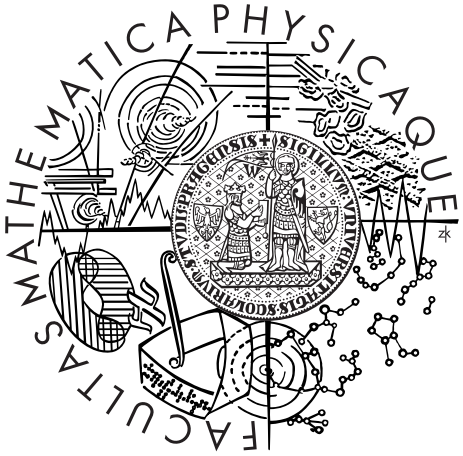
Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS



Adam Dominec

3D Surface Reconstruction from Video Sequences

Computer Science Institute of Charles University

Supervisor of the bachelor thesis: Mgr. Lukáš Mach

Study programme: Computer Science

Specialization: General Computer Science

Prague 2013

I dedicate this work to the community of Open Source Software, who gave me the possibility and enthusiasm to study computer graphics.

I am thankful in particular to my thesis supervisor Lukáš Mach for his comments, suggestions, and patience.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: 3D Surface Reconstruction from Video Sequences

Autor: Adam Dominec

Ústav: Informatický ústav Univerzity Karlovy

Vedoucí bakalářské práce: Mgr. Lukáš Mach, Informatický ústav Univerzity Karlovy

Abstrakt: Tato práce se věnuje metodě pro podrobný odhad natáčené scény (*dense scene reconstruction*) z videa, kdy interní i externí kalibraci kamery považujeme za předem známou. Jedná se o metodu modulární, sestávající z mnoha převážně nezávislých součástí. Popisujeme zde tedy všechny algoritmy užité pro řešení jednotlivých podproblémů: známé algoritmy pro řešení klasických úloh a nově navrhované postupy, kde jich je potřeba. Zde popsaná metoda výpočtu scény je iterativní a je možné ji přizpůsobit požadované přesnosti a rozlišení. Práce je doprovázena kompletní implementací popsaného algoritmu s otevřeným zdrojovým kódem.

Klíčová slova: structure from motion, hustá korespondence, počítačové vidění, optický tok

Title: 3D Surface Reconstruction from Video Sequences

Author: Adam Dominec

Department: Computer Science Institute of Charles University

Supervisor: Mgr. Lukáš Mach, Computer Science Institute of Charles University

Abstract: This work describes a method for dense scene reconstruction from video, assuming both the external and internal calibration of camera in each frame is known. The method is modular; in the cases of the well studied subproblems, a description of the corresponding algorithms is provided, and where necessary, we present our novel techniques. The method reconstructs the scene in an iterative manner and is highly adaptable to required precision and resolution of the output. This work is accompanied by a complete open-source implementation of the method described.

Keywords: structure from motion, dense matching, computer vision, optical flow

Contents

1 Introduction **6**
- 1.1 Overview of our method . . . 7
- 1.2 Related work . . . 8
- 1.3 Available software . . . 9
- 1.4 Organization . . . 10
- 1.5 Notation . . . 10
- 1.6 Basic notions . . . 11

2 Algorithms **12**
- 2.1 Polygonization of the point cloud . . . 12
  - 2.1.1 Alpha shapes . . . 12
  - 2.1.2 Poisson volume reconstruction . . . 14
  - 2.1.3 Marching cubes . . . 16
- 2.2 Dense optical flow . . . 17
  - 2.2.1 Linear approximation . . . 17
  - 2.2.2 Quadratic polynomial expansion . . . 19
- 2.3 Point triangulation . . . 21
- 2.4 Normal estimation . . . 23

3 Heuristics **24**
- 3.1 Camera selection . . . 24
- 3.2 Reprojection . . . 28
- 3.3 Estimating the optical flow error . . . 29
- 3.4 Point cloud filtering . . . 31

4 Implementation **33**
- 4.1 Polygonization methods . . . 35
- 4.2 Video corrections . . . 37
- 4.3 Optical flow algorithms . . . 38
- 4.4 Application of mesh rasterization . . . 38
- 4.5 User-defined parameters . . . 39
- 4.6 Adopted libraries and practical remarks . . . 39

5 Evaluation **41**
- 5.1 Error of the optical flow . . . 41
- 5.2 Testing sequences . . . 42
  - 5.2.1 Perlin sphere . . . 42
  - 5.2.2 Stone relief . . . 42
  - 5.2.3 Glossy car . . . 43

6 Conclusion **45**
- 6.1 Future work . . . 45

Bibliography **47**

Attachments **49**
A Common types of video degradation 49
A.1 Colorimetric issues 49
A.2 Optical distortions 49
A.3 Lossy video formats 50
B Sparse scene reconstruction 51
B.1 Overview 51
B.2 Feature tracking 52
B.3 Feature matching 52
B.4 Camera calibration 53
C Directory structure of the attached data 54

1. Introduction

This thesis describes an algorithm for automatic dense 3D reconstruction from a calibrated video sequence. In contrast to the sparse reconstruction approach, our goal is to utilize as much information from the input sequence as possible to obtain a highly detailed 3D model of the scene. The output scene is represented in an easily processable form of a triangle mesh. Since we are dealing with calibrated video sequences, we assume that both the internal camera parameters (e.g., focal lengths) and the external ones (camera positions and orientations) are known for each frame of the sequence. Obtaining such calibration data from the video sequence alone is highly non-trivial, however, it is a well researched problem both in theory and practice [15]. A handful of packages solving this task are available to the end user, both in the form of free (e.g., Bundler [1], PMVS2 [2], Blender [3]) and proprietary software (e.g., Adobe After Effects [4], Maxon Cinema 4D [5], Andersson Technologies SynthEyes [6]). The algorithm presented here can hopefully provide a useful addition to this tool-chain, enabling significantly more detailed 3D reconstructions.

The main expected application of this work is for visual effects in movies. A detailed geometrical representation of a scene enables us to realistically incorporate computer generated objects into the video, modify lighting of the scene as well as mask out unwanted features.

Dense reconstruction can also be used as a quick and inexpensive 3D scanner for objects of arbitrary scale. After some postprocessing, the resulting scene can be sent to a 3D printer to obtain a scale model. Alternatively, the acquired data can be archived for later visualization, which is desirable, e.g., for documenting historical artifacts.

Further applications can lie in the field of intelligent robotics. Here, a scene reconstruction is necessary for autonomous navigation of a vehicle. It may even be the case that the robot (typically an aerial vehicle) is used for terrain mapping, leading to further qualitative requirements on the reconstruction.

