As mentioned before the memory management is complex but crucial to the performance of the application. To find the best solution for the use of memories, different versions were implemented. The performance comparisons between the usage of the different memories is discussed in Section 4.4.

The data that has to be transferred to the GPU for each view combination is the keypoints and the projection matrices for both views. Additionally the calibration matrix, its inverse and the distortion coefficients have to be transferred to the GPU. As these do not change, they have to be copied only once. During the calculation of the 3D points, arrays for the normalized key-

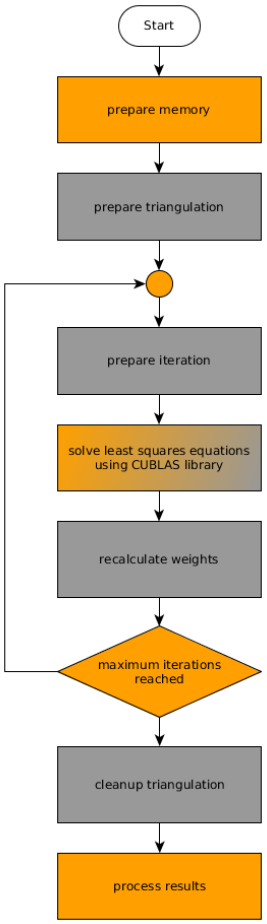


Figure 4.6: Flowchart showing the iterative linear LS triangulation approach as it is implemented in `GPUIterativeLinearLS`. Orange parts are executed on the host, i. e., the CPU, gray parts are executed on the device, i. e., the GPU. Solving of the LS equation is done using the CUBLAS library, which is needed to launch from the host but will perform the computation on the device.

points and their calculated weights have to be allocated on the device for each of the two views. For solving the LS problem, memory for the $\boldsymbol{A}$ and $\boldsymbol{b}$ matrices has to be allocated for each thread. Furthermore an integer array in device memory with elements for each thread is required for calling the CUBLAS function. The arrays containing the results, i. e., the triangulated 3D points and their reprojection errors, have to be allocated as well.

Memory allocation on the device takes a relatively long time, therefore all memory required for the calculation is allocated in advance. Furthermore the memory is not freed between calculations to avoid unnecessary dealloca-

tions and allocations. The constructor of `GPUIterativeLinearLS` takes an argument `pre_allocate_n` and allocates memory for the specified number of keypoints. If and only if more keypoints have to be calculated at a later time, the memory is reallocated.

A special case is the allocation of the arrays for $\boldsymbol{A}$ and $\boldsymbol{b}$ defining the LS equations. The CUBLAS function expects an array of pointers to two-dimensional arrays for each of the two matrices. Note that in C++ a two-dimensional array is a linear memory block. Therefore the allocation can be done in two steps: at first an array of `n_keypoints` pointers and an array of floating points is allocated. The size of the latter is `n_keypoints` times twelve elements for the $\boldsymbol{A}$ matrix and `n_keypoints` times four for the $\boldsymbol{b}$ vector. Then the elements of the pointer array can be filled with the addresses of the float array, where the address is increased by twelve, respectively four, for each element. This leads to each element pointing to a memory block with the size of the two-dimensional matrix. Since it is not possible to directly write to device memory from the host. The addresses have first to be inserted into a temporary array that is then copied to the device using `cudaMemcpy()`.

For the projection matrix, calibration matrix and distortion coefficients that are not changed by the triangulation and are the same for each 3D point, constant memory can be used. Since the algorithm uses the inverse calibration matrix $\boldsymbol{K}^{-1}$ and the multiplication of the projection matrices and the calibration matrix $\boldsymbol{K}\boldsymbol{P}_0$ and $\boldsymbol{K}\boldsymbol{P}_1$, the results are calculated in advance and saved to constant memory. This has the advantage to directly access the needed values instead of calculating them for each thread. The constant memory can be written from the host using the CUDA API.

As mentioned in Section 4.3.1, the NVIDIA Tegra K1 uses the same physical memory for the CPU and GPU. To avoid copying data from host memory to device memory, which would only copy memory within the RAM, the host addresses can be mapped. According to [5], this zero-copy memory is identical to the global memory but cannot be allocated or freed by the device. Mapping memory is either done using the CUDA functions `cudaHostRegister()` and `cudaHostGetDevicePointer()` or directly allocated using `cudaHostAlloc()` with the flag `cudaHostAllocMapped` set. Note that the former functions are not implemented for the ARM architecture, and therefore, can not be used on the NVIDIA Tegra K1. If the device is initialized with the `cudaDeviceMapHost` flag, the device pointer returned by the former or the host pointer returned by the latter option can be handled exactly the same as memory allocated directly on the device.

For the implementation discussed here, only the arrays for the keypoints of both views, the triangulated points and the reprojection errors are mapped to the host memory. Note that only the copying of data between host and device memory is prevented and the host memory has to be allocated as well. That is because the data itself, e. g., the keypoints, still has to be copied to

another host memory location, since the kernel expects it in a different format as the `frame` objects provide it. The keypoints are saved as vectors of OpenCV `KeyPoint` that are returned by the OpenCV feature detection algorithms. To directly map the keypoints saved in the `frame` objects is another point of optimization, which is not conducted in this thesis, since the compatibility to OpenCV functionality should be preserved. Mapping the remaining allocated memory, i. e., the arrays for normalized keypoints, weights, LS arrays, etc., is not necessary, since it is used on the device only. Therefore zero-copy memory would not hold any advantage, as the data is not processed by the CPU.

