A Modular Framework for Image-based 3D Reconstruction

Vorgelegt von
Carsten Brandt
aus Alfeld (Leine).

Von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Master of Science
- M.Sc. -
genehmigte Abschlussarbeit.

Gutachter : Prof. Dr.-Ing. Olaf Hellwich
Betreuer : Dr.-Ing. Ronny Hänsch

Eidesstattliche Versicherung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 27. März 2017 .....

Carsten Brandt

Acknowledgements

First of all I'd like to thank my beloved wife and kids for giving me the time to work on this thesis, and also the rest of my family for their support. Thanks for your patience and withstanding the time with me.

Also thanks to Boudewijn Vahrmeijer and Alexander Makarov for proof reading and giving me valuable feedback.

Thanks to my adviser Ronny Hänsch for the idea and for giving me the opportunity to work on this interesting topic.

Contents

1 Introduction **1**

2 Image-based 3D Reconstruction **3**
2.1 Structure from Motion . . . 4
2.1.1 Feature Extraction . . . 4
2.1.2 Feature Matching . . . 6
2.1.3 3D Geometry Estimation . . . 7
2.2 Dense Reconstruction . . . 15
2.3 Surface Reconstruction . . . 16

3 Review of Processing Chains **19**
3.1 Survey of Existing Processing Chains . . . 20
3.1.1 SfM Processing Chains . . . 20
3.1.2 MVS Implementations . . . 25
3.1.3 Surface Reconstruction Algorithms . . . 25
3.1.4 Further Processing . . . 26
3.2 Generic SfM Processing Chain and Data Structures . . . 26
3.2.1 Processing Steps . . . 27
3.2.2 Data Structures . . . 29
3.3 Software Review . . . 32

4 Design and Implementation of the Framework **35**
4.1 Data Model . . . 36
4.1.1 Module . . . 36
4.1.2 Input and Output Data . . . 37
4.1.3 Processing Chain . . . 38
4.1.4 Summary . . . 39
4.2 Execution of a Processing Chain . . . 39
4.2.1 Module Runner . . . 40
4.2.2 Data Manager . . . 41
4.2.3 Mapping on List Data Types . . . 42
4.3 User Interfaces and Configuration . . . 42
4.3.1 The Processing Chain Configuration File . . . 42
4.3.2 Graphical User Interface . . . 44
4.3.3 Command Line Interface . . . 49
4.4 Extending the Framework . . . 50
4.4.1 Implementing a Module . . . 50
4.4.2 Implementing a Data Type . . . 52

5 Implementation of SfM Data Types and Modules **55**
5.1 Preprocessing and Feature Extraction . . . 56
5.1.1 Data Structures . . . 56
5.1.2 Module Implementation . . . 56
5.1.3 Visualisations . . . 57
5.2 Feature Matching . . . 57
5.2.1 Data Structures . . . 57
5.2.2 Module Implementation . . . 58
5.2.3 Visualisation . . . 58
5.3 Filtering of the Image Graph . . . 59
5.3.1 Geometric Verification with RANSAC . . . 60
5.4 Geometry Estimation and Bundle Adjustment . . . 60
5.4.1 Data Structures . . . 61
5.4.2 Module Implementation . . . 61
5.4.3 Visualisation . . . 61
5.5 Dense Reconstruction . . . 62
5.5.1 Module Implementation . . . 62
5.5.2 Visualisation . . . 63
5.6 Surface Reconstruction . . . 64

6 Application to Example Use Cases **65**
6.1 Datasets . . . 65
6.2 Workflow . . . 65
6.3 The Processing Chain . . . 67
6.4 Results . . . 68

7 Conclusion **71**
7.1 Limitations and Future Work . . . 71

Appendix **73**

A Open Source **75**

B The PLY File Format **77**

C DVD with Code, Data, and Results **79**

Bibliography **81**

List of Figures

2.1 Point Detection using the Förstner Operator 5
2.2 The Pinhole Camera Model 8
2.3 External and Internal Camera Parameters 9
2.4 Depth Information from Two Cameras 9
2.5 Two View Geometry: Stereo Normal Case 10
2.6 Two View Geometry: Epipolar Geometry 11
2.7 Sparse and Dense Point Cloud 15
2.8 Triangle Mesh 16

3.1 3D Reconstruction Results from Related Work 21
3.2 Example of Outliers in Feature Matching 23
3.3 High Level Overview of a 3D Reconstruction Processing Chain . . . 27
3.4 Steps and Data Structures of a Generic SfM Processing Chain 28

4.1 Concept of a Module 36
4.2 UML Class Diagram of Data Types 37
4.3 Example Processing Chain with 3 Steps 38
4.4 Example of a Non-Linear Processing Chain 39
4.5 UML Class Diagram of the Processing Chain Model 40
4.6 Organisation of the Framework Components 43
4.7 Screenshot of the Framework GUI Layout 46
4.8 Visual Feedback in the GUI 49

5.1 Visualisation of Key Points 57
5.2 Visualisation of an Image Graph 58
5.3 Visualisation of Key Point Matches 59
5.4 Example of Wrong Key Point Matches 59
5.5 Visualisation of the Epipolar Geometry of an Image Pair 60
5.6 Visualisation of Camera Positions 62
5.7 Visualisation of a Point Cloud 63

6.1 Sparse Reconstruction and Camera Visualisation 66
6.2 Reconstruction of the fountain-P11 Dataset 67
6.3 Surface Mesh of the “Der Hass” Dataset 68
6.4 Dense Reconstruction Point Cloud of the “Herz-Jesu-P8” Dataset . . 69

Chapter 1

Introduction

Image-based 3D reconstruction has become a popular research topic within recent years and finds application in a wide variety of fields, such as the reconstruction of single buildings [Wefelscheid 2011], parts of a city [Frahm 2010, Agarwal 2011] or even whole regions of a city [Over 2010], the generation of CAD models from real world objects [Chen 2005], as well as real-time 3D scene approximation from video [Kauff 2007] and many more.

It has been shown that image-based 3D reconstruction can achieve the same accuracy as 3D reconstruction using laser scanners (LIDAR) [Strecha 2008]. Using images also has the advantage that texture information can be captured, which is not directly available when using laser scanners.

However, in contrast to reconstruction with laser scanners, the path from images to a reconstructed 3D model is long and complex and requires a great amount of different algorithms to be applied in a processing chain to work together. Also different techniques must be applied in different circumstances so in most applications it would be useful to be able to use an existing processing chain and exchange parts of it with a different method, which fits better for a specific use case [Wefelscheid 2011].

When working on specific parts of the whole process of 3D reconstruction, it is still necessary to create a complete processing chain to evaluate the effect of changes in a specific part of the algorithm on the final 3D reconstruction result.

Existing 3D reconstruction implementations however are not designed to be flexible in the way they run, but focus on solving their concrete problem statement well. This makes it hard to reuse existing implementations in different contexts. They either come as one binary that implements everything [Wu 2013], or use proprietary data formats, so that a conversion is necessary to use another tool for further processing [Snavely 2006].

The goal of this thesis is to develop a modular framework that allows the creation of processing chains for image-based 3D reconstruction in a flexible way, so that parts of the chain can be adjusted or replaced easily to adopt a processing chain to a different use case.

Additionally it should further simplify the task of developing a processing chain by providing methods for debugging the intermediate results of algorithms using visualisation methods.

The flexibility should not come at the cost of performance, so the design of the framework should not limit the algorithms developed with it, but allow the application to large scale reconstruction problems.

To achieve the goal of creating the framework, the first step is to understand the basics of image-based 3D reconstruction, which are described in Chapter 2. After that a review of existing implementations follows in Chapter 3, to extract the common processing steps and data structures of 3D reconstruction methods. Chapter 4 then contains a description of the framework's concept and implementation, which are applied to 3D reconstruction in Chapter 5. In Chapter 6 the framework is applied to real world use cases to demonstrate and validate its functionality. A conclusion is given in Chapter 7.

Chapter 2

Image-based 3D Reconstruction

Contents

2.1 Structure from Motion 4
2.1.1 Feature Extraction 4
2.1.2 Feature Matching 6
2.1.3 3D Geometry Estimation 7
2.2 Dense Reconstruction 15
2.3 Surface Reconstruction 16

In this section I will introduce the process of image-based 3D reconstruction by breaking it down into single, isolated steps and describing the theoretical background of the applied algorithms.

Image-based 3D reconstruction is the process of obtaining 3D structure from the real world and creating a digital model of it, based on digital images taken from a scene [Hartley 2003]. It differs from reconstruction via laser scanners (LIDAR) in the sense, that it is not only able to obtain the 3D structure, but also provides the texture of an object, i.e. it preserves the colour information of the captured object [Strecha 2008]. It has also been shown that image-based 3D reconstruction is able to be performed with the same accuracy as with laser scanners [Strecha 2008]. Another benefit is that dependent on the case, existing images can be used, which might not have been taken with the goal of 3D reconstruction in mind, because the methods used, do not require a calibration step [Hartley 2003]. Applications of this have been shown by [Frahm 2010], [Agarwal 2011] and others by taking images from photo sharing platforms on the internet to create 3D models of parts of a city.

The key part of image-based 3D reconstruction is Structure from Motion (SfM), which means to infer 3D structure from a moving camera, i.e. images taken from different positions in 3D space [Longuet-Higgins 1981]. SfM deals with the task of estimating the relative orientation between camera positions in 3D space from images, without additional information. It also estimates a sparse set of 3D points of the observed scene. After SfM further methods can be applied to generate a dense point cloud of the scene which can afterwards be used to obtain a 3D mesh.

In the following I will describe the techniques for Structure from Motion (Section 2.1), estimation of a dense point cloud from the known 3D geometry (Section 2.2), and obtaining a 3D mesh from point clouds, which is known as Surface Reconstruction (Section 2.3).

2.1 Structure from Motion

In Structure from Motion (SfM), the path of getting from the initial input images to the 3D structure involves a chain of different steps. There is a core set of steps that are involved in every application, but there are also different methods that involve different combinations of algorithms, dependent on the use case. In this section I will cover the theory of the basic algorithms of SfM. A detailed review of different methods applied in different practical applications can be found in Section 3.1.

The initial input to the SfM processing chain is a set of $n \geq 2$ images, while the result is the estimation of the original position, from which the image has been taken, as well as a sparse set of 3D points of the scene [Hartley 2003].

The common procedure of a SfM processing chain can be broken down into three major steps, which are (1) feature extraction and description, (2) feature matching, and (3) estimation of the 3D geometry.

2.1.1 Feature Extraction

SfM algorithms are able to work on an unordered set of images without prior knowledge or additional information [Zhang 1998]. To find related images that picture the same part of a scene, SfM involves feature extraction as the first step. The extracted features can then be compared to find images that capture the same part of a scene [Zhang 1995].

Features used for SfM are preferably point features called *key points* or *interest points*, because in the later steps corresponding point pairs can be used to determine the 3D coordinates of a certain point in the world [Zhang 1998, Schreer 2005]. The images are usually taken from different view points of a scene, which includes different viewing angles as well as different distances. To be able to compare point features between those images, the feature extractor must be invariant to rotation, scale, and affine transformation [Gil 2010].

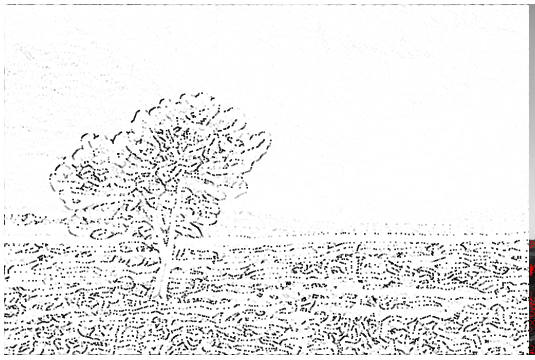
The extraction of point features includes two steps: (1) the detection of the points, and (2) their description. The detection step is for finding coordinates in the image that have certain properties. The description step creates a descriptor vector based on the point's pixel environment and should allow to find the point in other images by comparing these vectors [Gil 2010].

A commonly used point detector and descriptor in SfM approaches is SIFT (Scale Invariant Feature Transform) originally described by [Lowe 2004]. This descriptor is robust against up to 30° rotation between the image planes [Irschara 2009], partially invariant to change in illumination, and has been shown to perform best compared to other descriptors, when applied in the field of 3D reconstruction [Mikolajczyk 2005].

While SIFT covers both, the point detection and description, it is possible to use the descriptor part on points detected by other algorithms. This is done by [Wefelscheid 2011], who use the Förstner Operator [Förstner 1987] for detection and SIFT as the descriptor, to exploit the advantage of the Förstner Operator on images taken from man made architecture which usually contains sharp edges.

(a) An image of a tree, obtained by combining the intensity of three channels (RGB colour) into one channel (grey scale).

(b) Visualisation of the trace of the structure tensor, indicating areas with large change in intensity (black).



(c) Visualisation of the isotropy, which measures the uniformity of the gradient directions in the neighbourhood of a pixel. High values (black) indicate less uniformity, i.e. candidates for an interest point.

(d) The resulting interest points' positions painted as red circles on the original image.

Figure 2.1: Process of point detection using the Förstner Operator [Förstner 1987].

Another commonly used point detector is the Harris Corner and Edge Detector described by [Harris 1988]. It provides a method for point detection as well as edge detection. Edges and point features are very similar as they both are characterised by the change of intensity in an image. An interest point is considered in image regions, in which the gradient of image intensity is large in both, horizontal and vertical direction.

To provide a general idea on how point detection works, I will show a short summary of the techniques used by the Förstner Operator [Förstner 1987].

The first step is the computation of directional gradients, which is the change in intensity in x and y direction. Based on the gradient values g_x and g_y a so called *structure tensor* is defined:

$$A = \sum_W gg^T = \begin{bmatrix} \sum_W g_x^2 & \sum_W g_x g_y \\ \sum_W g_y g_x & \sum_W g_y^2 \end{bmatrix} \tag{2.1}$$

In Equation 2.1, W denotes a window over the pixel neighbourhood, which can be a box window using equal weights for all pixels in the neighbourhood or be based on a Gaussian kernel which applies different weights, so pixels which are further away count less to the gradient value than pixels which are near.

As the structure tensor provides information about the gradients in the pixel neighbourhood two properties of the tensor can be extracted that describe whether the pixel is a candidate for an interest point. The weight w denotes the strength of the gradients in the neighbourhood and is calculated based on the trace and determinant of the structure tensor $w = \frac{det(A)}{tr(A)}$. A visualisation of the trace $tr(A)$ is shown in Figure 2.1b. Another measurement is the isotropy q, which measures the uniformity of the gradient directions in the neighbourhood $q = \frac{4det(A)}{tr(A)^2}$. High values indicate low uniformity and thus indicate points that are candidates for an interest point, because a point means change of the gradient in two directions. The isotropy value is visualised in Figure 2.1c.

Interest points are found by calculating the weight w, and applying a non-max-suppression and a threshold for it. The same non-max-suppression and threshold is applied for the isotropy. So the points that have big change in gradient (weight) and whose gradient directions are not uniform in the neighbourhood (isotropy) are considered interest points. The detected points are highlighted in Figure 2.1d.

For further information on the topic, the reader may refer to [Gil 2010], which provides a good summary of point descriptors and their properties, and [Mikolajczyk 2005] for a performance evaluation of point detectors under different image transformations.

2.1.2 Feature Matching

The next step after feature detection is feature matching, which aims to test whether a point of a scene, that can be seen in one image, also exists in other images [Zhang 1995]. The result of feature matching is a graph of images, where images are connected that share some part of the scene. In the graph, images are the vertices and edges connect images that picture the same part of the scene. For each image pair a set of corresponding point pairs is detected [Wefelscheid 2011]. Additionally to that, an algorithm may extract the points that are seen in multiple images [Snavely 2006].

For finding image pairs, a correspondence search is performed by comparing the feature descriptor vectors. For this a similarity measure needs to be defined which can be for example the euclidean distance in the space of the descriptor vector [Wefelscheid 2011]. This space has 128 dimensions in case of SIFT [Lowe 2004], but may also have other dimensionality for other point descriptors [Gil 2010]. To find similar points, the image space can be searched using a nearest neighbour search approach as described by [Snavely 2006]. An overview over different approaches in different application scenarios is later given in Section 3.1.1.1.

For a complete graph, all images need to be compared with each other resulting in a complexity of this step of $O(n^2k^2)$ with n images and k features each

[Schönberger 2016a]. Thus when working with a large number of images, clustering approaches and pre-filtering need to be applied which are covered in Section 3.1.1.1.

The matches of feature points usually contain a large amount of outliers [Schönberger 2016a] so that after the matching a filter needs to be applied to make sure only matches are kept that are geometrically feasible. This is usually done after geometry estimation using statistical outlier removal methods like Random Sample Consensus [Fischler 1981], which will be explained in Section 2.1.3.5.

While most SfM implementations are designed to work with unordered images without additional information, the matching process can be improved by using prior knowledge about the camera positions and thus the relations of images to each other, when this information is available, as described by [Stathopoulou 2015].

2.1.3 3D Geometry Estimation

Given the image graph including a set of correspondent point pairs for each image pair, the 3D geometry can be estimated. For this at least two images need to be used. The geometry between two images can be described by the *Epipolar Geometry* and is fully described by the *Fundamental Matrix* $\mathbf{F}$, which can be estimated based on known point correspondences [Schreer 2005, Hartley 2003]. The geometry estimation between image triplets can also be computed, which results in a descriptor that is called *Trifocal Tensor* $\mathbf{T}$ [Hartley 2003].

2.1.3.1 The Pinhole Camera Model

To be able to infer information about points in 3D space from images, a model of the camera needs to be defined that was used to transform the light rays emitted by points from the scene onto the image.

The way light travels through lenses of a camera is usually very complicated to model, so a simpler model is desirable. For structure from motion we are using the pinhole camera model, which proposes the assumption that all light rays go through one single point, called the camera centre $\mathbf{C}$ [Hartley 2003, Schreer 2005]. Using this model, the geometric relations between pixel coordinates and coordinates in the 3D scene can be described by a linear transformation [Olague 2002].

Figure 2.2 illustrates the camera model by showing, how a point in the world is pictured on the image plane of the pinhole camera. The image on the image plane is created by mapping the light ray by a point reflection on the camera centre and is therefor inverted. For better understanding, the image plane is often shown in front of the camera centre, which is equivalent because of the point reflection.

The properties of the camera model can be described using a set of parameters which influence the way a point is projected onto the image plane. We distinguish between external and internal camera parameters [Schreer 2005].

External camera parameters are the position of the camera centre described by the vector $\mathbf{t} = \left[X, Y, Z\right]^T$, and its orientation described by spatial rotation angles Ω, Φ, Ψ in 3D space which may also be represented in terms of a rotation matrix $\mathbf{R}$.

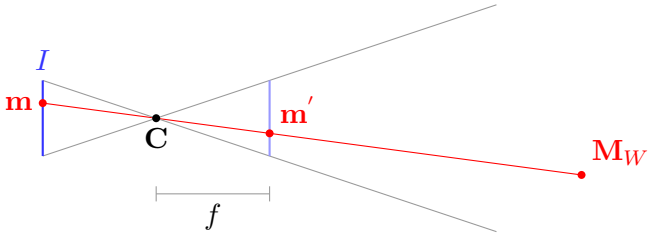


Figure 2.2: Projection $\mathbf{m}$ of a point $\mathbf{M}_W$ in the world onto the image plane I in the pinhole camera model. Light rays go through the camera centre $\mathbf{C}$, indicating a point reflection. This causes the image in distance f (focal length), in front of, and behind the camera centre to be the same except inversion.

A point in world coordinates $\mathbf{M}_W$ can be transformed into the camera coordinate system, which has the origin in the camera centre $\mathbf{C}$ by the following transformation:

$$\mathbf{M}_C = \mathbf{R}\mathbf{M}_W + \mathbf{t} \tag{2.2}$$

$\mathbf{M}_C$ expresses the position of the point relative to the camera [Schreer 2005].

Internal camera parameters describe the inner geometry of the camera and describe how points from the world are mapped to the image plane. Dependent on the camera, this can be modelled with different parameters. The basic internal camera parameters are the focal length f, which is the distance between camera centre and the image plane, the aspect ratio of the pixels on the image plane $\gamma = \frac{m_x}{m_y}$, and the position of the principal point (x_0, y_0) [Schreer 2005, Hartley 2003]. More complex models with more parameters for linear and also non-linear distortions are described by [Hartley 2003], but not covered here. Figure 2.3 shows the relation of these parameters in the camera model. These internal camera parameters form the calibration matrix $\mathbf{K}$ [Schreer 2005]:

$$\mathbf{K} = \begin{bmatrix} fm_x & 0 & x_0 \\ 0 & fm_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

To define the projection of a point in 3D space $\mathbf{M}_W$ into its projection on the image plane $\mathbf{m}$ the projection matrix $\mathbf{P}$ is defined based on Equations 2.2 and 2.3 in the following way [Schreer 2005]:

$$\mathbf{P} = \underbrace{\mathbf{K}}_{\text{internal}} \cdot \underbrace{[\mathbf{R}\ \mathbf{t}]}_{\text{external}} \tag{2.4}$$

This results in the following correlation of $\mathbf{m}$ and $\mathbf{M}_W$:

$$\tilde{\mathbf{m}} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \mathbf{P} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \mathbf{P}\tilde{\mathbf{M}}_W \tag{2.5}$$

In the above equation, $\tilde{\mathbf{M}}_W$ denotes the point in 3D homogeneous coordinates, and $\tilde{\mathbf{m}}$ is its projection on the image plane in 2D homogeneous coordinates.