Modern videogames also increasingly apply some form of scene reconstruction for user input. Typically, only simple algorithms are used in this setting because of its real-time nature. However, current research shows that even high quality dense reconstruction algorithms can be optimized to provide such performance.

Our algorithm is designed for mostly static scenes consisting of smooth surfaces with static lighting and negligible amount of reflections. Of course if the input sequence is a part of a movie, it is reasonable to expect moving objects, such as actors. Nevertheless, we do not account for these explicitly and hope the artifacts resulting from large-scale motion will be discarded during the outlier filtering phases of the reconstruction.

1.1 Overview of our method

The key idea of our approach is to use the current estimate of the scene to predict individual frames of the video. Comparing these predictions to the actual frames of the sequence allows us to update the estimate of the reconstructed scene. By repeating this process, we iteratively and adaptively refine the estimate of the scene until the reconstructed scene has a sufficient amount of detail. Both internal and external camera calibration must be known in advance; we do not try to compensate for their errors in any way.

To start, the algorithm needs an initial estimate of the scene. This is usually generated as a side product of camera calibration in the form of a set of points that lie on the real scene's surface. For more information about sparse scene reconstruction, please refer to Appendix B. This point cloud is converted to a mesh surface approximating the real scene.

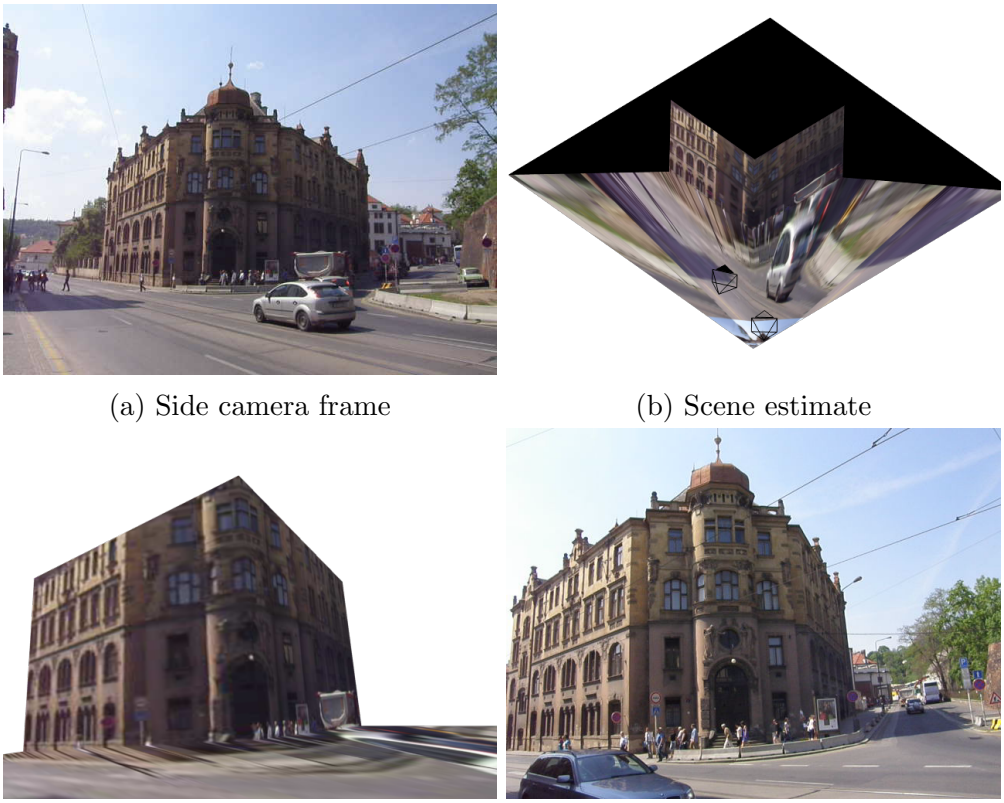
A randomized heuristic then chooses several frames of the video based on the viewpoints they were acquired from, independently of their temporal position in the sequence. One of these viewpoints is labeled as the *main camera,* others are *side cameras.*

Sequentially, the frame of each of the side cameras is projected onto the scene's surface and then rendered from the main camera. This is the predicted frame corresponding to the side camera. If our scene estimate was perfect, this prediction should be equal to the main camera's frame (except for occlusions).

This process is illustrated in Figure 1.1. The first image is a frame of the sequence corresponding to a side camera. Figure 1.1b shows an overall view of the scene estimate with this image projected. Figure 1.1c shows the view of the scene as seen from the main camera; this view is the predicted frame corresponding to the side camera considered. It is roughly similar to the actual frame corresponding to the main camera (Fig. 1.1d).

We then calculate dense optical flow between the main camera's image and each of the predicted frames. This flow reflects deficiencies in the current scene estimate with zero flow indicating the surface is correct. Using all the optical flow fields at once, we calculate the deformation of the scene surface that would minimize these optical flows. This updated surface is produced as a point cloud, allowing us to handle discontinuities and outliers effortlessly. Each pixel of the collection of optical flow fields gives rise to one point in the point cloud.

The process of choosing a main camera along with a bundle of side cameras is repeated to densely cover the scene. Then, the resulting point cloud is filtered to remove outliers and to reduce redundancy. It is again converted to a mesh surface to provide a new scene estimate. Finally, a heuristic determines if the amount of detail is sufficient and if not, we repeat the whole scene updating process.



(a) Side camera frame (b) Scene estimate

(c) Projection from the main camera (d) Main camera frame

Figure 1.1: Example of reprojection on a manually created model of the district Prosecuting Attorney's Office, Prague.

1.2 Related work

Structure from motion seems to have become a very fashionable topic over the last few years. There are many scientific papers and software packages that solve various tasks related to scene reconstruction, and the amount is still rapidly increasing.

This work uses an approach similar to an independent work by Newcombe and Davison [13] from 2010. However, their framework is optimized for real-time reconstructions. It processes data streaming from the camera, calibrating its pose and reconstructing a sparse point cloud on the fly. Their framework works with mesh data directly, creating a new rectangle-like patch from the camera's view when its viewport significantly changes. The iteration scheme to refine previously estimated geometry is used there as well, but only within the single patch being processed. To polygonize the point cloud (sparse or dense), they choose a technique based on compactly supported radial basis functions, for the reasons of computational efficiency. In areas where the newly created patch closely overlaps with previous data, the patch is simply filtered out.