Preparing Triangulation: The preparation of the the triangulation is done once for each call of `triangulate()`. For that matter, a kernel `prepare_triangulation` is called that initializes all variables used during the calculation. These are the normalized keypoints $\tilde{\boldsymbol{x}}_{C0}$ and $\tilde{\boldsymbol{x}}_{C1}$, the weights w_0 and w_1, the entry in the integer array used for the CUBLAS function, and the reprojection error. The kernel creates `n_keypoints` threads. Especially for the normalization of the keypoints the use of a kernel promises a speedup, since each keypoint can be processed in parallel. The normalized keypoints are computed by

$$\tilde{\boldsymbol{x}}_C = \boldsymbol{K}^{-1}\tilde{\boldsymbol{x}}, \tag{4.5}$$

where the inverted calibration matrix $\boldsymbol{K}^{-1}$ is either taken from global, constant or zero-copy memory, depending on the configuration of `GPUIterativeLinearLS`. The normalized and original keypoints reside either in zero-copy or global memory, since the size of constant and shared memory is limited.

Preparing Iteration: The preparation of the iteration has to be repeated before every attempt to solve the LS problem. The kernel `prepare_iteration`, that creates a thread for each keypoint pair, fills the $\boldsymbol{A}$ and $\boldsymbol{b}$ matrices. The matrices corresponding to the thread's index can be accessed using the array of two-dimensional matrices discussed earlier. The CUBLAS library is designed to be used with Fortran environments, therefore the `cublasSgelsBatched()` function expects the matrices to be in column-major format [4]. C and C++, however, use a row-major format. To overcome this format mismatch the matrices are simply filled using mapped indices. The following shows the indices in a row-major format on the left and the indices in a column-major format on the right-hand side

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \\ 9 & 10 & 11 \end{bmatrix} \leftrightarrow \begin{bmatrix} 0 & 4 & 8 \\ 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \end{bmatrix}. \tag{4.6}$$

The values for the entries in $\boldsymbol{A}$ and $\boldsymbol{b}$ are thereby the same as described in Equation (3.21). For each iteration the entries have to be multiplied by the weights computed from the triangulated point, therefore the matrices have to be rebuilt.

Solving the Least-Squares Problems: As previously mentioned the LS problem is solved using the CUBLAS library. For calling the library function a `cublasHandle_t` object is needed. Creating the handle takes 550 to 600 milliseconds, and therefore, is not created for each iteration but rather at creation of the `GPUIterativeLinearLS` class. The function then calculates the solution for all keypoints in parallel using the provided arrays of pointers to the $\boldsymbol{A}$ matrix and $\boldsymbol{b}$ vector. The result is saved to the $\boldsymbol{b}$ vector for each keypoint.

Recalculate Weights: The weights of the triangulated points are computed for both projection matrices and then saved to the float array in device memory. How the weight is computed was shown in Section 3.3. If the change in weight is below 10^{-9} the point is copied to the provided `Point3D` array and the thread will skip further iterations.

Cleaning Up Triangulation: When all iterations are done, the reprojection error has to be calculated. For the combined calibration and projection matrix $\boldsymbol{KP}$ two alternatives are implemented: If the usage of constant memory is enabled in `GPUIterativeLinearLS`, $\boldsymbol{KP}_0$ and $\boldsymbol{KP}_1$ are computed in advance and then copied to the constant memory while preparing memory. The other alternative uses an array allocated in shared memory, which is filled using twelve threads. Each thread is responsible for one element of the computed matrix, where the `threadIdx` is used as index for the array. To ensure that all elements of the projection matrices have been filled, a sync barrier has to be inserted after populating the arrays. The computation of the reprojection error is then performed as discussed in Section 4.2.6.

Processing Results: After the triangulation is completed, the computed 3D points and their reprojection errors have to be added to the point cloud. Before that can be done the data has to be copied from the device to the host memory. This can be done using the `cudaMemcpy()` function. As previously mentioned, no memory is freed at this point to avoid unnecessary reallocations. If, however, zero-copy is enabled no data has to be copied as it can be accessed directly.

Table 4.1: Results of the comparison of SURF feature detection implementations on CPU and GPU. Both algorithms are provided by the OpenCV library and no custom allocator is used. The times, except for construction, are average times from 100 runs. It can be seen that the GPU actually is slower than the CPU implementation when finding features in the images. Only the brute-force matching has a speedup greater one.

	CPU	GPU	Speedup
Class Construction	0.014 ms	0.009 ms	1.46
Finding Features Left	3520.23 ms	4467.83 ms	0.79
Finding Features Right	3519.68 ms	4448.12 ms	0.79
Matching Images	1283.02 ms	847.99 ms	1.51

4.4 Results

This section discusses the results of the previously presented optimizations. For that matter the original OpenCV implementation is compared to the optimized implementations of feature detection and triangulation using the GPU. Further the overall improvement of the approach to SfM is presented.