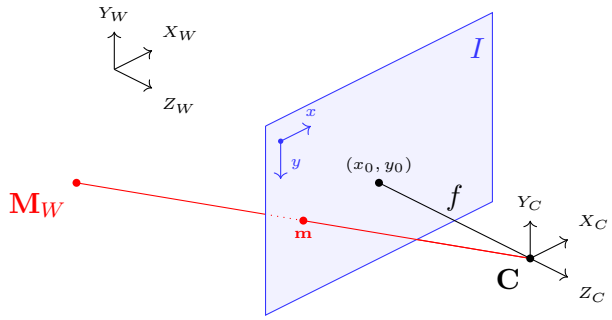


Figure 2.3: Visualisation of the internal and external camera parameters, as well as the three coordinate systems: (1) World coordinate system (X_W, Y_W, Z_W), (2) Camera coordinate system (X_C, Y_C, Z_C), (3) Image coordinates (x, y). Internal camera parameters are the focal length f, which is the distance between camera centre $\mathbf{C}$ and the image plane I. The position of the principal point (x_0, y_0) describes the position where the Z-axis of the camera coordinate system intersects with the image plane. Additionally the projection of a point $\mathbf{M}_W$ in the world, as point $\mathbf{m}$ on the image plain is shown, where $\mathbf{m}$ are the pixel coordinates of $\mathbf{M}_W$'s projection.

2.1.3.2 Two View Geometry

Given one image, to obtain the 3D structure of the scene, the depth information for each point in the image is missing. A single image provides the information about the line the point is located on in 3D space. This results from the pinhole camera model, which assumes light to go in straight lines through the camera centre $\mathbf{C}$. By using a second image, with knowledge of the pixel coordinates of the same scene point in that image, the 3D position of the point in 3D space can be triangulated. The idea is shown in Figure 2.4.

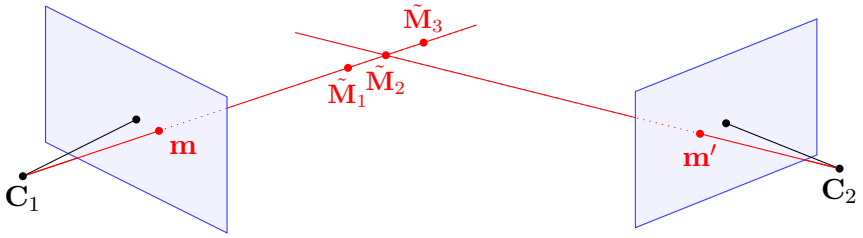


Figure 2.4: Given a point $\mathbf{m}$ in one image of camera $\mathbf{C}_1$ only provides the information on which ray it is located on in 3D space. Its original 3D position is unknown and could be anywhere on the ray, e.g. $\tilde{\mathbf{M}}_1$, $\tilde{\mathbf{M}}_2$, or $\tilde{\mathbf{M}}_3$. Only when we know the same scene point's position in another image $\mathbf{m}'$, we can estimate its original position by intersection of the rays, which is $\tilde{\mathbf{M}}_2$ in this case.

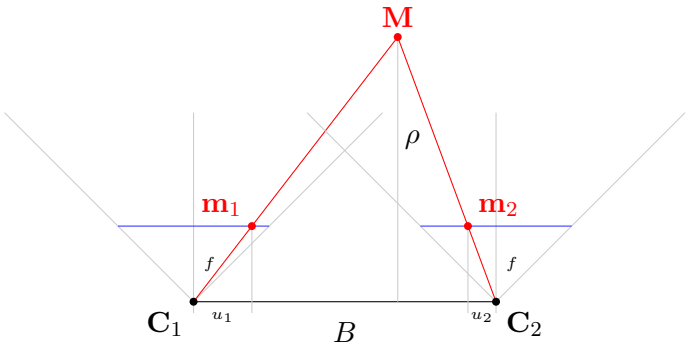


Figure 2.5: The geometric relations between a 3D point $\mathbf{M}$ projected on images of two cameras in the stereo normal case. Both image planes are parallel to and in the same distance f to the baseline B. This results in a relation between them, that can be described on a 2D plane to obtain the depth ρ. The disparity of the projection points $\mathbf{m}_1$ and $\mathbf{m}_2$ is $\delta = u_1 - u_2$.

To be able to perform this triangulation, the geometric relation between both cameras needs to be described. The relation between the two cameras can be described as the transformation of the camera centre of the first camera $\mathbf{C}_1$ by rotation and translation to the second camera position [Schreer 2005]:

$$\mathbf{C}_2 = \mathbf{R}\mathbf{C}_1 + \mathbf{t} \tag{2.6}$$

The simplest geometric case for this is the *stereo normal case*, where two images are taken from two camera positions where the relation between both camera positions is only a translation without rotating ($\mathbf{R} = \mathbf{I}$). In this case the depth of the point can be calculated by the following relation [Schreer 2005]:

$$\rho = \frac{B \cdot f}{\delta \cdot d_u} \tag{2.7}$$

In Equation 2.7, B is the length of the baseline between both cameras, i.e. the length of $\mathbf{t}$. f is the focal length, which is assumed to be the same in both cameras for this scenario, δ describes the disparity of two pixels in x direction, their y position is assumed to be the same, and d_u is the resolution of the CCD sensor in $\frac{\text{mm}}{\text{pixel}}$. See Figure 2.5 for a visualisation of the geometric relations behind this equation.

For the general case ($\mathbf{R} \neq \mathbf{I}$) a more complex model is needed to describe the geometric relation between 3D points and their projection on two images. This description is called *Epipolar Geometry* [Schreer 2005]. The Epipolar Geometry describes the relation between a point $\mathbf{M}_W$ in the world, and the camera centres $\mathbf{C}_1$ and $\mathbf{C}_2$, which together form a plane in 3D space, the *epipolar plane*. The connection line between the two camera centres is called the *baseline* B. The projection of $\mathbf{M}_W$ in the first image is denoted as $\mathbf{m}_1$, and for the second image $\mathbf{m}_2$. In the general case, there is an intersection point of the baseline with the image planes,

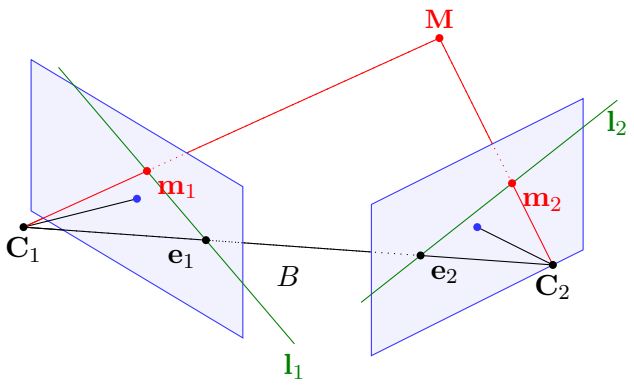


Figure 2.6: The relation of a point in 3D space with its projections onto images of two cameras in the Epipolar Geometry. The camera centres $\mathbf{C}_1$ and $\mathbf{C}_2$ are connected by the baseline B. The intersection of the baseline with the image planes introduces the epipoles $\mathbf{e}_1$ and $\mathbf{e}_2$. Using the projection of point $\mathbf{M}$ on the image as $\mathbf{m}_1$ and $\mathbf{m}_2$ the corresponding epipolar lines $\mathbf{l}_1$ and $\mathbf{l}_2$ can be derived.

which is called *epipole*. We have $\mathbf{e}_1$ for the epipole in the first image and $\mathbf{e}_2$ for the second image. The epipole may be located outside of the visible region of an image. In the stereo normal case the epipoles are located at infinity. The line connecting $\mathbf{e}_1$ with $\mathbf{m}_1$ is called epipolar line $\mathbf{l}_1$, there is $\mathbf{l}_2$ for the second image respectively [Schreer 2005, Hartley 2003]. An illustration of the relations in the Epipolar Geometry can be seen in Figure 2.6.

The Epipolar Geometry can be described in terms of a matrix, which is called Fundamental Matrix $\mathbf{F}$ by the following relation [Schreer 2005]:

$$\mathbf{l}_2 = \mathbf{F}\tilde{\mathbf{m}}_1 \quad \text{and} \quad \mathbf{l}_1 = \mathbf{F}^T\tilde{\mathbf{m}}_2 \tag{2.8}$$

This means a point from one image can be mapped to a line in the other image, which contains the corresponding point in that image.

When the Fundamental Matrix has been estimated, the projection matrices can be extracted as follows [Hartley 2003]:

$$\mathbf{P}_1 = \begin{bmatrix}\mathbf{I} & \mathbf{0}\end{bmatrix}\ ,\ \mathbf{P}_2 = \begin{bmatrix}[\mathbf{e}_2]_\times\mathbf{F} \,\middle|\, \mathbf{e}_2\end{bmatrix} \tag{2.9}$$

$[\mathbf{e}]_\times$ is a short notation for the following skew symmetric matrix [Schreer 2005, p. 71]:

$$\begin{bmatrix} 0 & -e_z & e_y \\ e_z & 0 & -e_x \\ -e_y & e_x & 0 \end{bmatrix} \text{ with } [\mathbf{e}]_\times = -[\mathbf{e}]_\times^T. \tag{2.10}$$

2.1.3.3 Three View Geometry

Similar to the two view geometry, the geometry between three images can be described in terms of a Trifocal Tensor. This is an extension of the Epipolar Geometry and involves Fundamental Matrices between all three images, $\mathbf{F}_{12}$, $\mathbf{F}_{23}$, and $\mathbf{F}_{31}$.

I am not describing the Trifocal Tensor here as it is a more advanced concept and not necessary to understand the theory described in the following sections. The interested reader may refer to [Schreer 2005, p. 184ff] or [Hartley 2003, p. 365ff] for a detailed description of this concept.

2.1.3.4 Estimation of F

In the last sections I have described the geometric relations in the Epipolar Geometry and introduced the Fundamental Matrix $\mathbf{F}$. The goal of SfM is the estimation of the Epipolar Geometry given corresponding point pairs as they have been extracted in Section 2.1.1 and matched in Section 2.1.2.

Based on these corresponding point pairs we can now estimate the Fundamental Matrix, which provides all information about the geometry between the images as well as external and internal camera parameters up to a scaling factor [Hartley 2003]. Following from Equation 2.8, for two corresponding points, a point $\tilde{\mathbf{m}}_2$ must be located on the corresponding epipolar line, induced by the point in the other image $\mathbf{l}_2 = \mathbf{F}\tilde{\mathbf{m}}_1$. For a point and a line in homogeneous coordinates their cross product is zero, if the point is located on the line, so the following relation exists [Schreer 2005]:

$$\tilde{\mathbf{m}}_2^T \mathbf{F} \tilde{\mathbf{m}}_1 = 0 \tag{2.11}$$

In Equation 2.11, which is also called epipolar constraint, the corresponding points, $\tilde{\mathbf{m}}_1$ and $\tilde{\mathbf{m}}_2$ are known, while $\mathbf{F}$ is unknown. With enough point pairs, a linear equation system can be formed which can then be used to estimate the unknown values of $\mathbf{F}$ [Schreer 2005]:

$$\begin{aligned} &\mathbf{u}_i^T \mathbf{f} = 0\,, \text{ with} \\ &\mathbf{u}_i = \begin{bmatrix} x_{i2}x_{i1} & x_{i2}y_{i1} & x_{i2} & y_{i2}x_{i1} & y_{i2}y_{i1} & y_{i2} & x_{i1} & y_{i1} & 1 \end{bmatrix}^T \\ &\mathbf{f} = \begin{bmatrix} f_{11} & f_{12} & f_{13} & f_{21} & f_{22} & f_{23} & f_{31} & f_{32} & f_{33} \end{bmatrix}^T \end{aligned} \tag{2.12}$$

In Equation 2.12 x_{ik} and y_{ik} refer to the x and y coordinate of a point $\mathbf{m}_{ik}$ respectively, $k \in [1, 2]$. So for n corresponding point pairs we get the following linear equation system [Schreer 2005]:

$$\mathbf{U}_n \mathbf{f} = 0, \text{ with } \mathbf{U}_n = \begin{bmatrix} \mathbf{u}_1^T \\ \vdots \\ \mathbf{u}_n^T \end{bmatrix} \tag{2.13}$$

To solve the equation system, at least 8 point pairs are needed [Longuet-Higgins 1981]. Given the fact that correspondences are often inexact or even contain outliers, it is desirable to use even more point pairs and form

a quadratic system to estimate the Fundamental Matrix to minimise the error [Schreer 2005]:

$$\min_{\mathbf{f}} \|\mathbf{U}_n\mathbf{f}\|^2 \tag{2.14}$$

When solving the linear equation system, two additional constraints have to be considered: (1) the equation system has the trivial solution of $\mathbf{f} = \mathbf{0}$, and (2) the Fundamental Matrix has a rank of 2, so the singularity constraint needs to be enforced [Schreer 2005, p. 82]. Using these constraints, an algorithm for estimating the Fundamental Matrix using 7 point pairs is explained by [Schreer 2005].

For the implementation of an algorithm it is also important to perform conditioning on the input values before estimating the Fundamental Matrix to achieve stable results. A detailed description of this concept can be found in [Schreer 2005] and [Hartley 1997].

2.1.3.5 Geometric Verification and Outlier Removal

After the estimation of the Fundamental Matrix from a set of point pairs, it is now possible to verify the result by testing whether the epipolar constraint, as given in Equation 2.11, is fulfilled for all point matches. At this point outlier removal techniques like RANdom SAmple Consensus (RANSAC) [Fischler 1981] can be applied to clean up the point matches from outliers.

A RANSAC approach for SfM works by selecting random samples of the minimum required size (7 or 8 point pairs dependent on the estimation algorithm) from all matched point pairs, and estimates the Fundamental Matrix from it. Afterwards all points, that are not in a sample, are tested whether they agree with the estimated Fundamental Matrix or not, i.e. whether the epipolar constraint is fulfilled for them [Schreer 2005]. If a sample contains outliers, a large amount of points will reject it. This way it is possible to find a sample that does not contain outliers which is eventually accepted as the result of the RANSAC algorithm [Fischler 1981]. RANSAC is very robust even when confronted with a lot of outliers and is therefor used in many SfM applications [Raguram 2008].

2.1.3.6 Triangulation of 3D Points

An estimation of $\mathbf{F}$ provides the geometric relation between the cameras and allows us to verify the point matches using the epipolar constraint. It does not include the positions of the 3D points, which are yet to be found. For estimating the positions of the points in 3D space, the projection matrices of both cameras need to be known. These can be extracted from the Fundamental Matrix as shown in Equation 2.9.

Because the geometry estimation is not exact when working with real data, the position of a 3D point must be approximated. A good approximation for the original position is the centre of the shortest line segment that connects the rays that go from the points projection on the image through the camera centre [Hartley 2003]. To find these positions a quadratic minimisation problem can be formulated, which is based on the fact that the cross product of two points induces the line between

them. For the projection of a 3D points on the image as defined in Equation 2.5 this can be used to create the following equation, as for equal points, the line between them does not exist so their cross product must be zero [Schreer 2005]:

$$\tilde{\mathbf{m}}_i \times \mathbf{P}_i \tilde{\mathbf{M}} = \mathbf{0} \tag{2.15}$$

There are two different approaches to solve the linear equation system described by [Schreer 2005]. One is the solution of the linear equation system using Direct Linear Transformation (DLT), the other is an optimisation problem solved by the least squares approach.

The triangulation coordinates are obviously unrelated to the real worlds coordinates and may differ in rotation and translation. The reconstruction is also ambiguous in scale, that means that it is impossible from a reconstruction of images, without additional information, to infer information about the original size of the scene in the real world [Hartley 2003].

For unknown camera calibration, the triangulation is even ambiguous up to a projective transformation. So in order to reduce the ambiguity, additional information, which for example can be in form of other images, has to be added. For more information on this topic, the interested reader may refer to [Hartley 2003, p. 262ff].

2.1.3.7 Bundle Adjustment

After the estimation of the Fundamental Matrix from a set of point pairs, it is possible to verify the result by calculating for each point pair the distance of each point from its corresponding epipolar line induced by the point in the other image. It is also possible to calculate the distance of the rays that go from points in the image into 3D space. Both of these distances should be zero in an optimal solution, so an optimisation problem can be formulated to reduce the overall error for a set of images, their estimated Fundamental Matrix and 3D point coordinates. Such a procedure is called Bundle Adjustment [Triggs 1999].

Bundle Adjustment is typically applied after estimation of the Fundamental Matrix and the 3D points to optimise the final result by reducing the overall error in the solution. It is mathematically formulated as [Hartley 2003]:

$$\min_{\hat{\mathbf{P}}^i, \hat{\mathbf{M}}_j} \sum_{ij} d(\hat{\mathbf{P}}^i \hat{\mathbf{M}}_j, \mathbf{m}_j^i)^2 \tag{2.16}$$

In Equation 2.16, $d(x, y)$ is the distance of two points in an image and describes the distance of the projection of an estimated point position $\hat{\mathbf{M}}$ onto the image plane, and the original pixel position from which $\hat{\mathbf{M}}$ had been estimated $\mathbf{m}$. This procedure is performed for all points and images together.

Figure 2.7: Comparison of a sparse reconstruction point could (left) and a dense reconstruction (right) of a building front. The example has been created using the Bundler SfM software [Snavely 2006] and PMVS2 [Furukawa 2010b] from a dataset of 29 images.

2.2 Dense Reconstruction

The next step towards a final 3D model after SfM is dense modelling or dense reconstruction using Multi-View-Stereo (MVS). SfM only provides a sparse reconstruction by providing 3D coordinates for the corresponding point pairs, but provides no information for the parts of the scene between these points. MVS algorithms provide methods for reconstructing a dense point cloud from images for which the projection matrix P has been estimated [Furukawa 2010b]. An example is shown in Figure 2.7, showing a sparse point cloud which is the result of SfM and a dense point cloud created by an MVS algorithm afterwards. Additionally to the estimation of the points location, MVS also includes the estimation of point normals, which is a valuable information for surface reconstruction [Schönberger 2016b]. The core problem solved by Multi View Stereo algorithms can be described as dense pixel-wise correspondence search [Schönberger 2016b].

The method provided by [Furukawa 2010b] uses an iterative approach, which starts from a sparse point cloud and expands it. Their approach is described as a match, expand, and filter iteration. The match step detects features in the images using the Harris Corner and Edge Detector [Harris 1988] in combination with a Difference-of-Gaussians operator. These are matched across multiple images and result in a sparse set of patches associated with the detected image regions. The expansion step takes existing patches and generates new ones in nearby empty spaces to generate a more dense set of patches. Incorrect matches are removed by filtering patches based on visibility and other constraints. The expand and filter steps are applied iteratively towards a dense reconstruction of the whole scene [Furukawa 2010b].

A comparison of different MVS implementations will follow in Section 3.1.2 by evaluating existing approaches and extracting more details about their implementation.

(a) Triangle mesh structure of a surface. (b) Smooth rendering of a surface mesh.

Figure 2.8: Triangle mesh of a surface which has been created using the Poisson Surface Reconstruction method described by [Kazhdan 2013] from the point cloud from Figure 2.7.

2.3 Surface Reconstruction

The task of estimating a surface from a point cloud of an observed scene is known as surface reconstruction [Kazhdan 2013] or surface fitting [Wefelscheid 2011].

The input of a surface reconstruction algorithm may be a point cloud with estimated normals [Kazhdan 2013], but also point clouds without normals or single depth images are possible [Berger 2013]. Surface reconstruction algorithms also vary in the form of the output i.e. the representation of the surface, which may be a parametric surface, an implicit surface, or a triangulated surface mesh [Berger 2013]. Parametric surfaces and implicit surfaces are defined by an equation in Euclidean space. They are usually not suited to represent sufficiently complex 3D structures which are typically observed in the task of 3D reconstruction, so triangulated surface meshes are used in this field [Berger 2013]. Figure 2.8a shows the triangle mesh structure of a surface estimated from the point cloud of a building front, which has been obtained from a dense reconstruction. The surface can also be rendered by filling the triangles with colour or texture which results in a visualisation of the 3D representation of an object or scene as it is seen in the real world. See Figure 2.8a for an example rendering result.

A large variety of different algorithms exist in the field of surface reconstruction which results the large amount of possible applications [Berger 2013]. A class of algorithms that works on unorientated point clouds (without normals) use a subset of the input points as vertices and start interpolating a surface from that using Delaunay triangulation. One example is the surface reconstruction by Voronoi filtering described by [Amenta 1999]. These algorithms are restricted by using only points that are available in the input source. By using the existing points without interpo-

lation they suffer from the same error and noise as the input data. This results in the reconstruction surface to be very uneven. Allowing the surface reconstruction result to contain points outside of the input set allows the creation of more smooth surfaces which is achieved by interpolating between many points [Boissonnat 2000].