Another dense reconstruction algorithm based on mesh data was proposed by Martin Bujňák [16]. It is designed for offline processing of uncalibrated video

sequences. The camera selection stage is not mentioned in much detail in the work; presumably the cameras are selected based on their position in the sequence. For merging meshes obtained from different views, the author proposes a sophisticated scheme motivated by practical scenarios.

Representing the data as mesh structure is very efficient for visualization. Unfortunately, it causes difficulties when dealing with discontinuities in the depth map.

A distributed algorithm based on volumetric information was proposed by Wendel et al. [33]. They use an aerial drone with a mounted camera and enough computational power to estimate its pose using sparse reconstruction techniques. The drone selects interesting frames out of its real-time sequence, based on the positions they were obtained from, and transmits these to a static computer, along with their calibration data. The server calculates depth information of the viewed geometry and inserts these values into a volumetric structure in the form of the signed distance function.[1] This volumetric data is directly visualized by raycasting.

Newcombe, Davison and Lovegrove presented [14] an algorithm based on volumetric representation, too. They managed to omit the sparse tracking phase and use dense volumetric data directly for camera pose estimation. The frames used are always successive in the sequence, as the approach relies on high precision calculations from images with small parallax.

Volumetric representation is very flexible and rather straightforward to process. Conversion of voxel data to mesh is not difficult; an algorithm solving this task is described in Section 2.1.3. The drawback of volumetric representation is, in most implementations, that the bounding volume must be a priori known.

An offline algorithm producing a point cloud only was proposed by Parsonage et al. [30]. They start by calibrating cameras for the whole sequence, obtaining a sparse point cloud as a side product, and then calculate and combine dense point clouds obtained from different views. Of interest is their camera selection algorithm. It considers both the temporal and spatial positions of the cameras, and their view of the point cloud currently known. Furthermore, the camera set is checked during the reconstruction to see whether its data corresponds to the current estimate; otherwise, the set is discarded.

1.3 Available software

Perhaps the best known software for sparse scene reconstruction is Bundler [1] written by Noah Snavely. Its principle of operation is described in a paper [31]

[1] The *signed distance function* has positive values outside of an object and negative inside of it; absolute value of the function is the distance from the nearest point on the object surface.

by the same author. In a related project called Photo Tourism [32], the geometric representation of entire cities was automatically reconstructed using photos collected from the Web. It gave rise to a proprietary package called PhotoSynth [10] by Microsoft.

A graphical interface for Bundler and related packages, called PMVS2, has been developed by Yasutaka Furukawa and Jean Ponce [2]. It is rather difficult to install and use, but it can solve most tasks related to structure from motion.

A C++ library for sparse reconstruction and a few related tasks was written by Keir Mierle under the name libmv [11] as his dissertation. Its main application is in the 3D package Blender [3], where it has been incorporated into the workflow.

Apart from these, there are many more structure from motion packages. Mainly, there are many commercial software solutions available, which are not listed in this section. Many universities have been developing their own implementations of structure from motion algorithms; however, few of these are comfortable to use. There are even more such programs being developed by individuals (such as the program accompanying this thesis), many of which rely on the OpenCV library [7] for their computations.

1.4 Organization

Chapter 2 gives a detailed description of widely known algorithms we utilized as components of our method. These are grouped by the task for which they are used in our program, and the tasks are ordered chronologically by their presence in our workflow. The next chapter describes novel algorithms applied in this work. Many aspects of their design are rather a matter of choice, so we concentrate on the motivation behind these. Chapter 4 describes the implementational aspects of our program in detail. It also provides a more detailed description of the overall method, as it has been outlined in Section 1.1. The last chapter evaluates the algorithm on both standard testing sequences and real world samples, and comments on its strengths and weaknesses. In that chapter we also discuss appropriate choices of parameters for these scenarios.

The directory structure of the accompanying data medium is explained in Attachment C.

1.5 Notation

We mostly follow the notation used in the classical book Multiple View Geometry [22]. Boldface letters (e.g., $\mathbf{x}$) denote real column vectors. Monospace letters (e.g., $\mathtt{C}$) denote real matrices. An upper index next to these (e.g., $\mathtt{C}^i$) identifies a matrix in an ordered set of matrices, n-th power of matrix would be always expressed

using parentheses, e.g., $(\mathtt{S})^n$. The central dot $\cdot$ can mean either scalar or matrix multiplication.

We often use the central circle $\circ$ to avoid multiple parentheses, in the meaning $f \circ g(x) = f\big(g(x)\big)$. We use a nonstandard upper index $^+$ of certain functions which means their parameter is added to their actual value: $f^+(x) = x + f(x)$.

Classical norm braces $|\cdot|$ denote the Euclidean norm of vectors (or the magnitude of a set). Doubled version of these, $\|\cdot\|$, denotes various other norms and norm-like mappings and is further clarified where used.

1.6 Basic notions

Grayscale image is a function $\Omega \to \mathbb{R}$, where $\Omega \subset \mathbb{R}^2$ is a rectangular image domain. Unless stated otherwise, we assume $\Omega = [-1, 1]^2$.

The real world is considered to be $\mathbb{R}^3$; however, we often use homogeneous coordinates $\mathbb{P}^3$, expressing 3D vector $(x, y, z)^\mathsf{T}$ as $(xw, yw, zw, w)^\mathsf{T}$ for an arbitrary $w \neq 0$. Using this ambiguous notation, projective mappings in $\mathbb{R}^3$ can be expressed as linear ones.

Camera projection is a non-singular 4×4 matrix. Multiplication by it transforms points from *scene space* to *camera space*, both being $\mathbb{P}^3$. Note the contrast to the definition more common in literature, by which the camera would have only three rows. The extra row of our camera can be viewed as obtaining a depth map value for each projected pixel. The visible part of camera space (usually referred to as the *clip space*) is considered to be a box of homogeneous vectors corresponding to $(x, y, z)^\mathsf{T} \in [-1, 1]^3$ (i.e., in the OpenGL convention). With increasing distance of points from the camera center in scene space, the projected camera-space depth z increases (nonlinearly); points with $z = -1$ and $z = 1$ are said to lie on the *near clipping* and *far clipping* planes, respectively. A function π to transform visible camera-space points further to image space Ω is defined when first needed, in Section 2.3.

Although the entire video sequence was necessarily acquired by a single moving camera, we use the intuition that there is a static camera corresponding to each frame, with all these cameras viewing the scene simultaneously.

2. Algorithms

The approach presented here utilizes many algorithms from different areas. In this chapter, we describe these algorithms in their general setting. Details about their specific use in our program are presented in Chapter 4.