4.4.1 Feature Detection

To compare the CPU and GPU implementations of the SURF algorithm found in the OpenCV library, a test application was implemented. This implementation first searches for features in two images and computes their descriptors using the `findFeatures()` method defined in the `IFeatureDetect` interface. The images are taken with the camera described in Section 3.1.1 with a resolution of 1280×720 pixels. The found features are then compared using the `matchFrames()` method.

The execution times for finding and matching the features is measured for both implementations, where the same SURF parameters are used. To compensate outliers, the average time of 100 runs is taken. The measured times for the test runs without the custom allocator are shown in Table 4.1. It can be seen that the GPU implementation actually is slower than the CPU when finding features in the images. The CUDA implementation of the brute-force matcher, however, achieves a small speedup of 1.51. The implementation was tested on a reference system to check whether the performance loss is in the implementation or hardware. The test GPU, i. e., an NVIDIA Quadro K1000M, achieved speedups of about 2.92 and 2.37 for finding the features and matching them, respectively, suggesting that the OpenCV implementation is designed for desktop systems with a dedicated graphics card.

Table 4.2: Results of the comparison of SURF feature detection implementations on CPU and GPU. A custom allocator for `GpuMat` is used that prevents reallocating memory. The number of features found in the images is equal in both implementations but the number of matching features differs. Additionally the found matches are not the same.

	CPU	GPU	Speedup
Class Construction	0.014 ms	0.01 ms	1.40
Finding Features Left	3520.23 ms	4436.27 ms	0.79
Finding Features Right	3519.68 ms	4412.71 ms	0.79
Matching Images	1283.02 ms	845.89 ms	1.52
Features Left	2250	2250	–
Features Right	2108	2108	–
Matching Features	1387	1378	–

To check whether the allocation of device memory is the time consuming part of the implementation, a custom allocator is implemented. Due to the use of the OpenCV `GpuMat` type, device memory is allocated anew for each set of data that is copied for finding features, computing descriptors or matching features. As discussed in Section 4.3.3 this is disadvantageous and has a negative effect on the performance. For that matter the custom allocator allocates memory once and reallocates only if the requested size is larger than the memory already allocated. The results of the tests can be seen in Table 4.2. The table shows that the speedups have not been increased by the proposed optimization, the same goes for the test on the reference system, what suggests that the overhead introduced by allocation is not critical for finding features and computing the descriptors.

The number of found features in each image are the same for CPU and GPU, the found matches, however, differ. To check whether the features found in the images are the same, the test application was extended. It was found that the features are not in the same order but – within a 10^{-4} pixel radius – equal. The `matchFrames()` method, on the other hand, does not return the same matches for the CPU and CUDA implementations.

4.4.2 Triangulation

The iterative linear LS triangulation was implemented using the CUDA API, so that through the usage of the GPU the 3D points can be triangulated in parallel. The implementation makes use of two potential memory optimizations: using constant memory for camera intrinsics and projection matrices and using zero-copy for data used by both, the CPU and GPU. That leads to four possible combinations to be tested.

The NVIDIA Tegra K1 is generally capable of running 1024 threads per block, the implementation discussed here, however, limits the maximal threads to 799. This has to be done, since the CUDA implementation allocates a maximum of 41 registers per thread during the calculation[1]. Dividing the registers available per block, i. e., 32768, by the maximum registers used per thread leads to the maximum number of threads per block, so that sufficient registers are available.

For the purpose of testing the performance of the GPU implementation, a comparison between `IterativeLinearLS` and `GPUIterativeLinearLS` was implemented. It measures the time the call of `triangulate()` takes for each implementation. The average of ten calls is taken and saved to a text file, which can later be analyzed using MATLAB. The measurement is repeated for different sizes of keypoints, where both objects are newly created before and destroyed after each measurement. The constructor of the GPU implementation resets the CUDA device, creates the CUBLAS handle and allocates memory for all keypoints, as mentioned in Section 4.3.3. The time it takes to create the object is measured, to see whether a large preallocation takes more time. Note that for the zero-copy implementation memory has to be allocated on the host instead of the device. Figure 4.7 shows the results of the measurements. Resetting the device and the creation of the CUBLAS handle do not vary with the number of keypoints and take about 165 and 550 to 600 milliseconds, respectively. It can be seen that the time for allocation is almost the same for all memory configurations. The time it takes to allocate memory is nearly constant for all numbers of keypoints below and all numbers above 1000 keypoints.

The execution times of the `triangulate()` method are shown in Figure 4.8, the speedup in Figure 4.9. As expected, the GPU is faster and the time is rising linearly with the number of keypoints. It can be seen that using the constant rather than the global memory for the projection matrices and camera intrinsics does not severely change the performance. The reason for that lies in the fact that constant memory is a cached part of the global memory. The NVIDIA Tegra K1 supports compute compability 3.2, and therefore, global memory is cached as well using a dedicated L2 cache that is flushed on write. There is no difference between constant and global memory in this implementation, as the matrices are read-only in both cases.

The zero-copy, on the other hand, does perform not as good as as the common approach that copies memory to the device. This behavior usually occurs if the GPU has its dedicated memory and is connected with the CPU via the, e. g., PCIe bus. In such cases mapping the host memory usually is slower than copying the data, since each access has to be done using the

[1]The maximum number of registers can be determined by launching NVCC with the `-keep -cubin` flags. The created .cubin file can be analyzed with `cuobjdump` and contains, among others, information about the registers used by a function.