For surface reconstruction from point clouds that include normals a large variety of different methods have been created, which include techniques like the computation of an indicator function, locally fitting functions and moving least squares approaches [Berger 2013]. A survey of different triangulation methods can be found in [Cazals 2006]. [Berger 2013] evaluate the performance of different methods for surface reconstruction in a benchmark.

Chapter 3

Review of Processing Chains

Contents

3.1 Survey of Existing Processing Chains 20
3.1.1 SfM Processing Chains 20
3.1.2 MVS Implementations 25
3.1.3 Surface Reconstruction Algorithms 25
3.1.4 Further Processing 26
3.2 Generic SfM Processing Chain and Data Structures 26
3.2.1 Processing Steps 27
3.2.2 Data Structures 29
3.3 Software Review 32

In this chapter I will describe and analyse existing Structure from Motion (SfM) processing chains to determine how algorithms are implemented and which common parts as well as differences exist among them. This will influence the design of the framework, by focusing the analysis on the requirements and the general design of current state of the art 3D reconstruction algorithms. The goal is to extract how different steps are separated and which data structures need to be implemented for the steps to interact with each other, to be able to create a modular implementation.

This chapter also builds up on Chapter 2 by filling the theory part with description of existing implementations, adding more details to how SfM algorithms are implemented and which work additionally to the theory part needs to be done to achieve 3D reconstruction in practise. This is the basis on which the design of the framework in Chapter 4 and the implementation of SfM data structures and algorithms in Chapter 5 will build upon.

The review is split into three parts, the first part, Section 3.1, is a survey of existing processing chains. Section 3.2 describes a generic processing chain extracted from the information gathered while reviewing other implementations. It also covers the data structures that are common about these implementations leading to an overview of how the data structures provided by the processing framework should be implemented. Finally an overview about existing open and closed source implementations is given in Section 3.3.

3.1 Survey of Existing Processing Chains

Image-based 3D reconstruction is a multi-step process, so research often focuses on a specific part of the process and provides a solution or improvement of that specific part of a problem. In this survey I am also separating the processing steps into three major parts, to discuss these specific parts of the implementations separately:

- Structure from Motion (SfM) – sparse reconstruction
- Multi View Stereo (MVS) – dense reconstruction
- Surface Reconstruction – estimation of a surface mesh from a point cloud

3.1.1 SfM Processing Chains

The methods applied in SfM processing chains differ dependent on the application scenario and input data. Dependent on the image source more or less pre-processing is needed and also the number of input images plays an important role when choosing which methods to apply.

In the following I differentiate between implementations which focus more on large scale reconstruction using images from diverse sources, and implementations which are focused on an environment with homogeneous camera parameters and known environment.

The applications of the former category deal with input images that are in unknown order and may not even be taken for the purpose of 3D reconstruction. These are for example images downloaded from the internet. The methods in this category I am going to review are [Agarwal 2011], [Wu 2013], [Frahm 2010], and [Schönberger 2016a].

Implementations of the latter deal with images that are covering only one object or a pre-defined area. This is mainly the case when images where observed with the goal of the 3D reconstruction in mind, mostly using a single camera. The reviewed implementations are [Irschara 2010], [Wefelscheid 2011], and [Daftry 2015].

3.1.1.1 "City-Scale" 3D Reconstruction from Unstructured Images

In [Agarwal 2011] a system for reconstruction of a city from a large set, of up to 150K images is described. They coin the term "city-scale 3D reconstruction" using images from an unstructured source, e.g. internet platforms like Flickr[1] by searching for an image tag like "Rome" and using those images to perform a 3D reconstruction of the inner city of Rome. An example point cloud is shown in Figure 3.1a.

The characteristics of the input images are that they are taken from a very diverse set of cameras with mostly unknown camera calibration and parameters. Also the resolution and quality of the images as well as illumination conditions vary a lot among different images. These conditions require a lot of processing to filter out irrelevant images and to find correspondences between images [Agarwal 2011].

[1] Flickr, a photo sharing platform: https://www.flickr.com/

(a) 3D reconstruction by [Agarwal 2011], using 2K images. Cameras in black.

(b) Point cloud of the inner city of Rome from 21K registered out of 75K images by [Schönberger 2016a]. Camera positions in red.

Figure 3.1: 3D Reconstruction result examples from related work.

Another method is described by [Wu 2013], which while aiming at a general purpose SfM processing chain, also apply to large scale of images like the dataset used by [Agarwal 2011] and is built to deal with inhomogeneous image input. They additionally evaluate the time complexity of the SfM process aiming to develop a method that is linear in the amount of images. This is done by using an incremental approach for the SfM part and applying heuristics to limit the amount of work spent on feature matching and Bundle Adjustment.

The approach by [Frahm 2010] focuses on maximum performance, implementing a system that is able to perform a dense reconstruction on 3 million images on a single PC with a computation time of one day.

The latest state of the art SfM algorithm is implemented by [Schönberger 2016a]. They provide improved accuracy compared to the previously mentioned methods and also leverage the possibilities of GPU programming to handle large amount of images. Their incremental SfM approach uses a Next Best View selection for finding relevant images to add to the reconstruction, to make sure the quality of the output is not influenced negatively by badly chosen image pairs. An example result is shown in Figure 3.1b showing a reconstruction of the inner city of Rome using an input of 75K images.

Preprocessing Preprocessing is an important part in large scale reconstruction applications as the image source is very inhomogeneous in terms of image dimensions, camera parameters, and illumination conditions [Agarwal 2011]. The preprocessing of the input images, is performed independently per image, so it can be massively parallelized. It involves extraction of information, as far as it is available. This includes meta data such as EXIF [CIPA 2012] which may provide camera parameters like focal length or GPS coordinates. The focal length can be used as a prior in the estimation of the internal camera parameters [Agarwal 2011]. Extracted GPS data can be used after the reconstruction process to anchor the model in the real world position, but also to find nearby images when clustering images for matching [Frahm 2010]. Large images can also be down-sampled, preserving their aspect ratios and scaling the focal length to reduce computation cost [Agarwal 2011].

Feature Extraction and Description After the information extraction and normalisation, SIFT features [Lowe 2004] are extracted from the grey-scale version of each image in most approaches [Wu 2013, Agarwal 2011, Schönberger 2016a]. While using SIFT features as the default, the implementation of [Schönberger 2016a] is flexible about the usage of the point descriptors. As an alternative, binary features [Heinly 2012] may be used, which are more efficient, but less robust.

Besides using point features, when working with a large number of images, additional image features are extracted to perform clustering on the images before matching. [Frahm 2010] proposes a method which uses GIST [Oliva 2001] features to describe the general appearance of the image, which is used to cluster images before matching.

Image Matching Given the diversity and also the amount of images used in these approaches multiple steps need to be performed to get sufficient results for image correspondence. In [Agarwal 2011] a combined approach using vocabulary tree search [Nister 2006] for searching match candidates and approximate nearest neighbour search for feature matching is used. A matching point pair, that passes Lowe's ratio test [Lowe 2004] is accepted.

Before the actual matching, [Frahm 2010] uses GIST [Oliva 2001] features to cluster images to obtain a set of iconic views and leverages GPU computing for massive parallel matching of the image features. Image matching is then performed inside of these clusters. Afterwards a representative image is selected for each cluster and nearby clusters are detected using a vocabulary tree search on these images. They also use geo-location information if available for a cluster to improve these matchings. The search for candidates before matching is necessary because for a number of images in the scale of 100.000 a pairwise matching, while computationally feasible, would be a huge waste of computation power, because most images do not match [Agarwal 2011]. [Wu 2013] describes a preemptive matching algorithm that reduces the amount of comparison operations for each image pair by sorting the feature vectors and skip an image pair early if the first few features do not match. Their approach aims to achieve SfM in linear time on the number of images. Using this approach, they are able to reduce the amount of computation significantly, as 75% to 98% of the images, in the large data sets they used, did not match and could be rejected in an early stage.

The result of the matching is a graph with images as the nodes and edges between corresponding images pairs. It is called *match graph* by [Agarwal 2011] and *scene graph* by [Wu 2013]. I will use the term *image graph* in the following, because in a general case, the graph contains images and the relation between them.

The image graph may be adjusted after it has been created, like in [Agarwal 2011], where they perform further steps on the image graph i.e. add further matching pairs after an initial instance of the image graph. Also in large data sets the graph typically consists of connected components which can be separated for computation in parallel [Agarwal 2011].

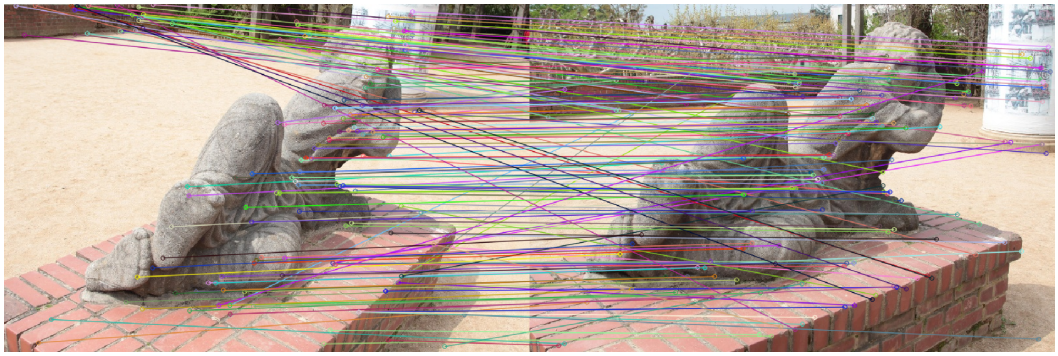


Figure 3.2: An example of outlier matches on the "Der Hass" dataset by [Fuhrmann 2014]. Despite other outliers, it can be seen that some points of the wall in the background are matched against the stand of the statue in the foreground.

Filtering of the Image Graph and Geometric Verification Automatically estimated corresponding point pairs are not guaranteed to belong to the same points in an image and especially in large size dataset the outlier rate is expected to be very high [Wu 2013]. Therefor after matching, or during the matching process, validation of the matched key points and geometric verification is performed to filter out wrong matches. Figure 3.2 shows an example of a point matching which contains a lot of outliers.

A commonly used technique for outlier removal is RANSAC [Fischler 1981], which has been explained in Section 2.1.3.5. This approach is used by [Agarwal 2011] to verify the results of the approximate nearest neighbour search. The estimated Fundamental Matrix calculated in this step may later be reused when estimating the 3D geometry [Agarwal 2011, Schönberger 2016a]. In [Frahm 2010] RANSAC is applied within each cluster of the image graph before connecting related clusters with each other.

[Schönberger 2016a] use an additional approach for geometric verification of an image pair. They try to find a homography for mapping points from one image to another, which can be based on the Fundamental Matrix. If a valid transformation can be found for a certain amount of points, the verification succeeds, otherwise an image pair is discarded.

3D Geometry Estimation and Bundle Adjustment After determining the image graph, an algorithm for 3D geometry estimation is applied on each connected component of the graph [Agarwal 2011, Wu 2013].

[Schönberger 2016a], [Frahm 2010] and [Wu 2013] use an incremental reconstruction approach starting with an initial image pair to determine the Epipolar Geometry and 3D coordinates of corresponding points. The reconstruction is then completed by incrementally adding more and more images. One approach of finding an optimal initial image pair is described in detail by [Beder 2006].

In [Schönberger 2016a] a Next Best View algorithm is used to find new images to add to the reconstruction. This algorithm selects images, which are supposed to add the best value to the current reconstruction state. New images are used to triangulate additional points, but also to verify and improve the existing solution, if the image adds information form a different view.

The approach of [Wu 2013] is an incremental approach aiming for a linear complexity of the algorithm. SfM as well as Bundle Adjustment [Triggs 1999] are applied incrementally, by adding more and more images. They are however not run in full, [Wu 2013] has measures to decide to add more images and skip Bundle Adjustment for a few iterations as well as applying it to a part of the scene. This makes their approach very efficient.

The result of this step is an estimation of the external and internal camera parameters in form of the projection matrix $\mathbf{P}$ for each image, as well as the Fundamental Matrix $\mathbf{F}$ for each image pair (cmp. Section 2.1.3.4). The resulting 3D points which can be calculated from the matched key points for each image pair result in a sparse 3D reconstruction [Wu 2013, Agarwal 2011] (cmp. Section 2.1.3.6).

The estimation of 3D geometry is often directly combined with Bundle Adjustment, which is applied on intermediate results [Frahm 2010, Wu 2013]. Bundle Adjustment does not change the existing data structure, but improves the existing result by minimising the overall estimation error [Agarwal 2011]. The idea behind Bundle Adjustment is explained in Section 2.1.3.7

3.1.1.2 SfM Using a Single Camera on a Known Scene

A processing chain described in [Wefelscheid 2011], and similar methods by [Irschara 2010] and [Daftry 2015], aim for the reconstruction of buildings by using images obtained from a camera mounted on an unmanned aerial vehicle (UAV).

In contrast to the methods described in the previous section, these methods operate in an environment where the images where taken explicitly with the goal of 3D reconstruction. The images in these approaches are taken from a single camera which may be calibrated beforehand so most of the image parameters are known. The known information can be completed even more by using the positioning information of the UAV to improve feature matching and the creation of the image graph as shown by [Stathopoulou 2015]. Using this approach the SfM algorithm can be initialised with prior information which is not far from the real values so a very accurate reconstruction can be performed [Wefelscheid 2011, Daftry 2015].

In [Wefelscheid 2011] key point detection is implemented using the Förstner Operator (cmp. Section 2.1.1), which is better suited for human made structure like buildings. The description of the key points is the same as in other approaches described before, using the SIFT descriptor.

A Loop closure step is used to detect images that capture the same part of a scene. This is done by computing a new descriptor per image based on the images SIFT features and comparing these features to find similar images.

After loop closure an image graph is computed. This works on image triplets

instead of image pairs. From these the Trifocal Tensor (cmp. Section 2.1.3.3) is computed which contains the Epipolar Geometry information between all images in the triplet. Similar to the Fundamental Matrices this provides the projection matrix for the images, which are used to compute 3D points for the corresponding point pairs. Their approach also includes Bundle Adjustment to minimise the error of estimated geometry in all images in the image graph [Wefelscheid 2011].

A very similar approach is described by [Irschara 2010] also using SIFT features, two matching steps, a graph and then SfM on image pairs. However they use image pairs instead of triplets.

3.1.2 MVS Implementations

After the estimation of the 3D geometry of a model, which includes the sparse reconstruction, the next step is to get a more detailed model of the scene covering a dense point cloud. Algorithms in this field are called Multi-View-Stereo algorithms, because they use multiple images for estimation of a dense point cloud. Multiple images are used, as it has been shown by [Rumpler 2011], that redundancy benefits the accuracy of the reconstruction.

Most MVS algorithms operate on the same input, which is a list of images for which the projection matrix $\mathbf{P}$ has been estimated [Furukawa 2010b, Schönberger 2016b]. [Frahm 2010] however goes directly from the image matching to the dense reconstruction without doing sparse reconstruction of the scene. For an accurate reconstruction images need to be normalised in terms of image resolution to avoid over or under-sampling [Schönberger 2016b]. Also if a distortion model has been estimated for an image, the images should be normalised by undistorting it before applying the MVS algorithm [Furukawa 2010b, Schönberger 2016b].

The methods described in [Furukawa 2010a] provide an algorithm that wraps around existing MVS implementations to allow them to operate on very large datasets. It includes a view clustering algorithm that can separate the input data set into multiple clusters and merge the resulting point clouds from MVS algorithms applied to these clusters.

For all reviewed implementations the output is a dense point cloud in 3D space which also includes normal information, which can be used for surface reconstruction [Furukawa 2010b, Frahm 2010, Schönberger 2016b].

3.1.3 Surface Reconstruction Algorithms

As already explained in Section 2.3 a large variety of surface reconstruction algorithms exist, which can be applied in different applications with different input and output data. For my thesis I am limiting the scope of description to one algorithm which is widely used in 3D reconstruction applications, which is the Poisson Surface Reconstruction described by [Kazhdan 2013]. To generate a surface mesh from the estimated dense point cloud, this approach is referenced in many of the processing chains reviews before [Schönberger 2016b, Furukawa 2010a, Furukawa 2010b].

3.1.4 Further Processing

The estimation of a 3D mesh is not necessarily the end of a processing chain. There are more steps that could be added before, after or in between the processing chain and a framework should be designed to allow this.

An example of processing steps to apply before 3D reconstruction is the estimation of optimal camera positions given a rough shape of the object that should be reconstructed. Such an algorithm has been described and implemented by [Brandt 2016], using a multi-step processing chain.

For post-processing after 3D reconstruction a variety of use cases exist. There is for example the possibility of aligning the estimated 3D model in the real world and combine it with a map as it has been done using geo reference information from image meta data in [Frahm 2010].

The alignment of multiple independently created meshes for indoor and outdoor scenes of a building are put together in [Cohen 2016] by detecting window positions and searching for similar window alignment in related models.

After a 3D mesh has been estimated certain operations can be performed to refine the solution. [Jawer 2015] describes an approach for finding erroneous parts in automatically reconstructed 3D meshes. This information can be used for further methods for mesh repairing which are described by [Attene 2010]. Their methods provide ways of ensuring certain properties of the mesh structure which are necessary for further mesh processing, e.g. the order of points around triangles or normal directions for determining inside and outside.

Another post-processing method for noise removement from triangle meshes is described by [Fabio 2003].

In the field of image-based 3D reconstruction it is also desirable to apply the texture, which is available from the images to the surface mesh. One approach, which is described by [Stathopoulou 2011] covers the task of projecting images onto a surface mesh. They describe the texturing of high resolution meshes that contains a lot of details and fine grained structure as well as low resolution models.

Another example of further processing is the evaluation of the estimated point clouds and mesh. An example is SyB3R, which is a benchmark for image-based 3D reconstruction algorithms described in [Ley 2016]. Their processing chain is designed to produce ground truth data by rendering images from 3D models, which are then fed into a SfM and MVS pipeline. The results can then be compared with the ground truth to evaluate the quality of the reconstruction.

3.2 Generic SfM Processing Chain and Data Structures

Concluding from the previous review of 3D reconstruction implementations, the common steps and intermediate data structures that are the same among most implementations can be extracted.

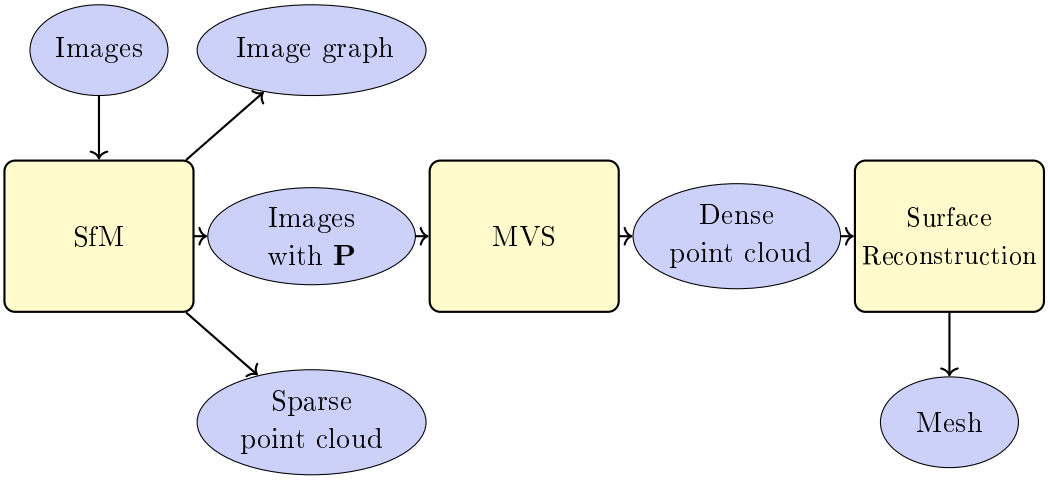


Figure 3.3: A high level overview of a generic 3D reconstruction processing chain. Images are the input of the SfM step, which produces an image graph, a sparse point cloud and the external and internal camera parameters encoded in the projection matrix **P**. These are then used by the MVS step to produce a dense point cloud. Surface reconstruction takes the dense point cloud, which includes point normals to estimate a surface mesh.

3.2.1 Processing Steps

The very high level of a 3D reconstruction processing chain consists of the following steps, which match the separation which has already been applied in the previous sections: SfM → MVS → Surface Reconstruction. The relevant data structures can be extracted, which are images, an image graph consisting of image pairs, a sparse and dense point cloud, and a surface mesh. The processing steps and their relation as well as involved data structures are illustrated in Figure 3.3.