2.1 Polygonization of the point cloud

2.1.1 Alpha shapes

Alpha shapes were introduced by Edelsbrunner, Kirkpatrick and Seidel [18] in 1983. An algorithm solving the alpha shape problem connects vertices of a 3D point cloud into triangle faces in a way that respects, in a certain sense, its overall shape. The input and output of the problem (for a general dimension $d \in \mathbb{N}$) has the following general form:

Name:	the **alpha shape** problem.
Input:	a set X of n points from $\mathbb{R}^d$, a radius $\alpha \in \mathbb{R}_0^+$.
Output:	a set of $(d-1)$-dimensional simplicies (e.g., a set of triangle faces for $d = 3$), whose vertices are from X.

We are mainly interested in the case $d = 3$. A triangle is included in the output if and only if its vertices are touched by a sphere of radius α whose interior does not contain any points of X. As α approaches infinity, the output degenerates to a convex hull of the input point cloud.[1] Conversely, if the parameter approaches zero, the output will be empty. A good choice of the radius parameter is very important in order to get useful results but — as we will see — the algorithm can be adapted to make a guess for a suitable α on its own.

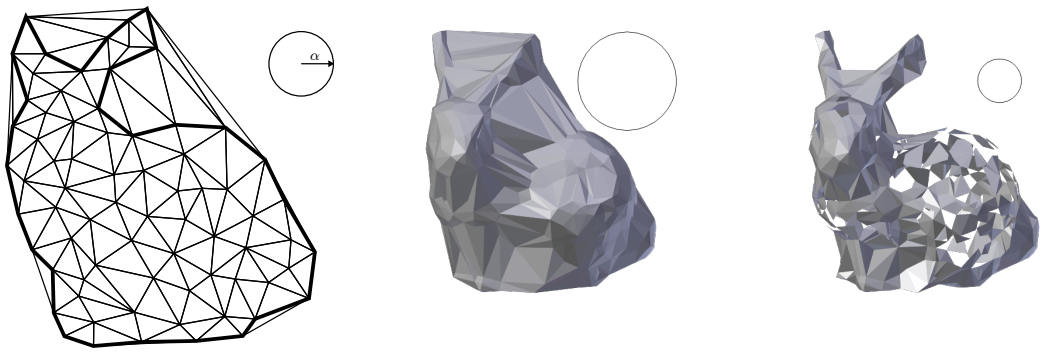
It helps to start by calculating a more complex structure: the Delaunay tessellation,[2] which can be later easily turned into any specific α-shape.

Definition 1. *Delaunay tessellation of a set of points from $\mathbb{R}^d$ is a set of d-dimensional simplices whose vertices are a subset of the input and whose circumscribing spheres do not contain any other points from the input.*

A Delaunay tessellation of a set of 72 points in $\mathbb{R}^2$ is shown in Figure 2.1a. The α-shape for a manually chosen α is marked as a thick outline. Figures 2.1b

[1] Sometimes, the radius is set to $\frac{1}{\alpha}$, which allows alpha shapes to generalize the convex hull. Furthermore, the range of permitted values of α can be extended to negative values.

[2] Commonly, it is referred to as the Delaunay triangulation, even when generalized to $d > 2$. We prefer the term *tessellation* because *triangulation* is used in a completely different meaning throughout this text.



(a) Delaunay tessellation and alpha shape in 2D

(b) Alpha shape in 3D, $\alpha = 1$

(c) Alpha shape in 3D, $\alpha = 0.4$

Figure 2.1: Illustration of alpha shapes for various input.

and 2.1c show α-shapes of the Stanford bunny 3D scanned data (1000 points) for two different values of α.

Many algorithms for calculating the Delaunay tessellation are used in practice. When the dimensionality of the problem is at least three, it appears to be practically feasible to use the following incremental approach. Delaunay tessellation of $d+1$ points is the simplex itself. Adding a new point to an existing Delaunay tessellation consists of finding its nearest neighbor in the current structure, determining the affected area around it and appropriately altering the corresponding simplices. All these three tasks can be carried out in linear expected time [27]. The resulting structure contains at most $O(n^{\lceil d/2 \rceil})$ simplices [29, p. 115–119], so in the case of $d = 3$, the output size of this algorithm matches the expected time complexity. Practical measurements performed on a set of points generated uniformly at random indicate that the performance tends to be even better, close to a linear dependency [27].

Conversion of a 3D Delaunay tessellation to an alpha shape is straightforward. Every simplex (tetrahedron) s_i of the Delaunay tessellation has a circumscribed sphere of radius $\alpha_{\max}(s_i)$. Similarly, every triangle face f_i of the tessellation has a circumradius denoted by $\alpha_{\min}(f_i)$. A face f_i is a member of the output for a given α if and only if $\alpha_{\min}(f_i) \leq \alpha \leq \alpha_{\max}(s_i)$ for $s_i \in \{\text{neighbor simplices of } f_i\}$. Note that most faces have two neighbor simplices; those with a single neighbor form the convex hull of the point set.

We observe that as α changes, the resulting α-shape changes only at finitely many thresholds, namely the circumradii of all faces and simplices of the tessellation. Clearly, as α decreases, the number of disconnected components (including all disconnected vertices) monotonically increases. Thus, we can pick a desired number of disconnected components and use bisection to search for the smallest α possible. We generally wish to keep α as low as possible, because such α-shape retains most details of the original point cloud. On the other hand, we cannot

allow it to decrease too much, since the α-shape would then degenerate to an empty set. To prevent that, we force the output to make a single component; but as discussed later in section 4.1, that is a questionable constraint.

2.1.2 Poisson volume reconstruction

A powerful numerical approach to volumetric data reconstruction from surface samples based on solving the Poisson equation has been proposed by Michael Kazhdan et al. [24]. The resulting volumetric representation has higher values inside of the estimated model than outside. Therefore, the surface of the model can be extracted as its isosurface.

Name:	the **volume reconstruction** problem.
Input:	a set of points $\{\mathbf{p}^i\} \subset \mathbb{R}^3$ on the surface of the model and the corresponding normals $\{\mathbf{n}^i\}$.
Output:	estimated characteristic function $\hat{\chi}$ of the model sampled in a regular grid.