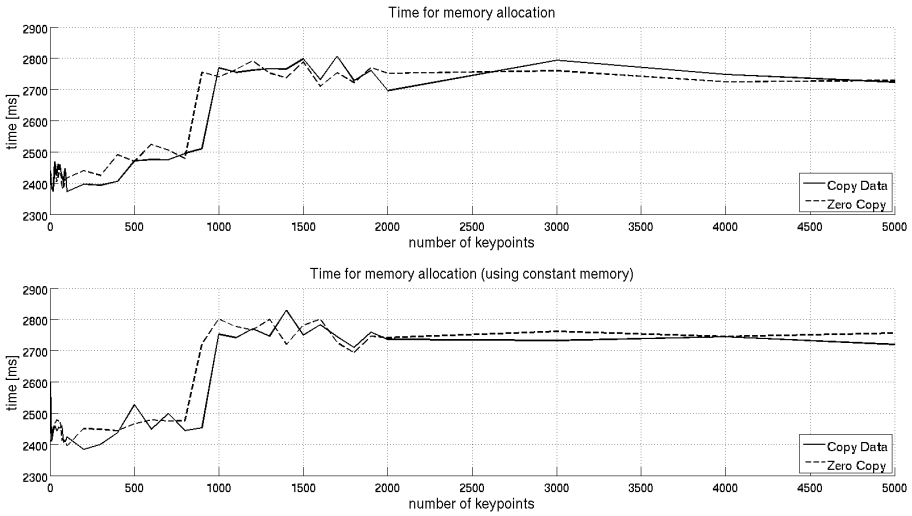


Figure 4.7: Time it takes to construct the object for the GPU implementation in milliseconds. During the creation the CUDA device is reset, the CUBLAS handle is created, and memory is allocated. The upper graph shows the times for the class not using constant memory, the lower graph with the usage of constant memory.

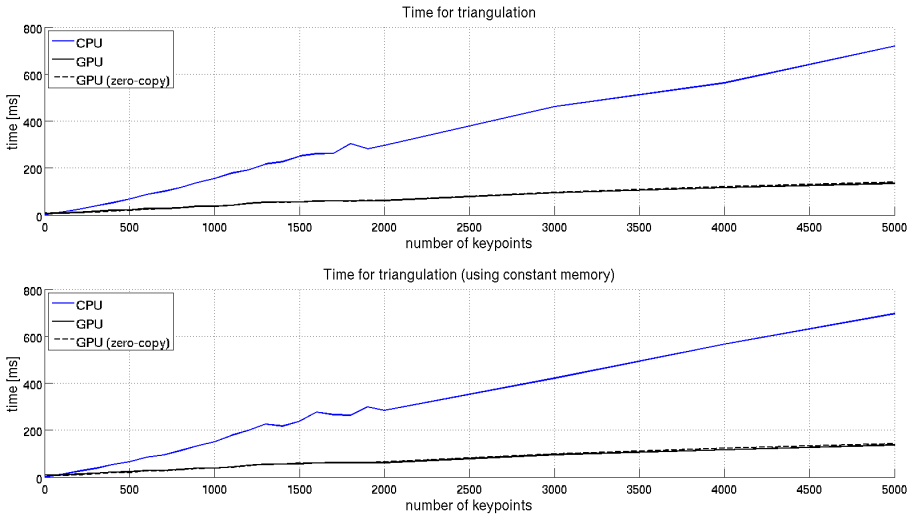


Figure 4.8: Computation times in milliseconds of the `triangulate()` method for both, the CPU (blue) and GPU (black) iterative linear LS triangulation implementation. The upper graph shows the times without, the lower graph the times with the use of constant memory. It can be seen that the times for global and zero-copy memory do not differ by much.

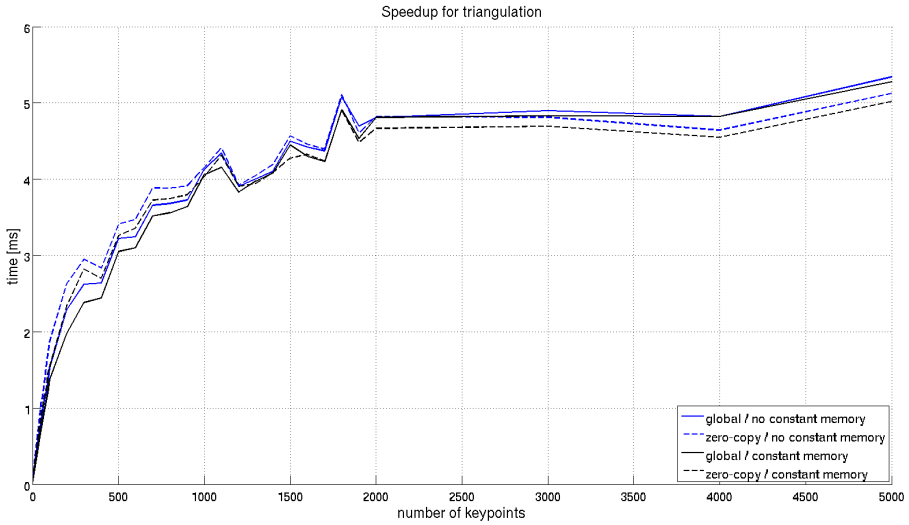


Figure 4.9: Speedup of the `triangulate()` method between the CPU and GPU implementation. Using zero-copy memory (dashed lines) is slower than copying data to the device memory when using global memory. Whether the global (blue) or constant (black) memory is used for the camera intrinsics does not change the speedup by much.

bus. On the NVIDIA Tegra K1, however, host and device memory are physically the same and therefore zero-copy should not differ from the common approach. The reason why it is slightly slower in the tests performed, might lie in the handling of the zero-copy memory in the CUDA runtime.