The SfM part can be broken down into further steps, which are: Preprocessing → Feature Extraction (Keypoint detection and description) → Feature Matching → Match Filtering → Geometry Estimation → Geometry Refinement. In most of the reviewed implementations, many of these steps are combined in one step, but could as well be covered by independent implementations. For example key point detection and description are mostly covered by SIFT in one step, but are separated in [Wefelscheid 2011] by using a different point detector. Also Feature Matching and Match Filtering, as well as Geometry Estimation and Geometry Refinement (Bundle Adjustment) may overlap or be merged into one step. The main data structures needed to represent the communication between these steps are images, key points, image pairs, an image graph, and a point cloud. Figure 3.4 shows a detailed relation of steps and data structures used in the SfM processing chain.

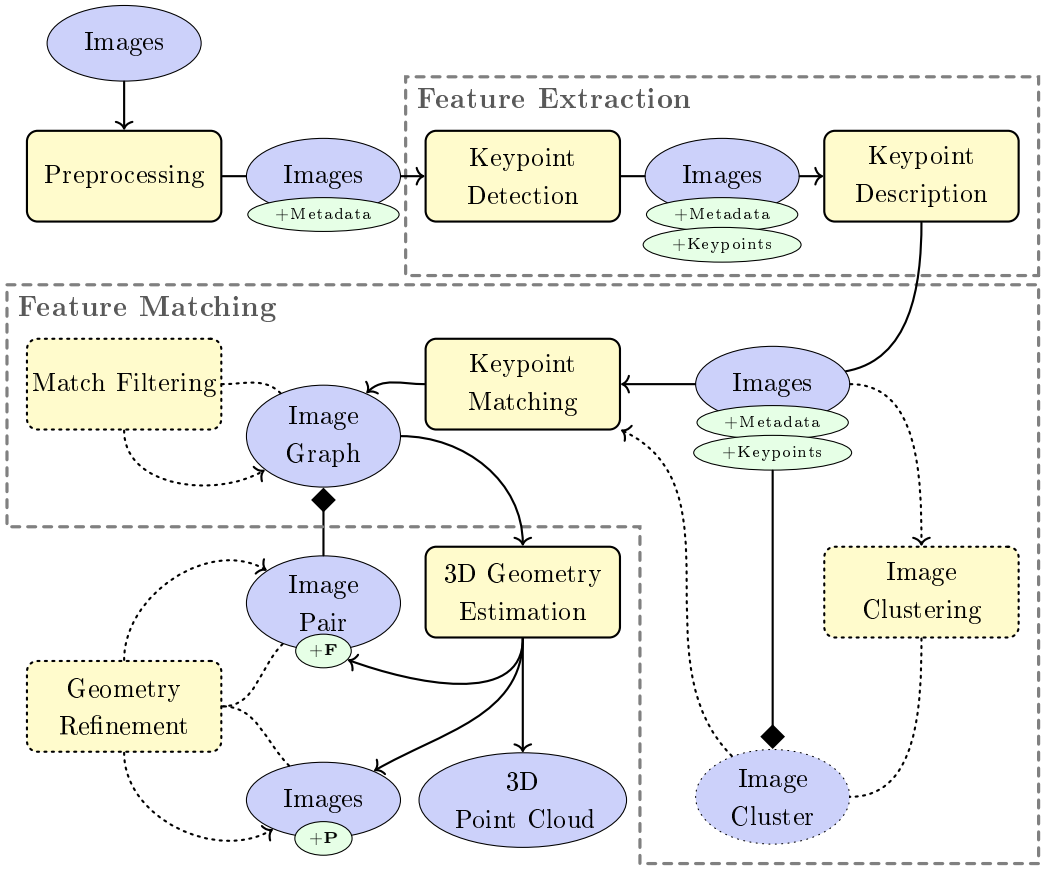


Figure 3.4: A detailed overview of the relation of processing steps and data structures in the SfM processing chain. Processing steps are in boxes, data structures in ellipses, optional steps are dashed.
The input of the processing chain are images. Preprocessing extracts meta data and attaches them to the images. The Feature Extraction part includes key point detection and description which attaches key point information to the images. The next part is Feature Matching, which optionally includes image clustering. After that the key points are matched, which results in an image graph which consists of image pairs. The matches may optionally be filtered, before performing 3D Geometry Estimation, which results in the Fundamental Matrix to be added to all image pairs, as well as a set of images with 3D geometry information in form of the projection matrix $\mathbf{P}$ and a sparse 3D point cloud. The estimated geometry may be refined in a Geometry Refinement step e.g. by using Bundle Adjustment.

3.2.2 Data Structures

In each of the processing steps of a SfM processing chain, different data structures are produced, which are given to the next step to work on them. For example after Feature Matching a set of matching point pairs between two images is returned, the 3D geometry estimation defines the Epipolar Geometry for each image pair and the result of the Dense Reconstruction is a point cloud in 3D space. An important part of the framework is the definition of these data types to ensure interoperability between different algorithms. In this section I define the data types by extracting the information from the previous review of processing chains.

Besides the properties listed below, all data types in the framework should be able to store additional meta data in form of key-value pairs as well as custom data attached, to allow implementing methods that have not been reviewed here.

3.2.2.1 Images

The main input source of each of the algorithms are images. Digital images can be stored in many different formats, so the framework should not be limited by the storage format of the image used. A library like OpenCV [Bradski 2000] provides methods for reading common image formats.

The image data structure should store a reference to the image file it represents. It should not directly load the raw image data when it is created, to save memory and allow the usage of a large amount of images, as it has been shown in the reviewed processing chains in Section 3.1. Methods for reading the image data are provided to be used when it is needed. Besides that the image data structure needs to be able to store meta data information such as EXIF [CIPA 2012] which is used by [Agarwal 2011, Frahm 2010].

Additionally also features like key points and their descriptors are stored for each image. This should be optimised for SIFT [Lowe 2004], because as explained above that is the approach used by most of the implementations. Besides SIFT features, the data structure should allow storing other descriptor types. [Frahm 2010] uses GIST features additional to SIFT for clustering images, so an option to add additional data references to images can be useful. Also [Wefelscheid 2011] computes an additional descriptor for describing each image.

After the SfM step an image should store the estimated camera parameters $\mathbf{K}$, $\mathbf{R}$, and $\mathbf{t}$, as well as the projection matrix $\mathbf{P}$.

The image clustering before matching depends a lot on the matching process, so I decided not to introduce a generic data structure for the image clusters but only implement the input of the matching process (list of images) and the output (image graph). For implementing image clustering no extra data structure is needed as a cluster would be just a list of image lists or can be implemented as a custom data structure which will be described in Section 4.4.2.

The following table shows a list of properties to be implemented for the image data structure. Optional means that creating a data structure does not require to add this information, it may be added in future processing.

Image			
Property	Optional	Type	Description
width	no	Integer	Image width in px.
height	no	Integer	Image height in px.
keypoints	yes	KeypointList	List of key points.
P	yes	Matrix 3×4	Projection matrix.
camera parameters	yes	CameraParameters	External and internal camera parameters.

CameraParameters			
Property	Optional	Type	Description
R	no	Matrix 3×3	Camera orientation.
t	no	Vector 3	Camera position.
K	no	Matrix 3×3	Internal calibration.
f	no	Float	Focal length.
ccd_width	no	Float	Resolution in $\frac{px}{mm}$.

KeyPointList			
Property	Optional	Type	Description
keypoints	no	List	List of key points.
descriptors	yes	List	List of key point descriptors.

3.2.2.2 Image Pair and Image Triplet

A pair or triplet of images can have common information, so it should have its own data structure. For an image pair the Epipolar Geometry in form of a Fundamental Matrix **F**, as well as the key point matches should be stored. For the image triplet the geometry is described by the Trifocal Tensor **T**. An image pair or triplet also contains references to the images it refers to. The following table describes the properties of the ImagePair and ImageTriplet data structure.

ImagePair / ImageTriplet			
Property	Optional	Type	Description
imageA	no	Image reference	First image.
imageB	no	Image reference	Second image.
imageC	no	Image reference	Third image. Only for ImageTriplet.
keypointMatches	yes	List	List of integer pairs referencing the key points that match.
F	yes	Matrix 3×3	The Fundamental Matrix. Only for ImagePair.
T	yes	Tensor	The Trifocal Tensor. Only for ImageTriplet.

3.2.2.3 Image Graph

The image graph is a list of image pairs or image triplets. It should allow to access the image pairs and the list of individual images separately to allow working with the pairs, as well as iterating over single images dependent on what is needed for an algorithm input.

An image graph may optionally store information about points seen in multiple images when a matching algorithm does not produce point pairs, but clusters of points seen in more than two images [Gil 2010]. For these matching key points found in multiple images, the term *track* is introduced by [Snavely 2006].

ImageGraph			
Property	Optional	Type	Description
imagePairs	no	List	List of ImagePair references. May be empty.
imageTriplets	no	List	List of ImageTriplet references. May be empty.
images	no	List	List of Image references.
tracks	yes	List	List of point references seen in multiple images.

3.2.2.4 3D Point Cloud (Sparse and Dense)

A 3D point cloud is a list of points represented by their 3D coordinates $[X, Y, Z]^T$. For each point additional information may be stored, if available. These can be point normals, which indicate a direction of the surface the point is on, which is typically estimated by an MVS algorithm [Schönberger 2016b, Furukawa 2010b]. Also colour information in RGB format can be stored, which is usually taken from the images' key points to allow a better visualisation of the point cloud.

PointCloud			
Property	Optional	Type	Description
coordinates	no	List	List point coordinates $[X, Y, Z]^T$.
normals	yes	List	List of point normal vectors.
colours	yes	List	List of point RGB colour information.

3.2.2.5 3D Mesh

A 3D mesh in the form that is used in 3D reconstruction is typically represented as a triangle mesh [Berger 2013]. A triangle mesh consists of vertices and a set of triangles that connect them to form the surface. The digital representation depends on the used library, so the framework would only provide a container class that has one property referencing the triangle mesh data structure.

3.3 Software Review

While the main goal of my thesis is the development of a framework, for this framework to be of any use, some algorithms need to be implemented using it. In the following I will review existing algorithm implementations, which can be used to create modules for the framework. The implementation of most algorithms from the reconstruction processing chain is not trivial, so the reuse of existing code is necessary to provide useful functionality.

Bundler is a SfM software written by Noah Snavely. It offers an implementation of the SfM pipeline for feature matching and 3D geometry estimation including Bundle Adjustment. The algorithms are described in [Snavely 2006], which covers the initial release of the software and [Agarwal 2011], in which the software has been extended to work in city scale applications. It is written in C and provides separate binaries for the matching and 3D geometry estimation step, which can be used to integrate them into the framework.

Language:	C
License:	Free Software (GNU General Public License)
URL:	`http://www.cs.cornell.edu/~snavely/bundler/`
	`https://github.com/snavely/bundler_sfm`

Visual SfM is a software created by Changchang Wu, which provides a graphical user interface for the whole SfM pipeline from key point extraction and matching over SfM to MVS. The SfM algorithms used in the implementation are described in [Wu 2013]. For MVS the methods and software by [Furukawa 2010a] (CMVS) and [Furukawa 2010b] (PMVS2) are integrated. The GUI software is closed source, however parts of it have been released as Open Source, which are SiftGPU and Multicore Bundle Adjustment, which are described below.

License:	Closed Source, free for personal or academic use.
URL:	`http://ccwu.me/vsfm/`

SiftGPU is an implementation of SIFT [Lowe 2004], which leverages GPU for computation of SIFT features and also provides an implementation for feature matching. It is written by Changchang Wu as part of Visual SfM [Wu 2013].

Language:	C++
License:	Open Source, custom license, free for personal or academic use.
URL:	`http://cs.unc.edu/~ccwu/siftgpu/`

Multicore Bundle Adjustment also known as Parallel Bundle Adjustment (PBA) is an implementation of Bundle Adjustment that leverages CPU and GPU parallelism and is part of Visual SfM by Changchang Wu. The algorithm and implementation are described in [Wu 2011].

Language:	C++
License:	Free Software (GNU General Public License)
URL:	http://grail.cs.washington.edu/projects/mcba/

PMVS2 is a MVS implementation developed by Yasutaka Furukawa and Jean Ponce. The algorithm is described in [Furukawa 2010b] and was mentioned among the reviewed implementations in Section 3.1.2.

Language:	C++
License:	Free Software (GNU General Public License)
URL:	http://www.di.ens.fr/pmvs/

CMVS is a system for internet scale MVS built on top of PMVS2 and was written by Yasutaka Furukawa. It takes the output of a SfM algorithm and decomposes it into clusters, which are then separately given to an MVS algorithm. The results are merged back into a complete model after computation. The algorithm is described in [Furukawa 2010a], which is part of the overview in Section 3.1.2.

Language:	C++
License:	Free Software (GNU General Public License)
URL:	http://www.di.ens.fr/cmvs/

COLMAP is a complete 3D reconstruction pipeline implementation written by Johannes Schönberger. It is an Open Source implementation providing separated executables for all processing steps from key point detection and matching over SfM to MVS and surface reconstruction. The software heavily relies on GPU computation for maximum performance, but is at the time of this writing limited to Nvidia GPUs, which makes parts of it unusable on other systems. The implemented algorithms are described in [Schönberger 2016a] and [Schönberger 2016b] and have been summarised in Section 3.1.

Language:	C++
License:	Free Software (GNU General Public License)
URL:	https://colmap.github.io/ https://github.com/colmap/colmap

OpenCV is a generic computer vision library, which also provides support for SfM [Bradski 2000]. It provides implementations of key point detection and description algorithms like SIFT [Lowe 2004], SURF [Bay 2008], Harris [Harris 1988], and many others. It also provides data structures and methods that can be used to implement key point matching and estimation of 3D geometry. Since version 3.1 it also includes a complete implementation of a simple SfM pipeline.

Language:	C, C++, Python
License:	Open Source (3-clause BSD License)
URL:	http://opencv.org/
	http://docs.opencv.org/3.1.0/d8/d8c/group__sfm.html

SyB3R is an Open Source 3D reconstruction evaluation framework implemented by Andreas Ley. The framework generates ground truth data as well as images from a 3D scene which can be put into a 3D reconstruction tool chain to compare the result with the ground truth for evaluation [Ley 2016].

Language:	C++
License:	Free Software (GNU General Public License)
URL:	http://andreas-ley.com/projects/SyB3R/

PoissonRecon is an Open Source surface reconstruction implementation by Michael Kazhdan. It implements the Poisson Surface Reconstruction methods described in [Kazhdan 2013].

Language:	C++
License:	Open Source (MIT License)
URL:	https://github.com/mkazhdan/PoissonRecon

Chapter 4

Design and Implementation of the Framework

Contents

4.1 Data Model 36
4.1.1 Module 36
4.1.2 Input and Output Data 37
4.1.3 Processing Chain 38
4.1.4 Summary 39
4.2 Execution of a Processing Chain 39
4.2.1 Module Runner 40
4.2.2 Data Manager 41
4.2.3 Mapping on List Data Types 42
4.3 User Interfaces and Configuration 42
4.3.1 The Processing Chain Configuration File 42
4.3.2 Graphical User Interface 44
4.3.3 Command Line Interface 49
4.4 Extending the Framework 50
4.4.1 Implementing a Module 50
4.4.2 Implementing a Data Type 52

In this chapter I describe the concepts of the framework that is developed for the creation of SfM processing chains. The implementation of the framework is based on a generic framework for image processing which has been developed at the Computer Vision and Remote Sensing department at TU Berlin. For my thesis, I have revised the current implementation and adjusted the data model and work flow design to be more flexible and extensible to fit the needs for implementing the SfM pipelines on top of it. This resulted in a rewrite of the underlying library.

I will start the description of the framework by defining the data model for representing processing chains in Section 4.1. After that, the components for executing a processing chain and handling the data are described in Section 4.2.

The framework provides two interfaces for working with processing chains, which is a Graphical User Interface (GUI) and a Command Line Interface (CLI). The GUI is used for creating and running a processing chain and also allows visualisation

of intermediate and final processing results. The CLI is used to run processing chains. Processing chains may be developed by hand as the file format used is human readable and writable. User interfaces as well as the configuration file are explained in Section 4.3. Finally Section 4.4 explains how to implement data structures and algorithms for the framework.

4.1 Data Model

The goal of the framework is to provide methods for creating implementations of algorithms that can be easily reused in different scenarios and also be configured using parameters at runtime. It should be possible to implement an algorithm and load it in different processing scenarios as a plugin, without the need to recompile the code in a different application.

To achieve this, the processing framework should define an interface that is common among all implemented plugins so the execution of an algorithm as well as input and output data can be handled in a standardised way.

4.1.1 Module

The first thing to define for this purpose is that algorithms are implemented as modules in form of a shared library, which can be dynamically loaded at runtime. A *module* is a self-contained implementation of an algorithm, that communicates with the framework through *inputs* and *outputs*, while its behaviour can be controlled using *parameters*. Input data and parameters are provided to the module before the execution. The output data is returned by the module as soon as it is completed and then retrieved by the framework for delivery to other modules or visualisation. The concept is visualised in Figure 4.1.

Besides output data, a module may also produce logging output to provide details about the processing. It may also export progress information to give an indication on how much of the processing work is finished to give the user feedback on when to expect the processing to be done.

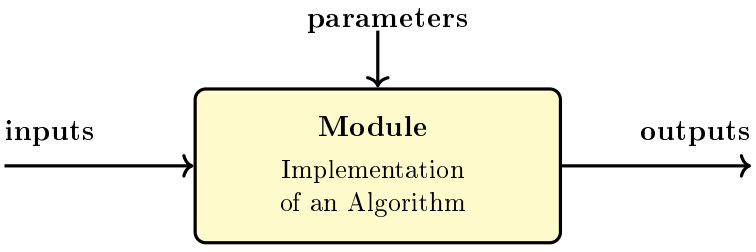


Figure 4.1: A module is an implementation of an algorithm, which converts input data into output data. The algorithm behaviour can be controlled using parameters.

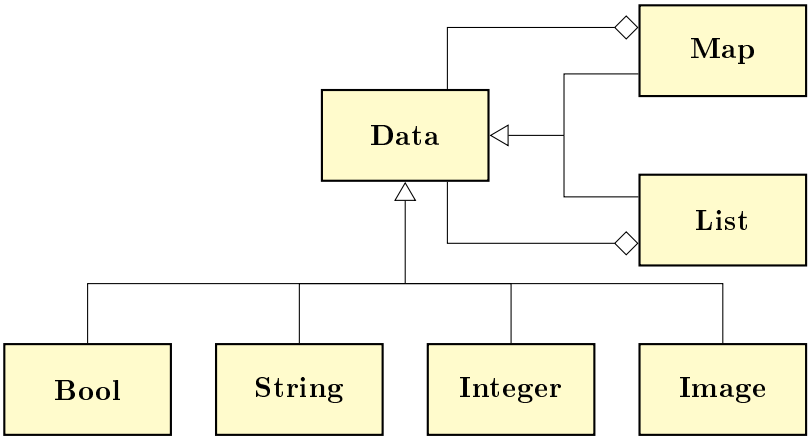


Figure 4.2: UML class diagram of the basic data types provided by the framework. Bool, String, Integer, Image, List, and Map extend from the Data class. Map and List are aggregations of Data.

4.1.2 Input and Output Data

For the framework to be able to handle the input and output data of modules in a meaningful way, the data structures also need to implement an interface defined by the framework. A data implementation is usually a wrapper around the data structures used internally, e.g. the ones provided by the C++ standard library or other data structures provided by other libraries.

The common interface of a Data structure is the `Data` class, from which all other types need to extend. Module implementations may use the data structures that are provided by the implementation or use their own data types by extending from the Data class.

The framework comes with implementations of basic data types like String, Integer, Boolean. These wrap the data types provided by the C++ language and standard library. It also comes with an Image type based on the OpenCV [Bradski 2000] Matrix data type, which is a flexible representation of an image. Additionally two collection types, List and Map are provided. List is based on the standard library vector data structure and is a simple collection of other data types. Map is based on the standard library map and provides a way to store data by indexing each item with a string key. A UML class diagram of the implemented data types is shown in Figure 4.2.

A data type may implement additional functionality to provide serialisation and visualisation. Serialisation can be used to pause a processing chain and store the data on disk to be able to close the program and reopen it to resume processing later. Serialisation can also be used to cache intermediate results to avoid re-computation of the same thing over and over again when testing changes on another module. The implementation of serialisation is optional, so if not implemented on a data type, the related functionality will not be available.

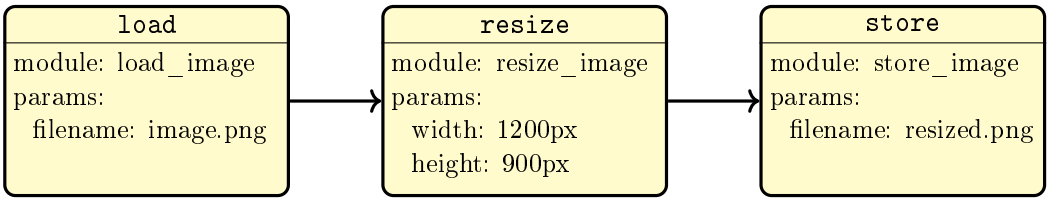


Figure 4.3: An example processing chain showing the implementation of the process of resizing an image. The processing involves three steps: (1) `load`, which reads the image from a file, (2) `resize`, which performs the desired operation based on given parameters, and (3) `store`, which writes the result back into a file.