The approach is targeted at scenes with a meaningful volumetric representation, i.e., compact solid objects. We begin by introducing the characteristic function:

Definition 2. *The characteristic function* $\chi : \mathbb{R}^3 \to \{0, 1\}$ *of a solid object has value* 1 *for points inside of the object and* 0 *outside.*

The main idea of the algorithm is that if the input data would be exact, then after applying subtle smoothing to the characteristic function χ its gradient at each point $\mathbf{p}^i$ would have the same direction as the corresponding inward normal $-\mathbf{n}^i$. The function χ itself cannot be used since it is not differentiable and thus the gradient is not always properly defined. To overcome this, the function is convolved with a blurring kernel F, obtaining a smooth function $\hat{\chi} = F * \chi$. Furthermore, we can define a continuous vector function $V \colon \mathbb{R}^3 \to \mathbb{R}^3$ based on the input samples that satisfies $V(\mathbf{p}^i) = -\mathbf{n}^i$. We get the following equation:

$$\nabla\hat{\chi} = V. \tag{2.1}$$

This equation does not necessarily have a solution $\nabla\hat{\chi}$ for a general V. Therefore, we seek a solution in the least-squares sense. The vector function V is defined at the end of this section.

Analytically, the sought function $\hat{\chi}$ minimizes the following:

$$E = \int_{\mathbb{R}^3} |\nabla\hat{\chi}(\mathbf{x}) - V(\mathbf{x})|^2 \mathrm{d}\mathbf{x}. \tag{2.2}$$

As a necessary condition for attaining the minimum value, $\hat{\chi}$ must satisfy the Euler-Lagrange equation on its whole domain. In order to apply this formalism, the functional L to be minimized is formulated as

$$L\big(\mathbf{x}, \hat{\chi}(\mathbf{x}), \nabla\hat{\chi}(\mathbf{x})\big) := |\nabla\hat{\chi}(\mathbf{x}) - V(\mathbf{x})|^2. \tag{2.3}$$

The parameter $\hat{\chi}(\mathbf{x})$ is just a technicality, it is not used in L.

Then, using the notation $\mathbf{x} = (x, y, z)^\mathsf{T}$ and $\nabla\hat{\chi}(\mathbf{x}) = (f, g, h)^\mathsf{T}$, the Euler-Lagrange equation is of the form

$$\frac{\partial L}{\partial\hat{\chi}(\mathbf{x})} - \frac{\mathrm{d}}{\mathrm{d}x}\frac{\partial L}{\partial f} - \frac{\mathrm{d}}{\mathrm{d}y}\frac{\partial L}{\partial g} - \frac{\mathrm{d}}{\mathrm{d}z}\frac{\partial L}{\partial h} = 0. \tag{2.4}$$

Simplifying and using the fact $\hat{\chi}(\mathbf{x})$ does not occur in L, we obtain

$$0 - \mathrm{div}(L) = \mathrm{div}(\nabla\hat{\chi}) - \mathrm{div}(V) = \Delta\hat{\chi} - \mathrm{div}(V) = 0. \tag{2.5}$$

An intuitive explanation is that we equate the curl-free parts of the vector fields from (2.1). This is reasonable since the gradient of any function is always curl-free. The result is known as the *Poisson equation* and typically formulated as $\Delta f = g$ for an unknown function $f \in \mathbb{R}^n$. Solving problems of this kind has a wide variety of scientific applications, many of them in computer graphics.

To tackle the problem numerically, we need to design a suitable finite basis and express the function $\mathrm{div}(V)$ (corresponding to the input data) as a vector in the resulting vector space. The solution $\hat{\chi}$ will be another such vector.

The basis proposed by the authors is a hierarchical structure of functions resembling wavelets. Each of the basis functions F_i is a transposed and scaled B-spline of order 3, i.e., 3rd convolution of the three-dimensional box function with itself. The hierarchy is given by an octree:[3] for each of its nodes the basis contains a function centered at the center of the corresponding cell, scaled according to the width of the cell and normalized to have a unit integral. The octree spans the whole input point cloud and its depth is chosen so that each input point $\mathbf{p}^i$ is contained in its own leaf node. Empty nodes are omitted where possible and therefore the size of the structure stays roughly proportional to the amount of detail of the sampled surface.

The vector function V is defined in its continuous form — but anyway using the octree structure. Its value at any point in space is a weighted average of the sampled normals in the corresponding cell and its neighborhood. The weights correspond to trilinear interpolation among the cells. In cases when the samples are not evenly distributed, a further weighting factor inversely proportional to the

[3] An *octree* is a tree of cubic cells, where each of its inner nodes is equally subdivided among its eight children.

number of points in the neighborhood is introduced to obtain accurate results. And finally, weight of each point may be part of the input, if the precision they were obtained with substantially varies for each point.

Expressing the functions $\mathrm{div}(V)$ and $\Delta\hat{\chi}$ in the basis of functions F_i, this problem becomes a discrete one. The authors suggest to discretize both functions by projection onto the basis, remarking that it is just an approximation, since the basis is not orthogonal. The former, $\mathrm{div}(V)$, becomes a constant vector $\mathbf{v}$, whereas the latter can be expressed as $\mathtt{A}\mathbf{x}$ for a constant matrix $\mathtt{A}$, due to the linearity of the Laplacian Δ. Vector $\mathbf{x}$ corresponds to the sought function $\hat{\chi}$. The linear system $\mathtt{A}\mathbf{x} = \mathbf{v}$ is solved using a multigrid iterative approach to leverage the hierarchical structure of the data and sparsity of $\mathtt{A}$.

2.1.3 Marching cubes

An algorithm for finding the isosurface of voxel data was proposed by Lorensen and Cline [28] in 1987. Here, we describe its regular variant but it can be extended to process grids with spatially varying resolution. The result of the Poisson reconstruction of Section 2.1.2 can be easily used as the input of this algorithm.

Name:	the **isosurface extraction** problem.
Input:	object density function $\hat{\chi}$ sampled at a regular grid; thresholding value t.
Output:	triangle mesh representing the isosurface $\hat{\chi} = t$.

A good choice of the threshold t is a (possibly weighted) average of $\hat{\chi}$ at the sampled surface points used previously for volumetric reconstruction. The amount of detail of the resulting polygonal approximation depends on the density of the grid.

The algorithm processes each cube of the sampling grid separately. If the value of $\hat{\chi}$ is above the threshold t at several vertices of a cube and below it at others, a patch of mesh surface is inserted inside the cube. Positions of vertices of the patch are given by linear interpolation of $\hat{\chi}$ along edges of the cube (i.e., line segments of the sampling grid). The edges and faces of the surface patch are defined for each possible configuration by a lookup table. This table is designed so that two patches in neighboring cubes always have identical intersections with their common face — ensuring that the resulting surface is a 2-manifold (except for boundaries, possibly).