Next to the speedup the correctness of the calculation is critical. The GPU implementation is done using single-precision, i. e., using datatype `float`, whereas the OpenCV implementation uses `double` values. Single-precision values have 32 bit, whereas double-precision uses 64 bit. A SM of compute capability 3.x consists of 192 CUDA cores for arithmetic operations and 32 SFUs for the computation of logarithms, square roots and others. Operations with single-precision data generally perform better than with double-precision, e. g., in one clock cycle 192 32-bit floating point multiplications can be computed. When using 64-bit floating point values, on the other hand, only eight multiplications can be performed in the same time [5].

In consideration of `float` computations being less accurate than `double` computations, a test program was written that computes the triangulation for a set of keypoints for both, the OpenCV and the GPU implementation. For that matter, keypoints are extracted from two images using the SURF implementation of OpenCV. The 3D points then are computed using both versions of the iterative linear LS algorithm and compared. The points are considered equal, if the absolute difference of the x, y, z and w coordinates

and the reprojection errors is below 10^{-4}. The feature detection detects 1437 keypoint pairs. From those 38 triangulated points differ more than 10^{-4} in at least one component. The average absolute differences from the erroneous points are $\Delta x = 8.7 \cdot 10^{-5}$, $\Delta y = 5.5 \cdot 10^{-5}$, $\Delta z = 2.2 \cdot 10^{-4}$, $\Delta w = 0$ and $\Delta E_{reprojection} = 0.01385$. Given this measurement it can be said that, although the triangulation is not exactly the same, the precision of the calculation is sufficient.

4.4.3 Combining the Optimizations

In the following the application is run with different configurations of the discussed optimizations.

Evaluate the Algorithms

To evaluate the approach to SfM the application discussed in Section 4.2 is executed on the NVIDIA Tegra K1 using multiple configurations. To verify the results of the implemented SfM algorithms, the manually extracted features of the images of the Rubik's cube from Section 3.4 are used. The verification is repeated for the two implementations of the iterative linear LS triangulation. It should be mentioned that the GPU implementation of the triangulation uses constant memory but no zero copy. The four views of the Rubik's cube are processed sequentially, where each view is compared to the previous one, leading to three iterations of the SfM implementation. In the first iteration the initial pose between view one and two is estimated, and as part of that, a first set of 3D points is computed. As mentioned before, the pose of the second view is relative to the first, why the PnP problem has to be solved using the set of 3D-to-2D correspondences. The resulting projection matrices can then be used to triangulate the points without reference to each of the two views. This assures that the following views use the same coordinate system, i. e., the world coordinate system, for their projection matrices. The remaining views skip the initial pose estimation and are directly recovered from found 2D-to-3D correspondences using the PnP approach.

Figure 4.10 shows point clouds computed by the test application. For the visualization of the points the point cloud library (PCL) in version 1.7.2, which was built from source for the NVIDIA Tegra K1, is used. The figure shows that both, the CPU and GPU implementation compute the same point clouds. It can also be seen that the point cloud generated during initial pose estimation, which actually is a byproduct of finding the projection matrices of both views, does not differ from the point cloud computed after re-computing the projection matrices for both views. This proves that the approach to first estimate a pose from a known set of 2D keypoints and triangulating the 3D points from that knowledge, then re-computing the pose

Table 4.3: Execution times of the SfM algorithm for the 91 manually extracted Rubik's cube keypoints.

	CPU	GPU	Speedup
Feature Detection	0.023 *ms*	0.023 *ms*	1.00
Initial Pose Estimation	23.97 *ms*	19.65 *ms*	1.22
PnP Pose Estimation	0.54 *ms*	0.56 *ms*	0.96
Triangulation	9.92 *ms*	9.86 *ms*	1.00

using the 2D-to-3D correspondences is legitimate. The bottom row of images shows the combined points of all four views. Although the triangulated points of the different view combinations are not equal, the 3D model still resembles the real-world object, i. e., the Rubik's cube. The average execution times of the SfM implementation can be seen in Table 4.3. Although by using only 91 keypoints the overhead of copying memory from and to the GPU is relatively large compared to the execution time. The CUDA implementation of the iterative linear LS triangulation performs slightly better than the CPU implementation. During initial pose estimation the triangulation is repeated up to four times. That explains the greater speedup for that measurement.

To test the application including the feature detection algorithm, a scene was set up. Five live images of the scene from different angles were taken using the camera discussed in Section 3.1.1. This leads to four different view combinations. Exemplary for the four combinations, Figure 4.11 shows the two images for third view combination and the detected features in each frame, where the color indicates the matches. Note that the matches were calculated by the more reliable CPU implementation of the brute-force matcher. It can be seen that many features are not matched correctly, e. g., the features on the right-hand side of the keyboard. The created point cloud from all views can be seen in Figure 4.12. The point cloud contains 950 points but the scene has not been successfully reconstructed. Since the only difference between this test run and the previous, using the Rubik's cube keypoints, is the live feature detection, this step is apparently the reason for the erroneous point cloud.