For inspecting intermediate processing data and also for visualising processing results, data types can implement visualisation functions that can use an API provided by the framework to show windows with images, graphs, or store annotated images. This feature is described in more detail in Section 4.3.2.3.

4.1.3 Processing Chain

In order to execute one or more modules, the framework provides a data structure called the *processing chain* which consists of *processing steps*. A processing step is identified by its name, which is unique inside of a processing chain.

Each processing step refers to exactly one module which defines the algorithm used in this step. It also defines the parameters used for executing the module. For example, a processing step named `to-grey-scale` could refer to a `convert-image` module with a parameter `channels=greyscale`.

The inputs of the processing step's module are defined via dependencies to other steps' outputs. So the data that is provided as the output of a previous step is used as the input of the current step. By defining these dependencies, a processing chain is formed as a dependency graph between processing steps. This graph also defines the order in which the modules have to be executed when running the processing chain. A simple processing chain example is shown in Figure 4.3 showing three modules, `load`, `resize` and `store`, to implement the process of resizing an image. It reads the image from a file, performs the resize operation and stores the result back into another file. The dependencies of the `resize` step on the `load` step define the input of `resize` to be populated with the result of `load`. The same applies to `store`, which takes the image from the `resize` step.

A module can have multiple inputs as well as multiple outputs, which are identified by a name. These are defined by the module implementation. For example a processing module which performs an averaging operation on an image could have two inputs: (1) the image, and (2) a kernel, which defines the averaging window. To apply Gaussian blur on the image, a processing chain that provides a Gaussian kernel can be created to provide the second input. See Figure 4.4 for an example.

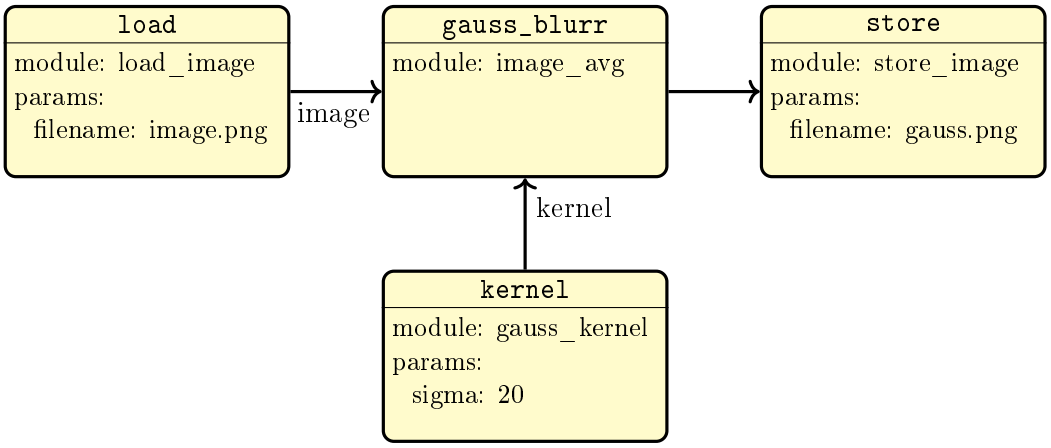


Figure 4.4: An example of a processing chain with non-linear flow. The `image_avg` module has two inputs. One is the *image* and the other is the *kernel* to apply for averaging. The processing chain defines which outputs of other modules should be passed to these inputs when the processing chain is executed. The data is retrieved after the processing steps `load` and `generate_kernel` have been executed. This example also shows how a generic reusable module implementation, like image averaging, is used to generate a concrete application using Gauss. The same module can be used in various different applications when other kernels are applied.

4.1.4 Summary

Figure 4.5 shows an UML class diagram visualising the relations between the entities of the data model. This model provides several benefits. It allows separation of the modules code via a well defined interface, which allows reusing the implemented algorithm in different scenarios without the need to adjust the code. It also allows a flexible processing execution which is controlled by the framework dependent on how modules are connected instead of hardcoded in a program. This way parallelism, when possible, can be exploited automatically by the framework, without the need for implementing parallelism in the algorithm itself.

4.2 Execution of a Processing Chain

A typical processing chain consists of steps that use modules from different sources. Some modules are provided by the framework, which are simple things like loading and storing files as well as basic image operations, or the SfM modules described in Chapter 5. Dependent on the use case the other modules are either written by the user or taken from third parties which provide certain functionality that is to be used in the processing.

A common approach for including external algorithms, which is also used for most of the SfM implementations in Chapter 5, is the wrapping of an executable

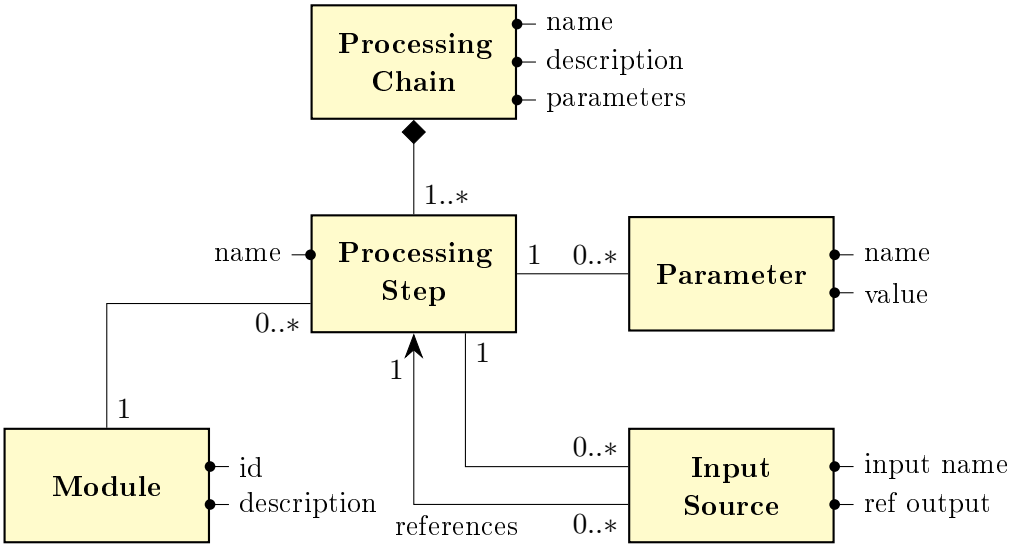


Figure 4.5: UML class diagram of the entities in a processing chain definition. The processing chain consists of processing steps. Each processing step references a module which defines the algorithm to execute. To determine the input of the module, the processing step has input sources which reference other processing steps to read their outputs. Additionally a processing step is assigned parameters which are passed to the module upon execution.

or script, by preparing the script input in an expected format or file, running the external program, and reading the processing result, converting it to data types used in the framework.

When developing new methods, the implementation would normally be written in C++ and is provided directly by the module or a separate library linked to it. Bindings to other programming languages or tools like Python or MatLab could also be implemented, but are not part of the current implementation. For using algorithms from other languages than C++, an approach like described before, by running an external program, can be used.

4.2.1 Module Runner

As already mentioned, a benefit of the processing chain description is that it only defines dependencies between the steps in a descriptive way. It does not contain any specifics about the execution of modules, so the framework can determine an execution process that is most efficient by exploiting parallelization to run independent steps. Separation of steps could also be used to implement methods, which allow distributing computation on different computers. The framework would then send the serialised input data to the execution server, run the module there and transfer the result data back, after execution has finished. The data type must implement the serialisation functions for this to work. This feature is not implemented yet.

The *Module Runner* provides the core functionality for running a processing chain. For running the processing chain, the steps are first ordered by their dependencies to ensure, that input data is available from modules, that have produced this data before. Circular dependencies are not possible and will result in an error.

A processing chain is also validated for matching input and output data types. A type-checking mechanism is applied to the processing chain before running to ensure the data passed from one module to another matches the expected type. This is done before execution of the processing chain by using meta data provided by the module, which for each input and output declares the data type it expects. Only when the data types match for all declared dependencies, the processing chain is executed. This way the module implementations can safely use type casting on the input data because the framework ensures that the data is provided in the expected type.

The Module Runner has two operation modes. The first mode is running a processing chain as a whole from start to end. The second mode is a step-by-step run, where the user has to start the next step manually when the current is finished. This allows debugging a processing chain by running a step, then inspecting the output and run the next step afterwards. A possible extension of the framework could also be to allow the user to modify data between steps, either to provide user input or to correct the result of automatic processing by manual verification. Switching between both modes is also possible, which in the first mode will cause a running chain to pause after the current step has finished and in the second mode to run further steps without stopping until the end of the processing chain has been reached.

4.2.2 Data Manager

The data transferred between the modules is managed by an entity called *Data Manager*. It consumes the output data of each module when the processing is finished and ensures, that the data is passed to other modules requiring it as input.

In normal operation mode the data is kept only until no other module, that is requesting it, needs to be run. To keep memory clean this data is then dropped. It is possible to enable a debug mode where all data is kept for the whole runtime of the processing chain so it can be inspected later.

For handling data of large size, that should not be kept in memory, the data can be stored on the disk and the data type will only store a reference to the data which can be used to access it. This works when the data type implements the serialisation methods.

For repeated runs of the processing chain the caching of specific data can be enabled to skip time consuming recalculation of some data when changing functionality or parameters of another module for testing. The data type must implement serialisation for this to work.

4.2.3 Mapping on List Data Types

A special feature for list data types exists that allows running a processing module for each item of the list. All modules that have a single input can be used in this way. The output of the processing step that uses this feature will then contain a list of all output elements of the module instead of one output element.

If the output of a previous processing step is a list, it can be connected with a module that accepts a single item by specifying the input appending `.map()`. For example if the output of a `load` processing step is a list of images the input should be specified as `image: load.images.map()`. The framework will then run the module for each item in the list and combine all output data into lists as well. This is useful to write modules that can work in parallel, so the framework can use this to distribute the work to different CPU cores. It also simplifies the algorithm implementation as the user can focus on the algorithms implementation without the need to care about the iteration over the list.

4.3 User Interfaces and Configuration

The framework provides two different interfaces for working with it. The Graphical User Interface (GUI) and Command Line Interface (CLI), which both share the core framework implementation which is contained in a shared library. The library provides all the core functionality, which is the interface for data structures and modules, the Module Loader, Runner, and Data Manger, as well as logging functionality. The configuration of a processing chain is stored in a file format which is the same for both interfaces so processing chains created with the GUI can be run with the CLI and vice versa. The general program layout is shown in Figure 4.6.

4.3.1 The Processing Chain Configuration File

To be able to store a created processing chain in a file for later use a file format needs to be defined. The selection of the file format as well as the format itself are explained in the following sections.

4.3.1.1 Selection of the Storage Format

From a programming point of view, the easiest solution for storing a processing chain would be to dump the binary content of the data structures form C++ into a file and read it back when needed. This approach however has several drawbacks.

The major problem with this is that it is not human readable so it is impossible to manually edit the file without the GUI, which would prevent several use cases for using the framework in a console environment only, for example via remote connection to a server. Binary formats are also harder to maintain when it comes to creating different versions of a processing chain. Differences between text files can be made visible using existing text diff tools. The possibility of being able to manually edit a file also allows the usage of comments to explain the layout of the

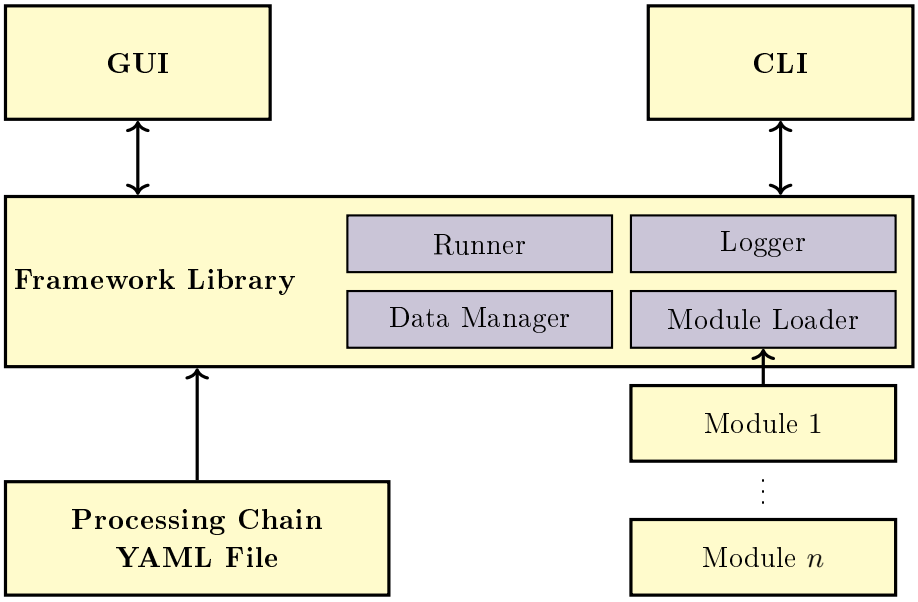


Figure 4.6: Organisation of the framework components. The framework libary provides all the functionality for loading and running modules as well as data managing and logging functionality. It also manages the access to the processing chain configuration file. Both, GUI and CLI access the framework functionality through the same programming interface.

processing chain if necessary. A binary format is also harder to extend in case new functionality is added to the framework. Therefor a text format should be defined for storing the processing chain.

It is preferable to use an existing format instead of inventing a new one as people may be already familiar with it, which reduces the learning curve for people starting to use the framework. Also for existing formats, there are already existing help resources as well as libraries dealing with it. Also other programs may be developed more easily for working with the files, e.g. for automatic generation of processing chains from other contexts.

There exists a variety of file formats, of which the following seem to be very popular JSON, XML, YAML [Ben-Kiki 2005].

YAML when compared to alternatives like JSON or XML provides the advantage of being easy to read. An advantage over JSON is that it is easier to write and it allows comments. XML is very verbose and also does not provide data types other than String. To define Integer or Boolean values in XML requires the annotation of each entry with meta data which makes simple specifications already quite verbose [Ben-Kiki 2005]. Following from this the storage format of choice for the processing chain file is YAML.

4.3.1.2 Structure of a Processing Chain YAML File

The base of a processing chain YAML file is a YAML map of key-value-pairs. It contains at least one key named `chain`. Further keys can store additional information about the processing chain. This is mainly done to easily extend the format later with additional information e.g. a description, or other options like global parameters or a working directory. These are currently not implemented.

The value of `chain` is a YAML map representing the processing steps. Each key represents the name of the processing step, while the values are again YAML maps for the step properties. A processing step name may contain any character in an ASCII compatible character set, except a dot and parenthesis (ASCII 40,41,46):

```
chain:
  step1: ...
  step2: ...
```

Properties of a processing step are `module`, `inputs`, and `params`. The `module` is a string with the id of the module used in this step. `inputs` is a YAML map where the keys reference the names of the inputs of the module of this step and the value is a string that is interpreted to determine how to obtain the input data. The currently supported options for determining the input data are the following:

- a string referencing another step and the output name separated by a dot (ASCII 46), for example `load.image` references the `image` output of the `load` processing step.
- as explained in Section 4.2.3, if the output of a module is a list of the data types needed for an input of this step, the reference may be appended by `.map()`, which will cause the module of this step to be applied to each element of that list. This will make the processing step itself output a list instead of a single data item. For example `load.images.map()` will apply the current steps module on each image that is contained in the `images` list of the `load` step.

Listing 4.1 shows a YAML configuration of the processing chain shown in Figure 4.4, which computes a Gaussian blur for an image.

4.3.2 Graphical User Interface

The Graphical User Interface (GUI) serves two different use cases. The first is the creation of a processing chain by combining modules, and the second is running a processing chain and exploring the output data of the processing steps.

These two use cases are reflected in the design of the user interface, which on the left hand side provides tools for manipulating the processing chain, and on the right hand side provides the control and data inspection tools for running and evaluating the processing chain. In the middle of the GUI window, the dependency graph of the currently loaded processing chain is visualised. A screenshot of the GUI layout is shown in Figure 4.7.

Listing 4.1 Example processing chain YAML file, representing the processing chain shown in Figure 4.4. It defines a processing chain containing 4 processing steps: `load`, `generate_kernel`, `gauss_blur`, `store`. The `load` and `generate_kernel` steps provide the input data for the `gauss_blur` step, which is defined via the dependencies in `inputs`. The `store` step depends on `gauss_blur` and will store the processing result in a file.

```
chain:
  load:
    module: load_image
    params:
      file: image.png
  generate_kernel:
    module: gauss_kernel
    params:
      sigma: 10
  gauss_blur:
    module: image_avg
    inputs:
      image: load.image
      kernel: gauss_kernel.kernel
  store:
    module: store_image
    inputs:
      image: gauss_blur.image
    params:
      file: image_blur.png
```

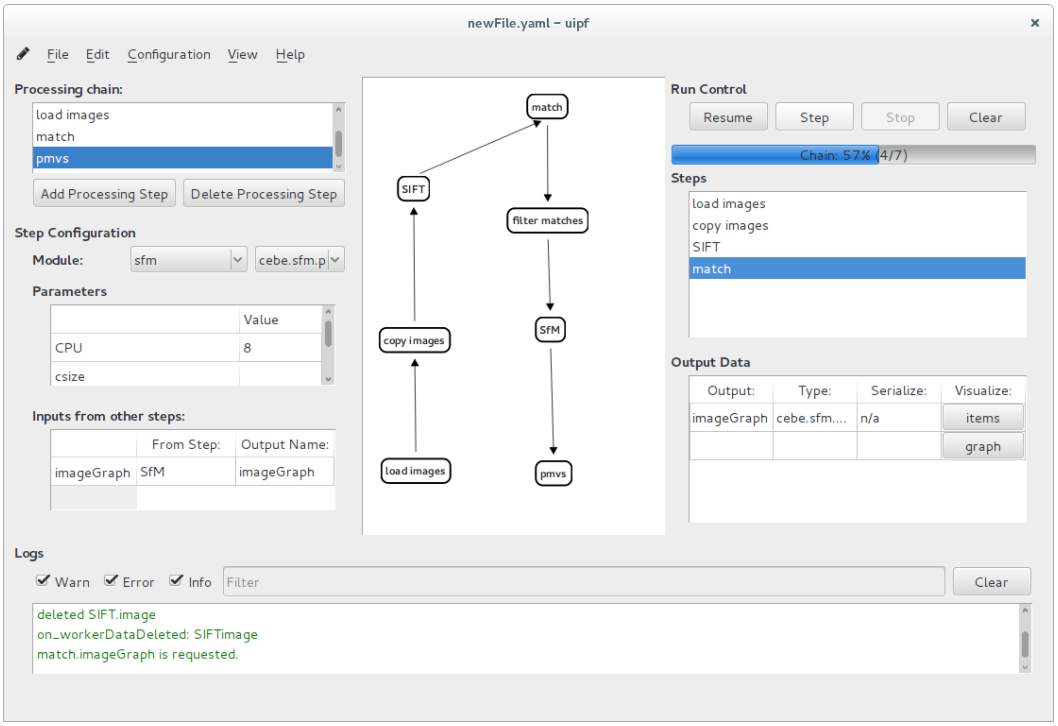


Figure 4.7: Screenshot of the GUI layout, showing the controls for manipulating the processing chain on the left hand side, a dependency graph visualisation in the middle, and the controls for running the processing chain and inspecting the output data on the right hand side.

4.3.2.1 Processing Chain Creation

The left part of the GUI for designing the processing chain is structured as follows (cmp. Figure 4.7): It contains a list of existing processing steps that reflects all steps currently added and which are also shown in the graph visualisation. Below that a selector for the module and two lists are displayed, which reflect the parameters and inputs of the currently selected step from the processing step list. The parameter list is a table of parameter names and their values. A parameter description is shown when the mouse is moved over the parameter row in the table. The input table consists of three columns: the first is the name of the input of the current step's module. After that the referenced source step can be selected. If the source step has multiple outputs the referenced output is selected in the third column. The selection is made via drop-down lists and also directly applies type checking by only providing options that match the type. If an output is a list, the third dropdown also provides the `.map()` option which was explained in Section 4.2.3

4.3.2.2 Processing Chain Execution

The right panel of the GUI is for controlling the execution of the processing chain and visualisation of the output data (cmp. Figure 4.7). The upper part of the panel contains the buttons for run control, the lower part lists the processing steps that have already been executed together with their output data.

The control buttons provide the option to run the chain as a whole or step through it. When the chain runs, a click on the "pause" button will pause after the current step has finished. Pausing will allow to inspect the intermediate output data. Inspecting the data is also possible while the chain is running and data will show up as soon as a module populates it. An already populated output data may be updated by the module multiple times while running. This allows inspecting intermediate results for modules that take a long time to run. It is possible to stop the chain at any time, which will result in the execution to be aborted. A clear button allows to remove all data to start the chain again.