2.2 Dense optical flow

We assume the scene being reconstructed is Lambertian, meaning that the color of each visible scene point does not depend on the direction it is viewed from. If the viewpoints of the two video frames being compared are close to each other, we can ignore occlusions and assume that each pixel of the first image has its modified position in the second image with the same color. The difference in position leads to a 2-dimensional vector field for the whole image. This displacement field is called the *dense optical flow*, or optical flow for short. We would like to estimate it to obtain information about the scene geometry. To simplify the problem, we consider only grayscale images.

Name:	the **dense optical flow** problem.
Input:	two grayscale images $I, P : \Omega \to \mathbb{R}$
Output:	displacement field $\text{flow}(\mathbf{x})$ such that $I(\mathbf{x}) = P(\mathbf{x} + \text{flow}(\mathbf{x}))$

In this section, we consider the rectangular image domain Ω has actual pixel dimensions of the images. It helps to simplify the notation.

2.2.1 Linear approximation

An optical flow algorithm based on local linear approximation of the image was presented by Horn and Schunck in 1981 [23], and further developed by many researchers. It is similar to the simple version of the popular sparse motion tracker proposed by Kanade, Lucas and Tomasi in the same year [25].

The algorithm iteratively minimizes the energy functional

$$E(\text{flow}) = \|I - P \circ \text{flow}^+\| + \|(\nabla\text{flow}_1, \nabla\text{flow}_2)^\mathsf{T}\| \tag{2.6}$$

for the abbreviation $\text{flow}^+(\mathbf{x}) = \mathbf{x} + \text{flow}(\mathbf{x})$, and flow_i being the components of the vector field. This energy can be well understood under the (nowadays classical) data term + smoothness term paradigm.

The mapping $\|\cdot\|$ does not have to be a true norm and a different one can be applied for each of the two terms. Typically, we let them differ by a multiplicative constant, leaving a free parameter for weighting between smoothness and data term accuracy. Here, we consider $\|f\| = \int_\Omega \Psi(f(\mathbf{x})^2)\mathrm{d}\mathbf{x}$ for an increasing smooth function Ψ. For the smoothness term, we sum squares of the matrix elements: $\|f\| = \int_\Omega \Psi\big(\sum_{i,j} f_{i,j}(\mathbf{x})^2\big)\mathrm{d}\mathbf{x}$. Typical choices of Ψ are discussed at the end of this section.

We assume that both images are locally well approximable by a linear function. Furthermore we assume flow to be smooth, implying that the gradients ∇I and ∇P of the images should be similar. We let G denote their average:

$$G = {}^1\!/_2 \left(\nabla I + \nabla P\right). \tag{2.7}$$

Together, this leads us to the approximation

$$P\big(\mathbf{x} + \mathrm{flow}(\mathbf{x})\big) \approx P(\mathbf{x}) + G \cdot \mathrm{flow}(\mathbf{x}). \tag{2.8}$$

The energy functional E then becomes

$$\begin{aligned} E(\mathrm{flow}) = \int_\Omega \Psi_D \left(\big(I(\mathbf{x}) - P(\mathbf{x}) - G \cdot \mathrm{flow}(\mathbf{x})\big)^2 \right) + \\ \Psi_S \left(|\nabla \mathrm{flow}_1(\mathbf{x})|^2 + |\nabla \mathrm{flow}_2(\mathbf{x})|^2 \right) \mathrm{d}\mathbf{x} =: \int_\Omega L\big(\mathbf{x}, \mathrm{flow}(\mathbf{x}), J_{\mathrm{flow}}(\mathbf{x})\big) \mathrm{d}\mathbf{x}, \end{aligned} \tag{2.9}$$

where $J_{\mathrm{flow}} = (\nabla \mathrm{flow}_1, \nabla \mathrm{flow}_2)^\mathsf{T}$ is the Jacobian matrix.

In order for flow to minimize E, the integrand L must satisfy the 2-dimensional form of Euler-Lagrange equations. To state these, we have to expand the vectors into variables as $\mathbf{x} = (x, y)^\mathsf{T}$, $\mathrm{flow}(\mathbf{x}) = (u, v)^\mathsf{T}$ and $G(\mathbf{x}) = (g, h)^\mathsf{T}$. Further, we define a shorthand $t = P(\mathbf{x}) - I(\mathbf{x})$. Using this notation, the integrand L reads as

$$L(\ldots) = \Psi_D\big(u^2g^2 + v^2h^2 + t^2 + 2ugvh + 2t(ug + vh)\big) + \Psi_S\big(\frac{\partial u}{\partial x}^2 + \frac{\partial v}{\partial x}^2 + \frac{\partial u}{\partial y}^2 + \frac{\partial v}{\partial y}^2\big), \tag{2.10}$$

where all the partial derivatives of u and v are considered to be its parameters.

The Euler-Lagrange equation for the coordinate x is of the form

$$\Psi'_D(\ldots) \cdot (2ug^2 + 2gvh + 2tg) - \Psi'_S(\ldots) \cdot \frac{\mathrm{d}}{\mathrm{d}x}\left(2\frac{\partial u}{\partial x}\right) - \Psi'_S(\ldots) \cdot \frac{\mathrm{d}}{\mathrm{d}y}\left(2\frac{\partial u}{\partial y}\right) = 0, \tag{2.11}$$

where $\Psi'_D(\ldots)$ and $\Psi'_S(\ldots)$ are derivatives of the functions at the original values of their respective arguments.

The equation can be modified to a simpler form, and a similar equation is obtained for the coordinate y:

$$\begin{aligned} \Psi'_D(\ldots) g(ug + vh + t) - \Psi'_S(\ldots)\Delta u = 0 \\ \Psi'_D(\ldots) h(ug + vh + t) - \Psi'_S(\ldots)\Delta v = 0. \end{aligned} \tag{2.12}$$

These two equations shall be satisfied by flow in every pixel of the image. Because of the derivatives Ψ'_D and Ψ'_S, they might be nonlinear in u, v. To overcome this problem, we locally approximate these as constants. Then, we

discretize the gradient of the image and the Laplacian of the optical flow by central finite differences to obtain a sparse linear system. If a value outside of image domain is necessary, we use the nearest pixel. This system can be efficiently solved by an iterative scheme, updating $\Psi'_D(\ldots)$ and $\Psi'_S(\ldots)$ to their current value after each iteration.