Evaluate the Performance

To evaluate the performance, images from the camera are taken and processed. For this the original and optimized implementations of feature detection and triangulation are considered, leading to four combinations. It should be mentioned that both SURF implementations are initialized with the same parameters, as already done in Section 4.4.1. Nevertheless the matching of

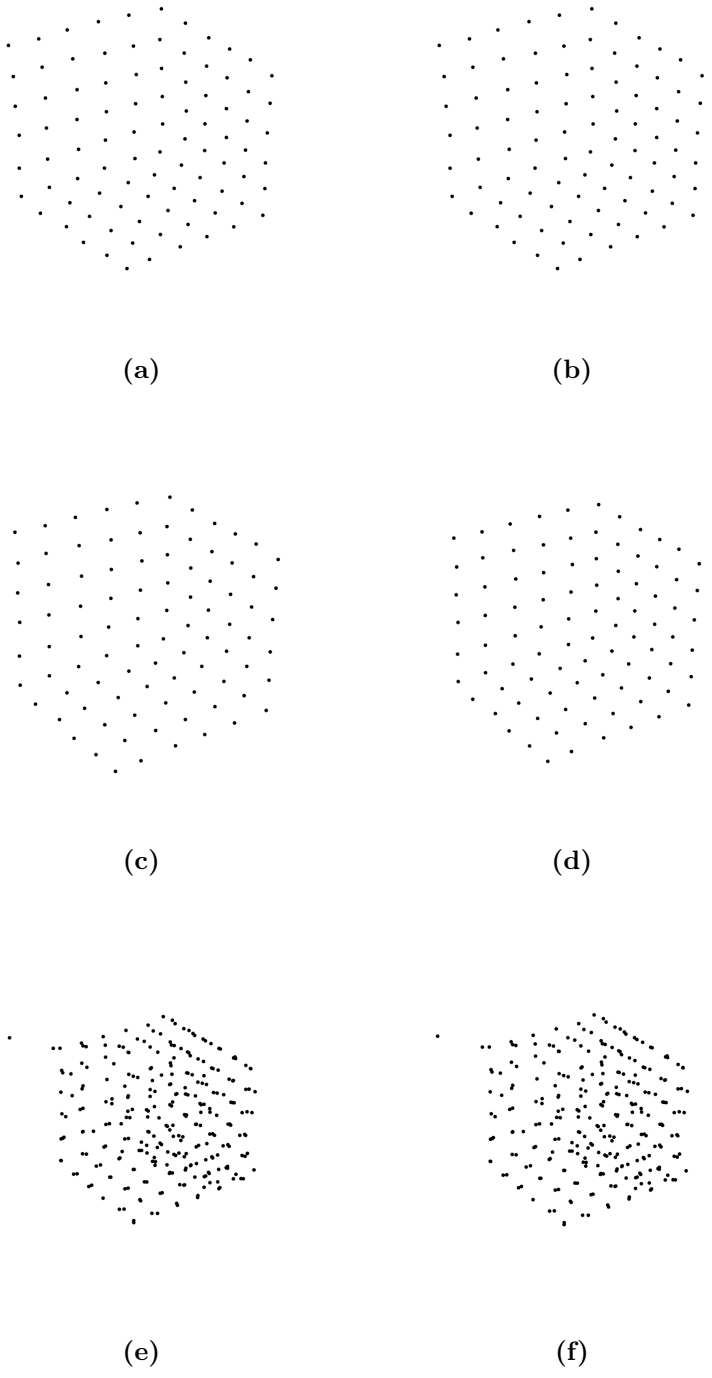


Figure 4.10: Point clouds computed by the test application using Rubik's cube keypoints. Figures (a), (c), and (e) are taken from the CPU triangulation, (b), (d), and (f) from the GPU triangulation. (a) and (b) show the triangulated points during initial pose estimation. (c) and (d) show the triangulated points after the positions of the first two views were re-computed using the PnP approach. (e) and (f) show the point cloud after processing all four views.

(a)

(b)

Figure 4.11: (a) and (b) show the two images taken for third view combination. The dots indicate features; same colored dots in the two images build a match.

features apparently is implemented differently, leading to discrepancies in the matches.

The averaged execution times can be found in Tables 4.4 and 4.5 for the CPU and GPU implementations of the SURF algorithm respectively. Most of the time is consumed by the feature detection stage. As already discussed in Section 4.4.1 the CUDA implementation of the brute-force matcher does not return the same keypoints. Generally the matches returned by both implementations were very few. In average only 50 to 80 matching features are found in two frames. This negatively affects the performance of the CUDA powered triangulation. Section 4.4.2 showed that the performance gain by using the GPU is minimal for small sets of keypoints, e. g., a speedup of more than four is not reached until more than 1000 matches are processed.

Figure 4.12: The point cloud resulting from the four view combinations. The scene has not been reconstructed correctly.

Table 4.4: Average execution times of the SfM algorithm, when using the CPU for feature detection and matching.