Below the control buttons a progress bar shows the overall progress of the processing chain. If a module does not provide any progress information, the progress bar will only move forward after a module is finished. In case the module provides progress data, an additional progress bar is displayed, which shows the module's individual progress. The overall progress is also adjusted accordingly.

$$p = \frac{n_f}{n_s} + \frac{1}{n_s} \cdot p_s \tag{4.1}$$

with n_s being the number of all processing steps, n_f the number of steps that are already finished, and $0 \leq p_s \leq 1$ the progress of the current module. This allows the user to get an idea about the current status of the processing which is helpful for complex computations to known whether progress is made and how long it may take for it to finish.

4.3.2.3 Output Data Visualisation

Below the run control buttons and progress bar a list of steps that have already finished is shown. When a step is selected, the outputs are shown and made available for visualisation. The visualisation system is designed to be extensible in the way that the implementation of the visualisation is part of the data structure class, which can use an interface provided by the GUI to open windows. A data structure implementation provides the following methods:

- `visualizations()`, which returns a list of string identifiers to show possible visualisation options. An empty list means that the data type can not be visualised.

- `visualize(option, context)` which is called by the GUI when a visualisation is selected. The GUI passes the selected visualisation option and an object called VisualizationContext, which provides methods for opening windows to display images or text.

Besides the possibilities of opening windows in the GUI, a visualisation may also just open an external program or display its own windows independently of the GUI program. This is for example implemented for visualising 3D models and point clouds. Modules can use a library that comes with the framework that controls the visualisation of 3D data in Geomview. Geomview [Phillips 1993] is an Open Source Software for displaying 3D visualisations. It allows other programs to control the output via a control language. This allows adding 3D objects to a scene on demand and also to replace or remove them, which provides a very flexible and powerful way for 3D visualisation.

4.3.2.4 Visual Feedback

The overall design of the GUI builds on visual feedback for the user so it is easy to follow the changes in the processing chain when editing or following the progress of processing chain execution.

The processing step dependencies are visualised as a graph, which allows the user to navigate even complex processing chains with many interdependent modules while still keeping a good overview.

The graph also provides feedback about the current state when running the processing chain. Processing steps that have finished are shown in green to indicate overall processing chain progress. In case of failure, the processing step that caused the failure is marked red, so it is directly visible where the problem occurred and it can be addressed by selecting the step, and checking its configuration.

Figure 4.8 shows the status of an example processing chain, which has executed and stopped after an error occurred in one step.

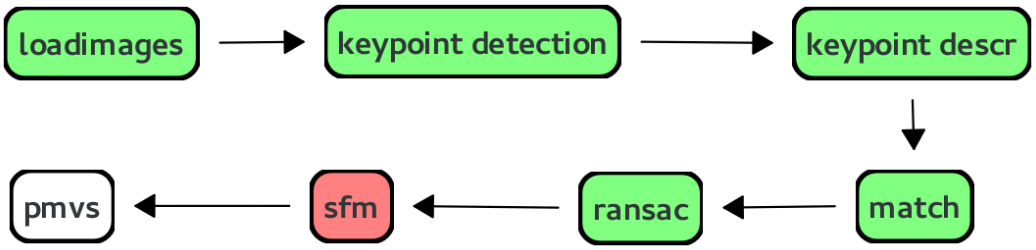


Figure 4.8: Visual feedback in the processing chain overview. Steps that have successfully finished are shown green. Unfinished steps are white. A failing step "sfm" has been marked red to indicate an error.

4.3.3 Command Line Interface

The purpose of the command line interface is mainly to run processing chains. This is useful for running processing chains on a server which is accessed remotely through a console connection like SSH.

It may also be used to execute single modules without implementing a processing chain which can be used for testing a module while developing or to exploit the functionality provided by a module in a context outside of a processing chain or the framework itself.

The command line interface comes as a binary that can be controlled with several options. The command `uipf -c chain.yaml` will read a processing chain from the file `chain.yaml` and execute it. The command line interface does not implement visualisation options, so modules have to be configured to write desired data to the disk. A command line option to specify output data, which should be visualised, could be added to provide this functionality, but this is currently not implemented.

The command `uipf <modulename>` can be used to create a processing chain on demand that runs a single module. The inputs and parameters may be specified on the command line using the `-i` option followed by the name of the input, a colon (`:`) and a value indicating how to obtain the input. This will be passed to the constructor of the data type implementation and in case of a file name, the data structure class will load this data from a file. Similarly parameters are specified with the `-p` option followed by the parameter name, a colon (`:`) and a parameter value.

4.4 Extending the Framework

The framework allows adding new functionality by creating new modules and data types. This section contains a short introduction into the creation of modules and data types in C++. A more detailed explanation can be found in the framework documentation, that is made available together with the code. See Appendix A for references.

4.4.1 Implementing a Module

For the implementation of a simple module, a single C++ file is sufficient. The implementation is based on extending a C++ class provided by the framework, which covers all that is needed for dynamic loading of the module, as well as communication with the framework. This results in a module implementation to only contain code, that is needed for the module to work.

An important part of the module implementation is the module meta data, that defines the information needed for the framework to understand how the module works. This includes a unique ID, name, description, the list of input and output data, as well as the parameters, that are available. Name, description, and category are used for displaying the module for selection in the GUI. The ID is used to reference a module from a processing step. It must be unique among all modules and should consist of multiple parts. It should begin with a vendor name followed by a dot (`.`) and then some string identifying its purpose. The vendor name should be chosen in a way that is unlikely to conflict with others, for example be based on the organisations name for which the module was created. Modules provided by the framework, which are unrelated to SfM are prefixed with `uipf` (for Unified Image Processing Framework), SfM modules for my thesis are prefixed with `uipfsfm`. Other modules, for example when created in the Computer Vision and Remote Sensing department at TU-Berlin could use `tuberlin-cvrs` as the vendor prefix. The part of the name after that prefix can be freely chosen and may use more dots to create a hierarchy, like e.g. `uipfsfm.keypoints.sift`.

Module inputs and outputs have a name, a description and a type. They also have a marker that indicates whether they are optional or not. So a module can be implemented to be used in flexible ways, e.g. some outputs may only be calculated if requested by other steps, or inputs and parameters may have default values if not provided.

Module meta data is defined using a set of C macros. The only definition in C++ code is a method `run()` on the module class defined by the `UIPF_MODULE_CLASS` macro. This method is invoked when running the processing chain and implements all the module logic. An example meta data definition is shown in Listing 4.2.

The module communicates with the framework via methods defined in the parent class. For retrieving input data there is `getInputData(name)`. Input data is requested by its name which will provide a pointer to the object holding the input data. The implementation uses the smart pointer feature of C++11 to make sure

Listing 4.2 Example module implementation showing the macros for module meta data definition. The module implementation should be placed in the `run()` method.

```
#include <uipf/data.hpp>
#include <uipf/data/opencv.hpp>

using namespace uipf;

#define UIPF_MODULE_ID "opencv.imgproc.resize"
#define UIPF_MODULE_NAME "Resize Image"
#define UIPF_MODULE_CATEGORY "opencv"
#define UIPF_MODULE_CLASS OpenCVResizeImage

#define UIPF_MODULE_INPUTS \
  {"image", DataDescription(data::OpenCVMat::id(), \
            "the input image.")}

#define UIPF_MODULE_OUTPUTS \
  {"image", DataDescription(data::OpenCVMat::id(), \
            "the resized image.")}

#define UIPF_MODULE_PARAMS \
  {"width",  ParamDescription("new width.") }, \
  {"height", ParamDescription("new height.") }

#include <uipf/Module.hpp>

void OpenCVResizeImage::run() {
  // Module Implementation goes here.
}
```

data objects can be passed around safely and get deleted, when they are not used anymore. A pointer for the output data is created by the module, when it is ready and can be passed to the `setOutputData(name, data)` method together with the output name. Output data may be updated, while the module is running, to allow inspection of the data visualisation, even when the final output is not computed yet. Subsequent calls to `setOutputData()` will update the visualisation data in the GUI. Parameters can be read using the `getParam(name, defaultValue)` method, which provides multiple ways of casting parameter values to different types. It can be used to retrieve String, Integer, Float and Boolean parameters. The conversion is made from the value provided in the YAML file, which is String by default. For Boolean parameters, the values `true`, `yes`, `y`, `t`, and `1` are considered true. All other values are converted to false.

Listing 4.3 An example CMakeLists.txt file for building a module with CMake.

```
cmake_minimum_required(VERSION 3.1.0)
project(my-module)

set(CMAKE_CXX_STANDARD 11)

find_library(ModuleBase uipf-module)

add_library(MyModule SHARED MyModule.cpp)
target_link_libraries(MyModule ${ModuleBase})
```

A running module may provide progress information to be displayed in the GUI. This can be done by calling the method `updateProgress(`p_i`, `p_n`)`, to indicate how many items p_i out of p_n have been processed. This method will calculate the current steps progress as $p_s = \frac{p_i}{p_n}$ and update the GUI as described in Section 4.3.2.2.

To integrate the module with the framework, it must be compiled into a shared library. For compiling a module on Linux the following command can be used:

```
g++ -std=gnu++0x -shared -o libMyModule.so MyModule.cpp \
    -fPIC -luipf-module
```

This will compile the module implemented in `MyModule.cpp` into a shared library `libMyModule.so` using C++11 and links this to the `libuipf-module` library provided by the framework. The equivalent configuration needed to build a module with CMake[1] is shown in Listing 4.3.

The framework will search for modules in various places, which includes the current working directory, where the YAML configuration file is located, as well as other paths which may be configured in a configuration file.

4.4.2 Implementing a Data Type

Implementing a data type is similar to implementing a module. The minimum amount of code however is much smaller, because there is no meta data. Listing 4.4 shows a minimal implementation of a data type that wraps a list of strings in C++. Data types are implemented in a header file, which is included in the code files where the type is used. In case additional methods are implemented for the data type, the implementation should be compiled into a shared library, to which all modules, that are using this data type, should be linked.

The data type may implement additional functionality like serialisation and visualisation. The implementation for these is done by extending methods of the parent class and will be placed between the `UIPF_DATA_TYPE_BEGIN` and `UIPF_DATA_TYPE_END` calls.

[1] CMake: A cross platform build software for C and C++. https://cmake.org/

Listing 4.4 Example data type implementation using the macro provided by the framework.

```
#include <string>
#include <vector>
#include <uipf/data.hpp>

UIPF_DATA_TYPE_BEGIN (StringList, "tu-cvrs.StringList", \
                      std::vector<std::string>)

// class member implementations will be placed here.

UIPF_DATA_TYPE_END
```

Serialisation is implemented by overriding the `isSerializable()` method to return true and implementing a method `serialize(ostream)` and a constructor that receives an input stream. The serialisation method should write the serialised form of the data to the output stream passed to the method. The constructor should read the serialised representation from the input stream passed to it and fill the class properties with the interpretation of it.

The implementation of the visualisation is already explained in Section 4.3.2.3.

Chapter 5

Implementation of SfM Data Types and Modules

Contents

5.1 Preprocessing and Feature Extraction 56
5.1.1 Data Structures 56
5.1.2 Module Implementation 56
5.1.3 Visualisations 57
5.2 Feature Matching 57
5.2.1 Data Structures 57
5.2.2 Module Implementation 58
5.2.3 Visualisation 58
5.3 Filtering of the Image Graph 59
5.3.1 Geometric Verification with RANSAC 60
5.4 Geometry Estimation and Bundle Adjustment 60
5.4.1 Data Structures 61
5.4.2 Module Implementation 61
5.4.3 Visualisation 61
5.5 Dense Reconstruction 62
5.5.1 Module Implementation 62
5.5.2 Visualisation 63
5.6 Surface Reconstruction 64

The goal of my work is not only to provide the framework, but also to have at least one working module for each step in the SfM processing chain. This chapter covers the implementation of data types and modules for a basic SfM processing chain using the framework described in Chapter 4. These will be derived from the analysis of existing processing chains in Chapter 3, specifically fill the steps visualised in Figure 3.4.

The structure of this chapter will be implementation driven and follow the creation process of the processing chain, covering the modules and data types needed for each step. The modules implemented here are all wrappers around existing libraries and programs which are provided as Open Source or executable binaries by the authors as listed in Section 3.3. For details of the implemented algorithms I will refer to their papers.

5.1 Preprocessing and Feature Extraction

The first steps in the processing chain are the pre-processing and feature extractions steps (cmp. Sections 2.1.1 and 3.1.1.1). These steps work on images only. The input data is a list of images, and the output is the same list of images holding meta data and a list of key points attached to each image.

5.1.1 Data Structures

For this implementation, the first data structure needed is the `Image` that is capable of storing a reference to the image's data file, which is located in a working directory of the processing chain and can be used to load the image for processing it. It should not hold the raw image data in memory all the time, because it will take a huge amount of memory, which is not acceptable if the processing chain should work on a large set of images. Additionally it should also store meta data, which was read from the EXIF [CIPA 2012] data contained in the image file, if available. Additional meta data are the image dimensions and, if available in the EXIF data, the focal length f. The focal length should be stored as it can later be used as prior information on the geometry estimation algorithm.

For the key point detection step the `Image` data structure must be able to store the key points as a list of key point items. For the implementation I use the characteristics of a `Keypoint`, as it is implemented in the OpenCV image processing library [Bradski 2000], which has the following properties: (1) the pixel coordinates of the key point (k_x, k_y), (2) the scale k_s, and (3) the orientation angle k_α.

Each key point also stores a descriptor vector, which in case of SIFT has a dimension of 128 [Lowe 2004], but may have other dimension for different key point detectors.

5.1.2 Module Implementation

The implementation of this step includes two modules, the *Load Images* module and the *SIFT* module.

The *Load Images* module loads image files from a directory on the file system, and provides one output, which is a list of `Image`s including their EXIF meta data. The *SIFT* module implements key point detection and description by wrapping the SIFT binary provided by David Lowe [Lowe 2004] on his website[1]. The module takes an image as input and returns that image unchanged, but adds the key point list data structure to it.

The processing chain of this step makes use of the `map()`-feature, described in Section 4.2.3, as the key point detection will be applied to each item of the list, which is returned by the *Load Images* module. The result will be a list of images with attached key points.

[1] SIFT Website: `http://www.cs.ubc.ca/~lowe/keypoints/`

Figure 5.1: Visualisation of detected key points in an image. It shows the position of the key points and indicates the scale and orientation by drawing a circle.

5.1.3 Visualisations

This step includes two types of data, which can be visualised, the image and the key points. A trivial visualisation is to show the loaded image for inspection. The framework is able to show the list of images loaded and to display each image in a window for the user to inspect the set of images, that will be used in later processing. For each image, also the loaded EXIF meta data can be listed as a text view.

Another visualisation is the visualisation of key points. This includes the key points' positions, as well as their scale and orientation. This way a manual validation of the processing result is possible after key points have been detected. An example visualisation is shown in Figure 5.1. Each key point is visualised as a circle around its position. The scale determines the radius of the circle and to indicate the angle, a line starting in the centre of the circle is drawn towards the outer line.

5.2 Feature Matching

This part of the processing chain implements a key point matching module based on the matcher program, which is part of the Bundler SfM program [Snavely 2006], which has been described in Section 3.3. The module finds matching point pairs in images to create an image graph from them.

5.2.1 Data Structures

This step introduces the `ImageGraph` and `ImagePair` data structures. The `ImageGraph` consists of two lists. The first is a list of all images contained in the graph to allow iteration over the single images in later steps, the second is a list of `ImagePair`s, that define the edges of the graph between connected images. The

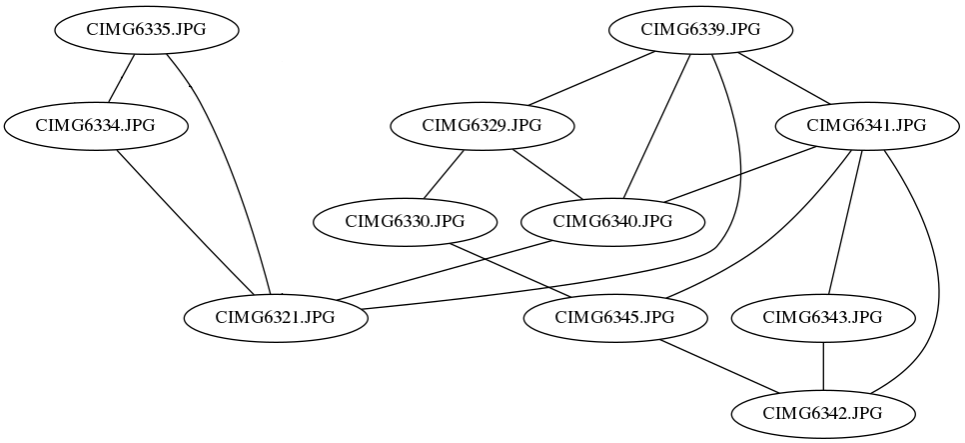


Figure 5.2: Visualisation of an image graph. Nodes are the images and connection lines indicate images that show the same part of the scene.

`ImagePair`s are created from the information given by the matcher program, which returns a list of images that are related, as well as the indexes of the matching point pairs. So each image pair has to store the indexes of the corresponding point pairs. In image pair may optionally also contain the Fundamental Matrix $\mathbf{F}$, if it is calculated for verification of the matches.

5.2.2 Module Implementation

The module implementation is a wrapper around the external program provided by Bundler [Snavely 2006]. It prepares the input key point lists for each image in the file format expected by Bundler and reads back the matching file, that is generated by it. The matcher program uses an approximate nearest neighbour search for matching the images and also includes a RANSAC approach for estimating the Fundamental Matrix (cmp. Section 2.1.3.5). This however is not returned by the program and can not be added to the ImagePair data structure.

5.2.3 Visualisation

A new data type also comes with adding new options for visualisation. The image graph can be visualised as a whole by drawing a graph showing which images are related. The nodes of the graph are created by showing the file name of the original image and draw connections for each image pair. The drawing of the graph is created using GraphViz [Gansner 2000]. An example is shown in Figure 5.2.

The inspection tool in the GUI can also show the list of image pairs. For each image pair a visualisation of their matching key points can be shown, as can be seen in Figure 5.3. Therefor the two images are displayed side-by-side and connection lines are drawn between matching key points' pixel positions. This allows the user to visually verify the behaviour of the matching algorithm.

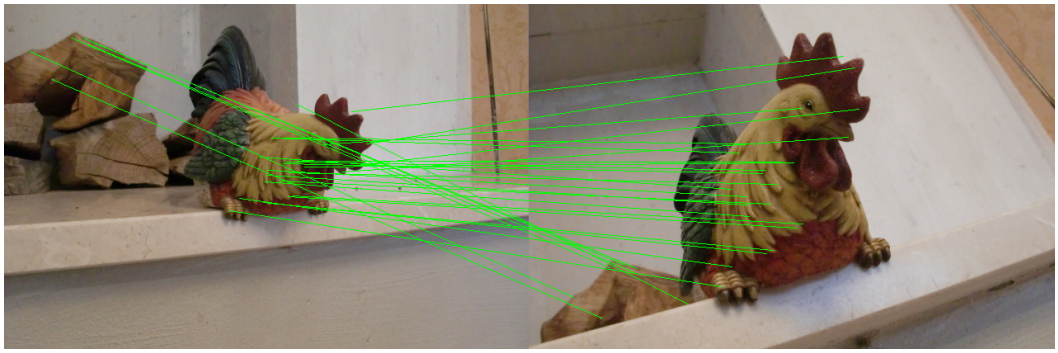


Figure 5.3: Visualisation of the matching of key points.

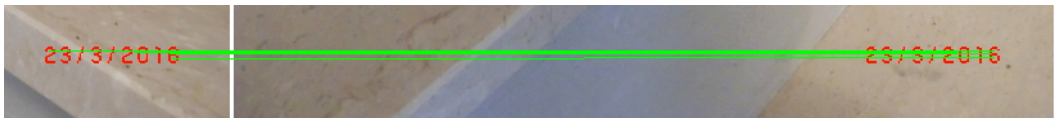


Figure 5.4: Wrong key point matching on the time stamp.

5.3 Filtering of the Image Graph

The example data set I used when implementing the modules in this chapter contains time stamps on each image, which have been added by the camera when the photos have been taken. I have not shown these in the previous figures to avoid distraction, however they cause problems with improper key point matching, as can be seen in Figure 5.4. To avoid this, I have implemented a simple filter module that filters these wrong matches from the image graph.

Regular key point matches will usually never have the same pixel coordinates in both images, because they picture the scene from another perspective. The filter module will remove matches with coordinates (m_{1x}, m_{1y}) in one image, where the coordinates in the second image (m_{2x}, m_{2y}) are within a window w:

$$(m_{1x} - w \leq m_{2x} \leq m_{1x} + w \ , \ m_{1y} - w \leq m_{2y} \leq m_{1y} + w) \tag{5.1}$$

Another option of the filter module is to filter out image pairs, that do not seem to have good matches. This allows setting a threshold for a minimum number of matches per image pair. All image pairs with fewer matches will be removed from the image graph.

This module is not an ideal solution, as the outliers should have been removed inside of the matching process to not influence the verification and RANSAC calculation used in the matching process. However as the module calls an external binary, there is no way to inject that functionality. It still removes the wrong matches sufficiently so they are not used when estimating the Fundamental Matrix in the next step and allows the creation of useful results. Without this module, the geometry estimation step failed on the dataset I tested.