However, the optical flow is usually too large and the linear approximation of the images is not sufficient. We can extend the scheme to make use of an estimated flow field flow_{old}, and refine it by a new field flow_{new}. This requires calculating an optical flow between the first image I and a *warped* version of the second image $P \circ \text{flow}^+_{\text{old}}$. The overall result of the computation is then $\text{flow} = \text{flow}_{\text{old}} \circ \text{flow}^+_{\text{new}}$.[4] The data term stays intact but the argument of Ψ_S must be calculated from $\text{flow}_{\text{old}} \circ \text{flow}^+_{\text{new}}$ to impose smoothness of the overall result.

This method of calculating a refined flow from an estimate is usually referred to as *warping*. The algorithm can be made to handle large flow fields if this estimate is obtained from downsampled versions of the input images — thanks to the fact that the magnitude of the flow field decreases with decreasing image size.

The resulting algorithm starts by downsampling both the images by a certain factor, typically 2, and calling itself recursively (unless the images are below a certain size limit, to stop the recursion). It upsamples the flow field obtained from the recursive call and multiplies it by the same scaling factor. Then, it refines this field by solving the system as described and by applying warping several times on the full-scale images. It outputs the resulting optical flow.

The functions Ψ can be set to a multiple of the identity function, in which case $\Psi'(\ldots)$ is actually a constant. However, it penalizes discontinuities in the flow field too strongly, as they can be actually present in the reality. One of possible ways around this is to use the Huber function: $\Psi(x) = {}^{x}\!/_{2}$ for $x < \varepsilon^2, \Psi(x) = \varepsilon(\sqrt{x} - {}^{\varepsilon}\!/_{2})$ otherwise. This Ψ and its derivative are continuous on $\mathbb{R}^+_0$.

A more detailed derivation of the algorithm presented here is given by Brox et al. [19].

2.2.2 Quadratic polynomial expansion

A method of optical flow calculation based on local polynomial expansion was introduced and researched by Gunnar Farnebäck [20]. Each pixel of both images is assigned a quadratic polynomial which best-fits its neighborhood in that image. Such model obviously provides more information than the linear approximation of Section 2.2.1. Due to this, a more sophisticated model can be used to locally

[4] By the assumption of flow_{old} being locally well approximable as a constant, the result can be simplified as $\text{flow} = \text{flow}_{\text{old}} + \text{flow}_{\text{new}}$.

approximate the optical flow as well. This in turn makes the resulting flow robust under a wider range of transformations, e.g., rotation.

Definition 3. *The (quadratic) polynomial expansion of image I around a pixel $\mathbf{x}^i$ is a polynomial $f_i(\mathbf{x}) = \mathbf{x}^\mathsf{T}\mathtt{A}^i\mathbf{x} + \mathbf{x}^\mathsf{T}\mathbf{b}^i + c_i$ that makes a least-squares fit to a chosen neighborhood of the pixel. Namely, the parameters $\mathtt{A}^i, \mathbf{b}^i, c_i$ minimize $\sum_\mathbf{x} w(\mathbf{x} - \mathbf{x}^i)\left|f_i(\mathbf{x}) - I(\mathbf{x})\right|^2$ for a windowing function w.*

The algorithm starts by calculating the quadratic polynomial expansion for each pixel, producing $\{f_i\}$ and $\{f'_i\}$ for I and P, respectively. That can be done efficiently using a coarse-to-fine scheme; the technique is described in more detail in [21].

The key idea of the algorithm is to assume that each pixel's polynomial is transformed by a translation on its own. We denote this translation as $\mathbf{d}^i$, so that $f_i(\mathbf{x}^i) = f'_i(\mathbf{x}^i + \mathbf{d}^i)$ should hold for each pixel $\mathbf{x}^i$. Equating the quadratic coefficients of these polynomials results in $\mathtt{A}^i = \mathtt{A}'^i$. This equation will usually not be satisfied, but leads us to an approximation $\hat{\mathtt{A}} = (\mathtt{A} + \mathtt{A}')/2$. Equating their linear coefficients results in $\mathbf{b}^i = \mathbf{b}'^i + 2\mathbf{d}^i\hat{\mathtt{A}}^i$, which can be used to solve for $\mathbf{d}^i$.

Instead of using this equation directly, a broader neighborhood of $\mathbf{x}^i$ is considered again and a local motion model $s_i(\mathbf{x})$ is fitted to it. For a pixel $\mathbf{x}^k$ in the neighborhood it should hold that $\mathbf{b}^k = \mathbf{b}'^k + 2s_i(\mathbf{x}^k)\hat{\mathtt{A}}^k$. We make a least squares fit for s_i, so formally minimize

$$\sum_k w(\mathbf{x}^k - \mathbf{x}^i)\left|2s_i(\mathbf{x}^k)\hat{\mathtt{A}}^k + \mathbf{b}'^k - \mathbf{b}^k\right|^2. \tag{2.13}$$

The windowing function w may be chosen differently than in the Definition 3 – typically it is much wider. Such a choice allows to use quite complex local model s_i:

$$s_i(\mathbf{x}) = \mathtt{S}(\mathbf{x})\mathbf{p}^i,$$

where

$$\mathtt{S}(\mathbf{x}) = \mathtt{S}\begin{pmatrix}x\\y\end{pmatrix} = \begin{pmatrix}1 & x & y & 0 & 0 & 0 & x^2 & xy\\ 0 & 0 & 0 & 1 & x & y & xy & y^2\end{pmatrix}$$

and $\mathbf{p}^i \in \mathbb{R}^8$ is a vector of the sought distortion parameters for each pixel. Plugging this into (2.13) produces a formula linear in $\mathbf{p}^i$, i.e., one linear least-squares system per image pixel.

To finish, we set $\text{flow}(\mathbf{x}^i) \leftarrow s_i(\mathbf{x}^i)$. And as before, this whole algorithm is called recursively in a warping scheme over different scale levels, and iterated several times on each level to get a more accurate result. To speed up the calculation, warping during iterations on the same level is done by directly sampling the corresponding polynomial f'_j from $j \sim \lfloor \mathbf{x}^i + \text{flow}(\mathbf{x}^i)\rfloor$.

2.3 Point triangulation

When observing a point from two cameras whose external and internal calibrations are known, it is possible to calculate the 3D position of the point in scene from the image-space positions of the two projections (assuming the positions of the two cameras differ non-trivially). In this section, we describe the method employed to perform this under the assumption that the coordinates of the point's projection from one camera (the 'main' one) are known accurately and the projections from all the other ('side') cameras are corrupted by normally distributed (Gaussian) noise.