SURF Implementation CPU

	Triangulation	
	CPU	GPU
Feature Detection	629.05 ms	635.89 ms
Initial Pose Estimation	66.19 ms	34.44 ms
PnP Pose Estimation	13.03 ms	9.78 ms
Triangulation	8.98 ms	7.90 ms
First Iteration Total	722.03 ms	742.25 ms
Total	651.06 ms	653.57 ms

The number of found matches can, however, be increased by modifying the SURF parameters. By setting these parameters a trade-off between number and distinctness of keypoints is made. For the tests previously discussed the parameters were chosen to find as distinct features as possible. It should also be mentioned that the execution times for the CUDA SURF implementation vary widely between 900 and 2700 milliseconds. Seeing the results it can be said that the CPU implementation is, at least for the NVIDIA Tegra K1, preferable.

Table 4.5: Average execution times of the SfM algorithm, when using the GPU for feature detection and matching.

SURF Implementation GPU

	Triangulation	
	CPU	GPU
Feature Detection	2288.84 ms	2017.92 ms
Initial Pose Estimation	147.97 ms	151.67 ms
PnP Pose Estimation	35.48 ms	49.45 ms
Triangulation	28.84 ms	9.40 ms
First Iteration Total	2832.80 ms	2823.59 ms
Total	2353.17 ms	2077.44 ms

Chapter 5

Conclusion

This thesis presented a way to generate 3D maps from multiple images taken from a single camera. After the theoretical foundations of projective geometry and CV were explained, the approach to SfM was discussed in detail and the presented algorithms verified using MATLAB. The approach can be summarized into three steps: detection of features in images and matching them, estimation of the projection matrix for each image, and triangulation of the 3D points. For pose estimation two solutions were developed, the first estimates the position of the views in relation to each other and the second estimates the position of a view from known 2D-to-3D correspondences using the ePnP algorithm. The triangulation problem was solved using the iterative linear LS approach.

The test application was implemented for the NVIDIA Tegra K1 using C++ and the OpenCV library. The task of feature detection and descriptor computation was performed using the SURF implementation of the non-free `xfeatures2d` OpenCV module and a brute-force feature matcher. The pose estimation problem was solved using the OpenCV function `findFundamentalMat()`, the class `SVD` and most importantly the class `Mat`, which implements matrix operations. The latter was also used during triangulation. To estimate the 3D points a linear system was set up and solved as LS problem using the OpenCV function `solve`.

Based on this implementation several optimizations were presented to take advantage of the GPU the SoC provides. First the CUDA programming model was presented and an overview of the memories available to the GPU was given. The CUDA SURF implementation of the OpenCV library was used for feature detection and it was found that – on the NVIDIA Tegra K1 – the CPU alternative performs better. A test was conducted whether excessive device memory allocation was the cause of the performance loss, which was disproved. The iterative linear LS algorithm was re-implemented to use the GPU. For that matter multiple configurations were tested that take advantage of different device memories, i. e., global, constant, shared and

zero-copy memory. It was shown that the performance on global and constant memory is equal, as the hardware architecture allows caching of both memories and due to the algorithm, the cached data is only read, preventing cache flushes. It was also pointed out that small keypoint sets do not achieve a satisfactory speedup, sets above 1000 keypoints however, perform at least four times better than the CPU alternative.

The final application was tested with both, manually and automatically extracted keypoints. Former use the same keypoints that were used in the MATLAB implementations and show that the application works correctly. The keypoints extracted using SURF, on the other hand, lead to a point cloud that does not resemble the recorded scene.

5.1 Future Work

The feature detection, extraction and especially the matching has to be improved, since erroneous matches have a critical impact on the implementation. Especially the CUDA brute-force matcher does not perform very well and should be analyzed. The parameters of the feature detector algorithm have to be adopted to increase the distinctness of the features. Since the distinctness of features in an image is inversely proportional to their number, the matched features should only be used for pose estimation, where a small number is sufficient. For triangulation, however, a large number of keypoints is advantageous for two reasons: the GPU implementation's speedup increases and the resulting point cloud consists of more points, and therefore, is recognizable easier. A possible way to generate a denser set of keypoints is by using an optical flow method.

In conclusion it can be said that implementing SfM using one camera on an embedded system is possible. It does, however, depend strongly on the algorithm used for feature detection, which on one hand, has to be as fast as possible, on the other hand should detect very distinct features. To find a suitable algorithm is essential for SfM.

References

Literature

[1] D. L. Baggio, S. Emami, D. M. Escriva, et al. *Mastering OpenCV*. 2012 (cit. on p. 27).

[2] J. Bloomenthal and J. Rokne. *Homogeneous Coordinates*. Department of Computer Science, The University of Calgary (cit. on p. 5).

[3] G. Bradski and A. Kaehler. *Learning OpenCV*. 1st ed. O'Reilly Media, Inc, 2008 (cit. on pp. 6, 7, 9, 10).

[4] NVIDIA Corporation. *CUBLAS Library User Guide*. 2014 (cit. on pp. 47, 50).

[5] NVIDIA Corporation. *CUDA C Programming Guide*. 2014 (cit. on pp. 42–46, 49, 56).

[6] R. B. Fisher et al. *Dictionary of Computer Vision and Image Processing*. 2nd ed. John Wiley & Sons Ltd, 2014 (cit. on p. 6).