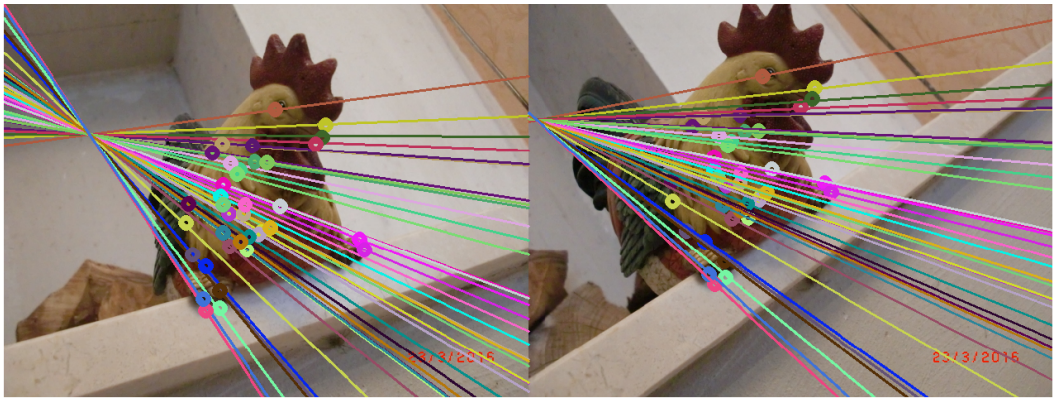


Figure 5.5: Visualisation of the epipolar geometry of an image pair. The image shows corresponding point pairs as well as their epipolar lines. Each point-line pair has a randomly assigned colour to be able to visually distinguish them. The intersection of the epipolar lines indicates the position of the epipole.

5.3.1 Geometric Verification with RANSAC

After the matching process it is desirable to further verify the matching and remove outliers. Therefor a module is implemented, which does geometric verification using RANSAC (cmp. Section 2.1.3.5). The computation of the Fundamental Matrix using RANSAC is implemented using the methods provided by OpenCV [Bradski 2000].

With this module we have an estimation of the Fundamental Matrix, which is added to the images and introduces the possibility of another visualisation method. We can visualise the Epipolar Geometry for each image pair by drawing the epipolar lines in an image. This is useful to visually verify the estimated Epipolar Geometry.

The visualisation is generated by selecting corresponding points pairs from the list of matches up to a threshold. The threshold is needed to make the visualisation useful in cases, where thousands of matching key points were found. For each point in one image the epipolar line can be calculated using the Fundamental Matrix (cmp. Section 2.1.3.2). Doing this for each point pair results in a line through each key point. A random colour assignment allows to easily find the corresponding point and line in the second image. An example of the epipolar line visualisation is shown in Figure 5.5.

5.4 Geometry Estimation and Bundle Adjustment

This step provides the core of the SfM processing chain, which is the estimation of the Epipolar Geometry (cmp. Section 2.1.3.4) and the sparse point cloud (cmp. Section 2.1.3.6). It includes the implementation of a module as well as some additions to the `Image` data structure and an additional visualisation option. It also introduces the `PointCloud` data structure and its visualisation.

5.4.1 Data Structures

For storing the result of the geometry estimation step, the `Image` data structure needs to provide storage options for its projection matrix $\mathbf{P}$, as well as the external camera parameters $\mathbf{R}$ and $\mathbf{t}$, and internal camera parameters $\mathbf{K}$, which includes the focal length f. These are added as additional properties of the class.

The `PointCloud` is a list of 3D points (holding X, Y, Z coordinates). For each point it may optionally contain colour information. The colour is also a three dimensional vector (holding R red, G green, and B blue), which takes values of $0 - 255$ for each channel.

5.4.2 Module Implementation

The geometry estimation module is based on the Bundler SfM program [Snavely 2006], which implements an incremental SfM algorithm including Bundle Adjustment. The module is wrapping Bundler's functionality by creating the expected files, executing the Bundler program and reading the resulting output from files generated by the program.

The input to the Bundler program is a list of images, as well as the list of matching point pairs for each image pair. The output generated by the Bundler program contains external camera parameters $\mathbf{R}$ and $\mathbf{t}$, and the focal length f, from which, given the image dimensions, the calibration matrix $\mathbf{K}$ can be derived as explained in Section 2.1.3.1. Given $\mathbf{K}$, $\mathbf{R}$ and $\mathbf{t}$, the projection matrix $\mathbf{P}$ can be derived using Equation 2.9.

5.4.3 Visualisation

In this step additional data has been generated, which can be visualised. This is (1) the estimated camera positions in 3D space, and (2) the sparse point cloud of the scene.

The camera positions for each image are described by the external parameters $\mathbf{R}$ and $\mathbf{t}$. The position of the camera centre is defined as $\mathbf{C} = -\mathbf{R}^{-1}\mathbf{t}$. The viewing direction of the camera, assuming that an unrotated camera will look along the negative z-axis is defined as:

$$\mathbf{d} = \mathbf{R}^{-1} \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} \tag{5.2}$$

Using $\mathbf{d}$, we can visualise the field of view of the camera by deriving a set of four spanning vectors that span up the field of view starting in the camera centre.

$$\mathbf{v}_{1,2,3,4} = f \cdot \frac{\mathbf{d}}{\|\mathbf{d}\|} \pm \underbrace{\frac{o_w}{2} \cdot \frac{\mathbf{n}_x}{\|\mathbf{n}_x\|}}_{\text{span in } x} \pm \underbrace{\frac{o_h}{2} \cdot \frac{\mathbf{n}_y}{\|\mathbf{n}_y\|}}_{\text{span in } y} \quad \text{with} \quad \mathbf{n}_x = \mathbf{d} \times \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \; \mathbf{n}_y = \mathbf{n}_x \times \mathbf{d} \tag{5.3}$$

In Equation 5.3, o_w and o_h are the width and height of the CCD chip of the camera, f is the camera's focal length. Using these vectors, a pyramid can be drawn, which

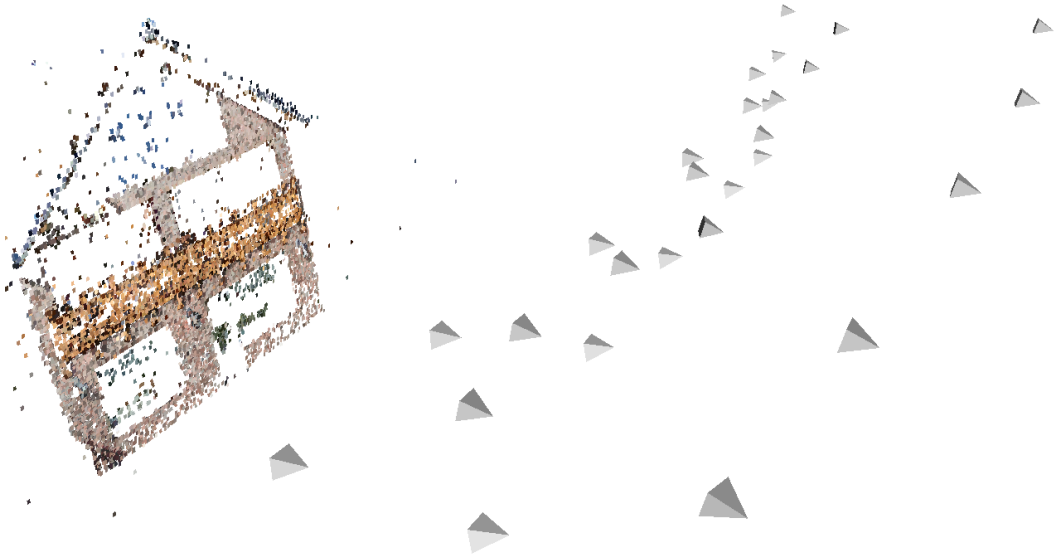


Figure 5.6: Visualisation of camera positions relative to the reconstructed point cloud of a building front.

has the height of the focal length of the camera and spans the field of view of the camera so the viewing direction of the camera is equal to the horizontal line going from the top of the pyramid through the bottom plane. An example visualisation is shown in Figure 5.6.

The visualisation of the point cloud is explained in Section 5.5.2 below.

5.5 Dense Reconstruction

The dense reconstruction part adds a module to the processing chain. The input is a set of images, for which the projection matrix **P** has been estimated, and the output is a dense point cloud. From the data structure point of view there is not much difference to the sparse point cloud, except that the dense reconstruction provides normal information for the points in the point cloud, so the data structure and visualisation has to be adjusted to support this.

5.5.1 Module Implementation

The module implementation is a wrapper around the PMVS2 program created by Yasutaka Furukawa and Jean Ponce [Furukawa 2010b]. This has been explained in Section 3.3. The module works in a temporary working directory where the expected directory structure is created. The inputs to the PMVS program are the set of images, the projection matrix **P** for each of them, and a configuration file. These can be generated from the `ImageGraph` data structure by copying the images into a directory called `visualize` under the working directory following the naming

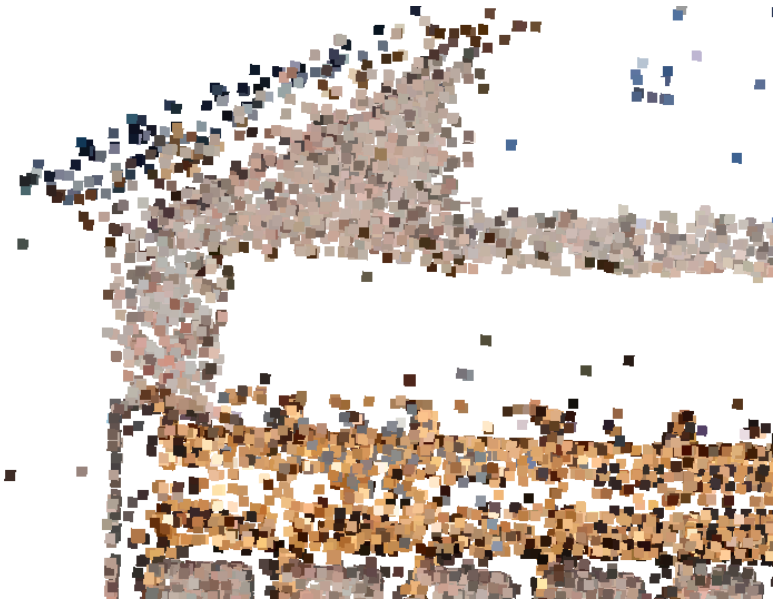


Figure 5.7: Visualisation of a point cloud as squares.

schema expected by the program. For each image, the projection matrix is stored in a text file in a format described in the PMVS documentation.

A configuration file is created, which indicates which images should be used for reconstruction. Additionally all parameters of the PMVS configuration file are created as module parameters, so that they can be controlled from the frameworks processing chain configuration.

The output of the program is a file in PLY-format [Bourke 2009, Turk 1998], which contains the dense point cloud, which for each point stores the normal estimation as well as a colour. The PLY file format is described in Appendix B.

5.5.2 Visualisation

The visualisation of the point cloud is not trivial, because a point has the size of zero. To make the points visible an object has to be created that describes the position of the point. In case colour information is available, this object should also be coloured to make it easier to recognise the original scene when observing the point cloud.

I have implemented the visualisation of points by converting the point cloud into a polygon mesh, that contains small squares around each point's position. The four corners of the square are calculated by adding vectors to the point position, which are derived from the point normal if available. If no point normal is available the normal direction is assumed to be along the z-axis. Figure 5.7 shows a point cloud visualisation of a part of a building front.

5.6 Surface Reconstruction

Surface reconstruction is the next step after dense reconstruction to create a surface mesh. A surface reconstruction module has been implemented similar to most of the modules before. It wraps the PoissonRecon software which implements the algorithm by [Kazhdan 2013]. It is also part of the software list in Section 3.3.

The input data of the module is a point cloud, which is serialised and stored in a file using the PLY file format (Appendix B). This file is then given to the PoissonRecon program to perform the surface reconstruction.

The algorithm works on points clouds that contain normal information. The point normal is a vector indicating the orientation of the surface the point is located on in the real scene. The vector stands orthogonal to the surface and can be used for reconstruction of the surface.

PoisonRecon can also use colour information to add a colour for each triangle on the mesh. This allows visualising the object in a similar way that would be possible with real texture. A real mesh texturing algorithm however has not been implemented.

The output of the Surface Reconstruction is a triangle mesh which is also stored in the PLY file format, which can be used for further processing or to view the mesh in a mesh viewing program.

Chapter 6

Application to Example Use Cases

Contents

6.1 Datasets 65
6.2 Workflow 65
6.3 The Processing Chain 67
6.4 Results 68

In this chapter I will apply the framework to example use cases to show how it works and validate the functionality.

This is done by an experiment to show that all implemented modules work together in order to create a 3D reconstruction of a scene. The task is to create a processing chain and to apply it to different datasets of scenes with a single camera and a concrete object, to see whether a 3D reconstruction can be obtained.

I will also explain how the framework helps with the task of creating a processing chain and evaluating the results.

6.1 Datasets

The datasets used in this experiment are "Der Hass" by [Fuhrmann 2014], as well as "fountain-P11" and "Herz-Jesu-P8" by [Strecha 2008].

The images in the "Der Hass" dataset show a statue on a stand that has been photographed from positions on a circle around it. It is a rather simple dataset which does not contain many challenges to be solved by an SfM algorithm and provides good structure for point detection and matching. The dataset consists of 79 images.

The "fountain-P11" dataset contains 11 images showing a fountain in front of a stone wall. This dataset also contains a lot of structure which is good for point matching.

The "Herz-Jesu-P8" dataset are photographs of a building front which consists of 8 images. This dataset is a bit more complicated, as it contains repetitive structure which could cause false matches.

6.2 Workflow

For creating the processing chain by putting together existing modules and developing new modules, a small subset of a dataset is used. Images are also resized to

Figure 6.1: Visualisation of the sparse reconstruction of the "Der Hass" dataset by [Fuhrmann 2014]. It shows the sparse point cloud as well as estimated camera positions, which can be selected for display separately.

reduce the computational load and thus the waiting time for results, when changes are made.

Resizing of the image is done by implementing a `resize` module that takes an image as input and outputs a resized version of it. The module can be controlled by parameters which either define a fixed target size or a maximum width or height. Dependent on the specified parameters the desired action is performed on the images. If only one dimension is given, e.g. width, the height is computed by keeping the aspect ratio of the image.

The processing chain for this preprocessing step is: Load Images → Copy Images → Resize → Store Images. The Copy Images step is necessary, to not overwrite the original files. The `resize` module is now part of the framework so it can be reused in later applications.

Using the generated data set with small images, the modules are added to the processing chain and tested whether they work as expected. If something does not work, the visualisation methods help to find the issue. For example a bug in the serialisation method of the key points has been spotted which stored the coordinates in the wrong order (x and y swapped). The only indicator of the problem was a not working reconstruction step, so debugging this would have been much harder without visual verification options.

After the processing chain has been created it can be taken to a different computer or a server to execute it on different datasets. The processing chain file makes it easy to duplicate the same process by storing all information and parameters in one place. This is useful for repeatability of results.

The YAML format also allows copying the processing chain to different directories to run experiments with different set of parameters. In each directory the same

Figure 6.2: From left to right: Sparse point cloud, dense point cloud, surface reconstruction of the "fountain-P11" dataset by [Strecha 2008].

results structure will be generated after all processing chains have been executed.

When running the processing chain on a server using the command line interface, all desired output data should be stored using modules, that are able to store them. For point cloud and image graph these modules have been implemented, which are able to store a point cloud as a file in PLY format and an image graph in NVM format, that contains all images as well as matching key points and sparse point cloud. When results are obtained from the server, data reading modules can be used to load the data for visualisation in the GUI.

6.3 The Processing Chain

The processing chain developed for this case uses the modules that have been implemented in Chapter 5. Key point detection uses the SIFT module, which wraps the SIFT binary by [Lowe 2004]. For feature matching and SfM, the Bundler programs are used [Snavely 2006]. The module for dense reconstruction is PMVS2 [Furukawa 2010b]. From the dense point cloud a surface mesh is created using the Poisson reconstruction module which implements the methods by [Kazhdan 2013].

The steps of the processing chain are Load Images → SIFT → Bundler Matcher → Bundler SfM → PMVS → Surface Reconstruction. Additional processing steps are added, that store the sparse dense point clouds and also the image graph.

Figure 6.3: Surface mesh of the "Der Hass" dataset.

6.4 Results

The processing chain is executed on the dataset to produce a sparse reconstruction. Using the visualisation methods provided by the framework the result can be inspected in a 3D view. It is possible to display the resulting point cloud as well as the estimated camera positions. The display of these can be selected separately. It is also possible to select only specific cameras to be displayed by choosing the single camera visualisation for an image. An example is shown in Figure 6.1.

The visualisation showed, that for the "fountain-P11" dataset Bundler was not able to produce a reconstruction without prior information. The Load Image module allows to specify a focal length as image meta data. By setting the focal length to a certain value, that approximates the real value, the reconstruction succeeds as can be seen in Figure 6.3. The width of the image seems to be a good indicator. On the "Der Hass" dataset the reconstruction was possible without specifying prior information, because this was given in the image EXIF meta data and had been extracted automatically. The reconstruction of the "Herz-Jesu-P8" dataset can be seen in Figure 6.4.

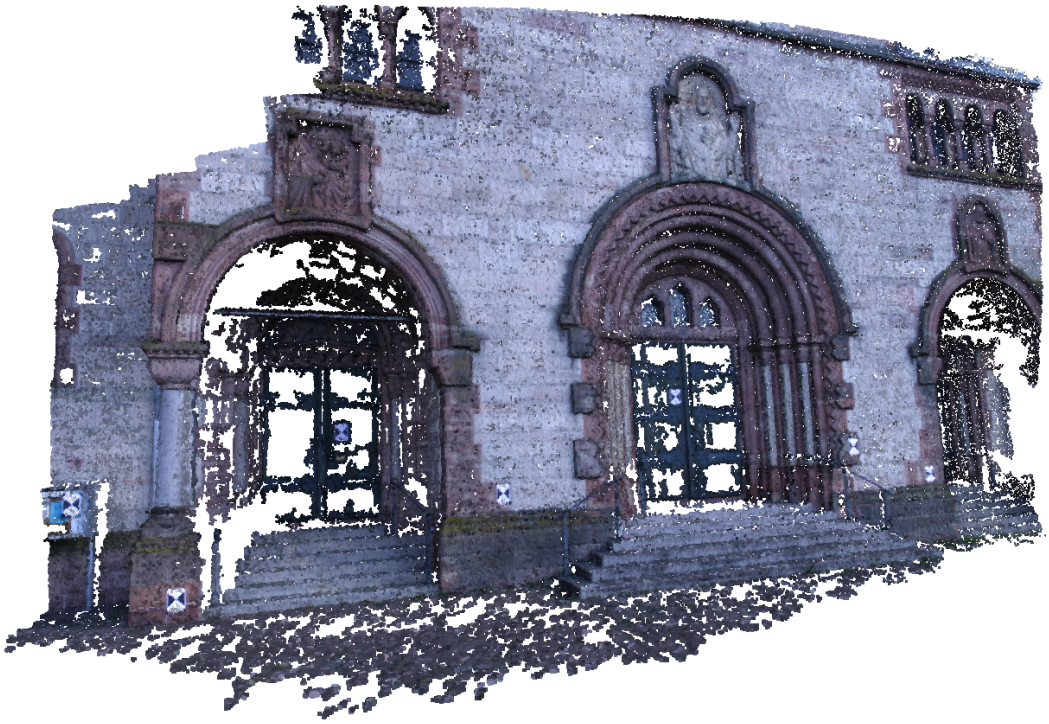


Figure 6.4: Dense reconstruction point cloud of the “Herz-Jesu-P8” dataset.

Chapter 7

Conclusion

The goal of my thesis was the development of a modular framework for image-based 3D reconstruction. I have described the theory behind the process of 3D reconstruction in Chapter 2 and reviewed existing algorithm implementations as well as practical applications in related work in Chapter 3 to get an understanding of how 3D reconstruction works in theory and is implemented in practise.

Based on this, in Section 3.2, the requirements of a framework have been developed, which resulted in the description of a generic 3D reconstruction processing chain, that shows a list of possible processing steps and data structures which need to be supported by a framework to fulfil the desired task.

In Chapter 4 a generic framework for the creation of modular processing chains has been developed, which was filled with the basic functionality for 3D reconstruction in Chapter 5.

To evaluate and validate the functionality of the framework it has been used to carry out reconstructions on different datasets, which are described in Chapter 6.

I have also shown that the framework is extensible, as it allows adding custom implementations of algorithms in different parts of the processing chain. One example is the implementation of a custom filter that has been created for a problem that is specific to a certain dataset in Section 5.3. It is further possible to implement own data structures for processing that goes beyond what has been discussed in my thesis.

Compared to implementing an algorithm in a custom program the overhead of the framework is minimal. The implementation of a module inside of the `run()` method is very similar to writing a C program's `main()` function, but comes with automatic parameter handling, as well as visualisation methods, which otherwise would need to be implemented manually.

The framework has been published as Free Software to allow widespread use and extension. See Appendix A for references.

7.1 Limitations and Future Work

While the current implementation is able to solve the initially stated problem, there are still parts that can be improved to support more use cases and improve the performance. Besides small improvements that can be made to enhance the overall user experience, I want to highlight a few major points that could be improved in future work.

The current model of a processing chain does not allow recursion, which prohibits the exploitation of the incremental reconstruction approach as described by [Wu 2013] and [Schönberger 2016a] on framework level. It is currently not possible to define a processing chain that implements the methods described in these papers from modules that implement Geometry Estimation, Bundle Adjustment or a Next Best View selection. With the current framework design these must be implemented in a single processing step if an incremental approach is desired, or a processing chain has to be executed multiple times by adding more and more images in each execution round. This can be improved by allowing recursive structures in the processing chain, which can be defined in a similar manner as the mapping on list data types, that is described in Section 4.2.3.

The model of the processing chain is designed in the way, that modules are self-contained units, which can be executed independently of each other as long as no dependency is defined via input and output data. This allows the framework to distribute the work among multiple CPU cores. In future work the framework could be extended to expand this concept from a single PC to a cluster of worker servers to allow the execution of the processing chain on large reconstruction problems.

Another improvement can be made in the way modules are implemented. The current implementation is limited to C++ as the programming language. There are a lot of algorithms written in other programming languages such as Python or Matlab, so the development of module bindings for these languages would add value to the framework.

The current framework implementation is focused on algorithms that work fully automatic. The visualisation system is currently designed to only show data produced by the algorithms. To allow the implementation of algorithms that need some kind of user input, the visualisation system could be extended to allow the user to input data or correct visualised results, e.g. remove wrong image matches that are spotted by manual verification.

Finally, the core execution framework described in Chapter 4 is not bound to the use case of 3D reconstruction and provides a very flexible model for processing chain creation, so it would be possible to broaden the application field to other applications in image processing, image analysis, or even fields outside of this scope. I hope that the fact that the code is released as Free Software will encourage others to use it and adopt it to other fields.

Appendix

Appendix A

Open Source

The code created during my thesis is released as Free Software under the GNU General Public License under the name Unified Image Processing Framework (UIPF). The code is available on the Github code hosting platform under the following URL:

`https://github.com/uipf/uipf`

The following is a reference of the implemented Structure from Motion modules, including a short description as well as web links for obtaining the code.

SfM data types and basic modules	`https://github.com/uipf/uipf-sfm`
SIFT module	`https://github.com/uipf/uipf-sfm-sift`
Bundler module	`https://github.com/uipf/uipf-sfm-bundler`
PMVS module	`https://github.com/uipf/uipf-sfm-pmvs`
Poisson reconstruction module	`https://github.com/uipf/uipf-sfm-poissonrecon`

Appendix B

The PLY File Format

The PLY - polygon file format [Bourke 2009, Turk 1998] is a generic file format for storing 3D data structures. It can store points and triangular meshes with additional information like colour or normal vectors. Its overall design is the be very flexible in the type of data that can be stored. The following description is a short summary. For a detailed specification see [Bourke 2009].

A PLY file always starts with three characters `ply` followed by a header that contains meta information about the data stored in the file. The header consists of lines that introduce this information. A line may start with the following identifiers:

- `format`: A PLY file may be stored in plain text or binary format. This field indicates the file type as well as the format version.
- `comment`: A comment can contain a general description of the file and is not meant to be read by a machine.
- `element`: Starts the description of an element, which can be `vertex` or `face`. A face can be for example a triangle in a triangle mesh. It also indicates the number of items that are stored.
- `property`: An element line is followed by several property lines to define properties of the element, which are identified by a type and a name. For example `property float x` describes the x coordinate of a point. Properties describe how the data is structured that is following the header.

The following is an example PLY file which stores a triangle mesh with 959695 vertices and 1919126 faces. The vertices have x, y, and z coordinates as well as colour information. The faces are stored as references to the vertices.

```
ply
format binary_little_endian 1.0
element vertex 959695
property float x
property float y
property float z
property uchar red
property uchar green
property uchar blue
element face 1919126
property list uchar int vertex_indices
```

```
end_header
...
```

Appendix C

DVD with Code, Data, and Results

The attached DVD contains the framework code as well as data used in the experiments, the processing chain configuration files and exemplary results.
The following things are included:

- A PDF version of the thesis
- Source code of the framework and the modules
- Usage documentation of the framework and installation instructions
- Binary packages for Debian and Ubuntu
- Datasets and processing chains from the experiments including results

Bibliography

[Agarwal 2011] Sameer Agarwal, Yasutaka Furukawa, Noah Snavely, Ian Simon, Brian Curless, Steven M Seitz and Richard Szeliski. *Building rome in a day.* Communications of the ACM, vol. 54, no. 10, pages 105–112, 2011. 1, 3, 20, 21, 22, 23, 24, 29, 32

[Amenta 1999] Nina Amenta and Marshall Bern. *Surface reconstruction by Voronoi filtering.* Discrete & Computational Geometry, vol. 22, no. 4, pages 481–504, 1999. 16

[Attene 2010] Marco Attene. *A lightweight approach to repairing digitized polygon meshes.* The Visual Computer, vol. 26, no. 11, pages 1393–1406, 2010. 26

[Bay 2008] Herbert Bay, Andreas Ess, Tinne Tuytelaars and Luc Van Gool. *Speeded-up robust features (SURF).* Computer vision and image understanding, vol. 110, no. 3, pages 346–359, 2008. 34

[Beder 2006] Christian Beder and Richard Steffen. *Determining an initial image pair for fixing the scale of a 3d reconstruction from an image sequence.* In Pattern Recognition, pages 657–666. Springer, 2006. 23

[Ben-Kiki 2005] Oren Ben-Kiki, Clark Evans and Brian Ingerson. *YAML Ain't Markup Language (YAML^{TM}) Version 1.1.* `http://yaml.org`, Tech. Rep, 2005. 43

[Berger 2013] Matthew Berger, Joshua A Levine, Luis Gustavo Nonato, Gabriel Taubin and Claudio T Silva. *A benchmark for surface reconstruction.* ACM Transactions on Graphics (TOG), vol. 32, no. 2, page 20, 2013. 16, 17, 31

[Boissonnat 2000] Jean-Daniel Boissonnat and Frédéric Cazals. *Smooth surface reconstruction via natural neighbour interpolation of distance functions.* In Proceedings of the sixteenth annual symposium on Computational geometry, pages 223–232. ACM, 2000. 17

[Bourke 2009] Paul Bourke. *Ply-polygon file format.* Online resource: `http://paulbourke.net/dataformats/ply`, 2009. 63, 77

[Bradski 2000] Gary Bradski and Adrian Kaehler. *OpenCV.* Dr. Dobb's Journal of Software Tools, Online: `http://opencv.org/`, 2000. 29, 34, 37, 56, 60

[Brandt 2016] Ludmilla Brandt. Path planning for structure from motion under position constraints. Master's thesis, Technische Universität Berlin, `http://www.cv.tu-berlin.de/?id=48006`, Berlin, Germany, 2016. 26

[Cazals 2006] Frédéric Cazals and Joachim Giesen. *Delaunay triangulation based surface reconstruction.* In Effective computational geometry for curves and surfaces, pages 231–276. Springer, 2006. 17

[Chen 2005] SY Chen and YF Li. *Vision sensor planning for 3-D model acquisition.* Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on, vol. 35, no. 5, pages 894–904, 2005. 1

[CIPA 2012] CIPA. *Exif 2.3 metadata for XMP.* Camera & Imaging Products Association, `http://www.cipa.jp/std/std-sec_e.html`, no. CIPA DC-010-2012, 2012. 21, 29, 56

[Cohen 2016] Andrea Cohen, Johannes L Schönberger, Pablo Speciale, Torsten Sattler, Jan-Michael Frahm and Marc Pollefeys. *Indoor-Outdoor 3D Reconstruction Alignment.* In European Conference on Computer Vision, pages 285–300. Springer, 2016. 26

[Daftry 2015] Shreyansh Daftry, Christof Hoppe and Horst Bischof. *Building with drones: Accurate 3d facade reconstruction using mavs.* In 2015 IEEE International Conference on Robotics and Automation (ICRA), pages 3487–3494. IEEE, 2015. 20, 24

[Fabio 2003] Remondino Fabio*et al.* *From point cloud to surface: the modeling and visualization problem.* International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, vol. 34, no. 5, page W10, 2003. 26

[Fischler 1981] Martin A Fischler and Robert C Bolles. *Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography.* Communications of the ACM, vol. 24, no. 6, pages 381–395, 1981. 7, 13, 23

[Förstner 1987] Wolfgang Förstner and Eberhard Gülch. *A fast operator for detection and precise location of distinct points, corners and centres of circular features.* In Proc. ISPRS intercommission conference on fast processing of photogrammetric data, pages 281–305, 1987. 4, 5

[Frahm 2010] Jan-Michael Frahm, Pierre Fite-Georgel, David Gallup, Tim Johnson, Rahul Raguram, Changchang Wu, Yi-Hung Jen, Enrique Dunn, Brian Clipp, Svetlana Lazebnik*et al.* *Building Rome on a cloudless day.* In Computer Vision–ECCV 2010, pages 368–381. Springer, 2010. 1, 3, 20, 21, 22, 23, 24, 25, 26, 29

[Fuhrmann 2014] Simon Fuhrmann, Fabian Langguth and Michael Goesele. *MVE-A Multi-View Reconstruction Environment.* In GCH, pages 11–18, 2014. 23, 65, 66

[Furukawa 2010a] Yasutaka Furukawa, Brian Curless, Steven M. Seitz and Richard Szeliski. *Towards Internet-scale Multi-view Stereo.* In CVPR, 2010. 25, 32, 33

[Furukawa 2010b] Yasutaka Furukawa and Jean Ponce. *Accurate, dense, and robust multiview stereopsis.* Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 32, no. 8, pages 1362–1376, 2010. 15, 25, 31, 32, 33, 62, 67

[Gansner 2000] Emden R. Gansner and Stephen C. North. *An open graph visualization system and its applications to software engineering.* SOFTWARE - PRACTICE AND EXPERIENCE, Online: `http://graphviz.org/`, vol. 30, no. 11, pages 1203–1233, 2000. 58

[Gil 2010] Arturo Gil, Oscar Martinez Mozos, Monica Ballesta and Oscar Reinoso. *A comparative evaluation of interest point detectors and local descriptors for visual SLAM.* Machine Vision and Applications, vol. 21, no. 6, pages 905–920, 2010. 4, 6, 31

[Harris 1988] Chris Harris and Mike Stephens. *A combined corner and edge detector.* In Alvey vision conference, volume 15, page 50. Citeseer, 1988. 5, 15, 34

[Hartley 1997] Richard I Hartley. *Self-calibration of stationary cameras.* International Journal of Computer Vision, vol. 22, no. 1, pages 5–23, 1997. 13

[Hartley 2003] Richard Hartley and Andrew Zisserman. Multiple view geometry in computer vision. Cambridge university press, 2003. 3, 4, 7, 8, 11, 12, 13, 14

[Heinly 2012] Jared Heinly, Enrique Dunn and Jan-Michael Frahm. *Comparative evaluation of binary features.* In Computer Vision–ECCV 2012, pages 759–773. Springer, 2012. 22

[Irschara 2009] Arnold Irschara, Christopher Zach, Jan-Michael Frahm and Horst Bischof. *From structure-from-motion point clouds to fast location recognition.* In Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on, pages 2599–2606. IEEE, 2009. 4

[Irschara 2010] Arnold Irschara, Viktor Kaufmann, Manfred Klopschitz, Horst Bischof and Franz Leberl. *Towards fully automatic photogrammetric reconstruction using digital images taken from UAVs.* 2010. 20, 24, 25

[Jawer 2015] Jens Jawer. Detection of erroneous parts in automatic 3d reconstruction. Master's thesis, Technische Universität Berlin, `http://www.cv.tu-berlin.de/?id=48006`, Berlin, Germany, 2015. 26

[Kauff 2007] Peter Kauff, Nicole Atzpadin, Christoph Fehn, Marcus Müller, Oliver Schreer, Aljoscha Smolic and Ralf Tanger. *Depth map creation and image-based rendering for advanced 3DTV services providing interoperability and*

scalability. Signal Processing: Image Communication, vol. 22, no. 2, pages 217–234, 2007. 1

[Kazhdan 2013] Michael Kazhdan and Hugues Hoppe. *Screened poisson surface reconstruction*. ACM Transactions on Graphics (TOG), vol. 32, no. 3, page 29, 2013. 16, 25, 34, 64, 67

[Ley 2016] Andreas Ley, Ronny Hänsch and Olaf Hellwich. *SyB3R: A Realistic Synthetic Benchmark for 3D Reconstruction from Images*. In European Conference on Computer Vision, pages 236–251. Springer, 2016. 26, 34

[Longuet-Higgins 1981] HC Longuet-Higgins. *A computer algorithm for reconstructing a scene from two projections. 293: 133–135*, 1981. 3, 12

[Lowe 2004] David G. Lowe. *Distinctive image features from scale-invariant keypoints*. International journal of computer vision, vol. 60, no. 2, pages 91–110, 2004. 4, 6, 22, 29, 32, 34, 56, 67

[Mikolajczyk 2005] Krystian Mikolajczyk, Tinne Tuytelaars, Cordelia Schmid, Andrew Zisserman, Jiri Matas, Frederik Schaffalitzky, Timor Kadir and Luc Van Gool. *A comparison of affine region detectors*. International journal of computer vision, vol. 65, no. 1-2, pages 43–72, 2005. 4, 6

[Nister 2006] David Nister and Henrik Stewenius. *Scalable recognition with a vocabulary tree*. In Computer vision and pattern recognition, 2006 IEEE computer society conference on, volume 2, pages 2161–2168. Ieee, 2006. 22

[Olague 2002] Gustavo Olague and Roger Mohr. *Optimal camera placement for accurate reconstruction*. Pattern Recognition, vol. 35, no. 4, pages 927–944, 2002. 7

[Oliva 2001] Aude Oliva and Antonio Torralba. *Modeling the shape of the scene: A holistic representation of the spatial envelope*. International journal of computer vision, vol. 42, no. 3, pages 145–175, 2001. 22

[Over 2010] M Over, Arne Schilling, S Neubauer and Alexander Zipf. *Generating web-based 3D City Models from OpenStreetMap: The current situation in Germany*. Computers, Environment and Urban Systems, vol. 34, no. 6, pages 496–507, 2010. 1

[Phillips 1993] Mark Phillips, Silvio Levy and Tamara Munzner. *Geomview: An Interactive Geometry Viewer*. Notices of the American Mathematical Society, Online: `http://geomview.org/`, pages 985–988, October 1993. Computers and Mathematics Column. 48

[Raguram 2008] Rahul Raguram, Jan-Michael Frahm and Marc Pollefeys. *A comparative analysis of RANSAC techniques leading to adaptive real-time random sample consensus*. In European Conference on Computer Vision, pages 500–513. Springer, 2008. 13

[Rumpler 2011] Markus Rumpler, Arnold Irschara and Horst Bischof. *Multi-view stereo: Redundancy benefits for 3D reconstruction.* In 35th Workshop of the Austrian Association for Pattern Recognition, volume 4, 2011. 25

[Schönberger 2016a] Johannes Lutz Schönberger and Jan-Michael Frahm. *Structure-from-Motion Revisited.* In IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016. 7, 20, 21, 22, 23, 24, 33, 72

[Schönberger 2016b] Johannes Lutz Schönberger, Enliang Zheng, Marc Pollefeys and Jan-Michael Frahm. *Pixelwise View Selection for Unstructured Multi-View Stereo.* In European Conference on Computer Vision (ECCV), 2016. 15, 25, 31, 33

[Schreer 2005] Oliver Schreer. Stereoanalyse und bildsynthese. Springer-Verlag, 2005. 4, 7, 8, 10, 11, 12, 13, 14

[Snavely 2006] Noah Snavely, Steven M Seitz and Richard Szeliski. *Photo tourism: exploring photo collections in 3D.* In ACM transactions on graphics (TOG), volume 25, pages 835–846. ACM, 2006. 1, 6, 15, 31, 32, 57, 58, 61, 67

[Stathopoulou 2011] Ellie K. Stathopoulou, A Valanis, JL Lerma and A Georgopoulos. *High and low resolution textured models of complex architectural surfaces.* International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, vol. 38, no. 5, 2011. 26

[Stathopoulou 2015] Ellie K. Stathopoulou, Ronny Hänsch and Olaf Hellwich. *Prior knowledge about camera motion for outlier removal in feature matching.* In VISAPP 2015-International Conference on Computer Vision Theory and Applications, pages 603–610, 2015. 7, 24

[Strecha 2008] Christoph Strecha, Wolfgang Von Hansen, Luc Van Gool, Pascal Fua and Ulrich Thoennessen. *On benchmarking camera calibration and multi-view stereo for high resolution imagery.* In Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on, pages 1–8. Ieee, 2008. 1, 3, 65, 67

[Triggs 1999] Bill Triggs, Philip F McLauchlan, Richard I Hartley and Andrew W Fitzgibbon. *Bundle adjustment—a modern synthesis.* In International workshop on vision algorithms, pages 298–372. Springer, 1999. 14, 24

[Turk 1998] Greg Turk. *The PLY Polygon File Format.* Georgia Institute of Technology, 1998. 63, 77

[Wefelscheid 2011] Cornelius Wefelscheid, Ronny Hänsch and Olaf Hellwich. *Three-dimensional building reconstruction using images obtained by unmanned aerial vehicles.* International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, vol. 38, page 1, 2011. 1, 4, 6, 16, 20, 24, 25, 27, 29

[Wu 2011] Changchang Wu, Sameer Agarwal, Brian Curless and Steven M Seitz. *Multicore bundle adjustment.* In Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on, pages 3057–3064. IEEE, 2011. 33

[Wu 2013] Changchang Wu. *Towards linear-time incremental structure from motion.* In 3D Vision-3DV 2013, 2013 International Conference on, pages 127–134. IEEE, 2013. 1, 20, 21, 22, 23, 24, 32, 72

[Zhang 1995] Zhengyou Zhang, Rachid Deriche, Olivier Faugeras and Quang-Tuan Luong. *A robust technique for matching two uncalibrated images through the recovery of the unknown epipolar geometry.* Artificial intelligence, vol. 78, no. 1-2, pages 87–119, 1995. 4, 6

[Zhang 1998] Zhengyou Zhang. *Determining the epipolar geometry and its uncertainty: A review.* International journal of computer vision, vol. 27, no. 2, pages 161–195, 1998. 4

A Modular Framework for Image-based 3D Reconstruction

Abstract: In this thesis I analyse the state of the art of Structure from Motion and 3D reconstruction algorithms to develop a framework, that improves the process of working with these algorithms. This framework allows reusing existing parts of a processing chain, as well as developing new algorithms efficiently. This is done by providing a data model and interfaces, which allow inspection of intermediate results, as well as reorganisation of the processing flow.

I first cover the basic principles of 3D reconstruction, and then review existing implementations. Based on that the framework is developed to allow the implementation of modular and flexible 3D reconstruction processing chains.

I also show how this framework can be applied to existing problems to quickly come to a working implementation, as it allows the user to focus on the algorithm implementation details instead of the need to deal with the overall processing chain design.

The implementation of the framework is released as Open Source software.

Keywords: 3D Reconstruction, Structure from Motion, Processing Framework, Data Visualisation, Digital Image Processing

Ein modulares Framework für bildbasierte 3D-Rekonstruktion

Zusammenfassung: In dieser Arbeit befasse ich mich mit dem Thema der 3D-Rekonstruktion und Struktur aus Bewegung. Im ersten Teil der Arbeit analysiere ich die aktuellen 3D-Rekonstruktionsalgorithmen aus der Literatur, um ein Framework zu entwickeln, welches die Arbeit mit solchen Algorithmen vereinfacht. Das Framework ermöglicht existierende Implementierungen wiederzuverwenden und auch Teile bestehender Algorithmen durch eigene Implementierungen zu ersetzen. Das Datenmodell ist darauf ausgelegt Zwischenergebnisse zu visualisieren, und den Datenfluss flexibel zu variieren, was die Arbeit bei der Entwicklung von Algorithmen erleichtert.

Basierend auf den Grundlagen der 3D-Rekonstruktion und Evaluierung bestehender Implementierungen, wird das Framework mit dem Ziel einer modularen und flexiblen Gestaltung der Prozesskette entwickelt.

Ich zeige außerdem, wie das Framework auf bestehende Problemstellungen angewendet werden kann und wie es die schnelle Lösungsentwicklung unterstützt. Dieses erfolgt, indem das Framework dem Anwender erlaubt, sich auf die Implementierungsdetails des Algorithmus zu konzentrieren, anstatt den gesamten Prozessablauf betrachten zu müssen.

Die Implementierung des Framework ist als Open-Source-Software veröffentlicht.

Schlüsselwörter: 3D-Rekonstruktion, Struktur aus Bewegung, Prozess Framework, Daten Visualisierung, Digitale Bildverarbeitung