[7] R. I. Hartley and P. Sturm. "Triangulation". In: *Computer vision and image understanding* 68.2 (1997), pp. 146–157 (cit. on pp. 26, 27).

[8] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. 2nd ed. Cambridge University Press, 2004 (cit. on pp. 7, 14–16, 22–24, 26).

[9] W. Hugemann. *Correcting Lens Distortion in Digital Photographs*. Tech. rep. Ingenieurbüro Morawski + Hugemann, 2010 (cit. on p. 10).

[10] V. Lepetit, F. Moreno-Noguer, and P. Fua. "EPnP: An Accurate O(n) Solution to the PnP Problem". In: *International Journal Computer Vision* 81.2 (2009). URL: http://cvlab.epfl.ch/EPnP/index.php (cit. on pp. 26, 40).

[11] NVIDIA. *Tegra K1 Mobile Processor Technical Reference Manual*. Oct. 2014. URL: http://developer.download.nvidia.com/assets/mobile/secure/TRM/TK1/TegraK1_TRM_DP06905001_public_v03p.pdf?autho=1430345487_4e2d8adb51e7bf1896aa7501ae64a901&file=TegraK1_TRM_DP06905001_public_v03p.pdf (cit. on p. 33).

[12] D. P. Robertson and R. Cipolla. “Practical Image Processing and Computer Vision”. In: ed. by M. Varga. John Wiley, 2009. Chap. 13. Structure From Motion (cit. on pp. 15, 16, 24).

[13] C. Schmid, R. Mohr, and C. Bauckhage. “Evaluation of Interest Point Detectors”. In: *Int. J. Comput. Vision* 37.2 (June 2000) (cit. on pp. 12, 13).

[14] R. Szeliski. *Computer Vision: Algorithms and Applications*. 1st. New York, NY, USA: Springer-Verlag New York, Inc., 2010 (cit. on pp. 4, 5, 7, 9–16, 22–24, 40).

[15] J. Weng, P. Cohen, and M. Herniou. “Camera calibration with distortion models and accuracy evaluation”. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 14.10 (Oct. 1992), pp. 965–980 (cit. on p. 9).

Online sources

[16] J. Y. Bouguet. *Camera Calibration Toolbox for Matlab*. 2013. URL: http://www.vision.caltech.edu/bouguetj/calib_doc (cit. on pp. 18, 27).

[17] *FLANN - Fast Library for Approximate Nearest Neighbors*. 2015. URL: http://www.cs.ubc.ca/research/flann/ (cit. on p. 37).

[18] Raspberry Pi Foundation. *Raspberry Pi*. 2015. URL: https://www.raspberrypi.org/ (cit. on p. 1).

[19] V. Hedau. *Epipolar Geometry*. 2010. URL: http://web.engr.illinois.edu/~dhoiem/courses/vision_spring10/lectures/Lecture22%20-%20Epipolar%20Geometry.pdf (cit. on p. 15).

[20] Itseez. *OpenCV*. 2015. URL: http://www.opencv.org (cit. on p. 34).

[21] Itseez. *OpenCV*. 2015. URL: http://itseez.com/OpenCV/ (cit. on p. 34).

[22] Itseez. *OpenCV-Contrib Repository*. 2015. URL: https://github.com/Itseez/opencv_contrib (cit. on p. 34).

[23] A. D. Jepson and D. J. Fleet. *Epipolar Geometry*. 2006. URL: http://www.cs.toronto.edu/~jepson/csc420/notes/epiPolarGeom.pdf (cit. on p. 14).

[24] L. W. Kheng. *Camera Models and Imaging*. URL: http://www.comp.nus.edu.sg/~cs4243/lecture/camera.pdf (cit. on p. 5).

[25] P. D. Kovesi. *MATLAB and Octave Functions for Computer Vision and Image Processing*. URL: http://www.csse.uwa.edu.au/$%5Csim$pk/research/matlabfns (cit. on p. 22).

[26] Logitech. *Logitech HD Pro Webcam C920*. 2015. URL: http://www.logitech.com/en-hk/product/hd-pro-webcam-c920 (cit. on p. 18).

[27] NVIDIA. *Linux For Tegra R21.3*. 2015. URL: https://developer.nvidia.com/linux-tegra-r213 (cit. on p. 34).

[28] NVIDIA. *Official Wiki for the Jetson TK1*. 2015. URL: http://elinux.org/Jetson_TK1 (cit. on p. 33).

[29] NVIDIA. *Tegra K1 Next-Gen Mobile Processor*. 2015. URL: http://www.nvidia.com/object/tegra-k1-processor.html (cit. on p. 33).

[30] J. van Oosten. *CUDA Memory Model*. 2011. URL: http://www.3dgep.com/cuda-memory-model/ (cit. on pp. 44, 45).

[31] E. W. Weisstein. *Rotation Matrix*. 2015. URL: http://mathworld.wolfram.com/RotationMatrix.html (cit. on p. 24).

[32] WolfWings. *Barrel distortion visual example*. 2008. URL: http://en.wikipedia.org/wiki/File:Barrel_distortion.svg (cit. on p. 9).

[33] WolfWings. *Pincushion distortion visual example*. 2008. URL: http://en.wikipedia.org/wiki/File:Pincushion_distortion.svg (cit. on p. 9).