Name:	the **point triangulation** problem.
Input:	main camera $\mathtt{C}^{\text{main}}$ and image-plane coordinates $(x, y)^\mathsf{T}$; several side cameras $\{\mathtt{C}^i\}$, estimates $\{\mathbf{s}^i\} \subset \mathbb{R}^2$ of the projection of the unknown 3D point, and the corresponding covariance matrices $\{\Sigma^i\}$.
Output:	maximum likelihood estimate $\mathbf{p} \in \mathbb{R}^3$ constrained along the back-projection ray from the main camera; the resulting probability.

In this and some of the following sections, we will need to formalize the conversion from homogeneous to the Cartesian coordinates:

Definition 4. *The projective division function $\pi : \mathbb{P}^3 \to \mathbb{R}^2$ maps a homogeneous 4-vector to its Cartesian counterpart and discards its depth information. That is, $\pi((x, y, z, w)^\mathsf{T}) = \frac{1}{w}(x, y)^\mathsf{T}$.*

The Maximum Likelihood estimate $\mathbf{p}$ we wish to calculate is constrained along the corresponding ray of the main camera and has only one degree of freedom. We can express this as $\mathbf{p}(z) = (\mathtt{C}^{\text{main}})^{-\mathsf{I}}(\mathbf{m} + z\mathbf{d})$, where $\mathbf{m} = (x, y, 0, 1)^\mathsf{T}$ and $\mathbf{d} = (0, 0, 1, 0)^\mathsf{T}$. Recall that (according to the convention used in this work) values of $z \in [-1, 1]$ give points in the visible camera range along the line through the camera center. Via the choice of this unknown z, we try to maximize the combined probability density of all side cameras' distributions.

Each vector $\mathbf{s}^i$ is formed by the estimated coordinates of the same scene point $\mathbf{p}$ as projected from the respective side camera $\mathtt{C}^i$. In practice, its value is obtained from an optical flow field. It is assumed to be related to the exact value $\pi(\mathtt{C}^i\mathbf{p})$ as $\mathbf{s}^i = \pi(\mathtt{C}^i\mathbf{p}) + \mathbf{n}^i$, where $\mathbf{n}^i$ is a random vector normally distributed with covariance matrix Σ^i. This situation is illustrated in Figure 2.2, with the probability distributions rendered as dashed ellipses and cones.

For a screen-space vector $\mathbf{x}$, the probability density given by a side camera's distribution that it is the exact value is $\exp(-\|\mathbf{x} - \mathbf{s}^i\|^2_{\Sigma^i})$, up to a multiplicative

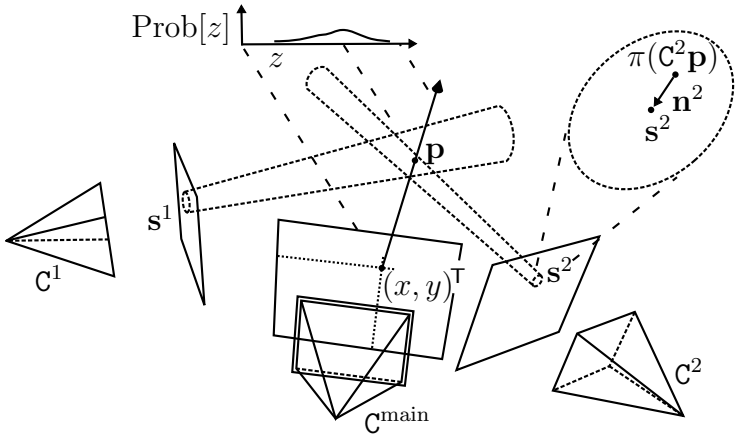


Figure 2.2: Illustration of the triangulation problem with two side cameras.

constant. The mapping $\|\cdot\|_\Sigma$ is the Mahalanobis norm: $\|\mathbf{v}\|_\Sigma^2 = \mathbf{v}^\mathsf{T}(\Sigma)^{-\mathsf{I}}\mathbf{v}$. The combined probability density of all cameras is the product of these exponentials.

Thanks to the fact that $\exp(-x)$ is a strictly decreasing function, maximizing its value equals minimizing x. In our case, the argument x is the sum of Mahalanobis distances in each side camera's projection with respect to its covariance matrix Σ^i. It may be viewed as an energy function to be minimized by moving the triangulated point along its only degree of freedom:

$$E(z) = \sum_i \|\pi(\mathtt{C}^i\mathbf{p}(z)) - \mathbf{s}^i\|_{\Sigma^i}^2 = \sum_i \left(\pi(\mathtt{C}^i\mathbf{p}(z)) - \mathbf{s}^i\right)^\mathsf{T} (\Sigma^i)^{-\mathsf{I}} \left(\pi(\mathtt{C}^i\mathbf{p}(z)) - \mathbf{s}^i\right).$$

Due to the projective division, $E(z)$ is not a polynomial and therefore impractical to minimize explicitly. Instead, we use the Gauss-Newton method. It has a good intuitive motivation, so we describe the process in more detail.

Because the current scene surface $\widehat{\mathcal{S}}$ gives us a good initial estimate for z, we can approximate the camera projections $\pi\left(\mathtt{C}^i(\mathtt{C}^{\text{main}})^{-\mathsf{I}}\mathbf{x}\right)$ as affine mappings $\mathtt{J}^i_{\mathtt{C}^i}$ around $\mathbf{p}(z) = \mathbf{m} + z\mathbf{d}$, by taking partial derivatives. Formally, each $\mathtt{J}_{\mathtt{C}^i}$ is a 2×4 Jacobian matrix, but we use only its third column $\mathtt{J}^i_{\mathtt{C}^i}\mathbf{d}$. This approximation leads to a quadratic function

$$\widehat{E}_z(\zeta) = \sum_i \left(\zeta\mathtt{J}^i_{\mathtt{C}^i}\mathbf{d} + \pi(\mathtt{C}^i\mathbf{p}(z)) - \mathbf{s}^i\right)^\mathsf{T} (\Sigma^i)^{-\mathsf{I}} \left(\zeta\mathtt{J}^i_{\mathtt{C}^i}\mathbf{d} + \pi(\mathtt{C}^i\mathbf{p}(z)) - \mathbf{s}^i\right),$$

which is minimized at $\zeta = -\widehat{E}'_z(0)/\widehat{E}''_z(0)$. We then update the value of z, setting $z \leftarrow z + \zeta$. Because the affine approximation might be too crude, we re-estimate $\widehat{E}_z$ for the new value of z and iterate.

When the minimization converges, we return $\mathbf{p}(z) = (\mathtt{C}^{\text{main}})^{-\mathsf{I}}(\mathbf{m} + z\mathbf{d})$. We also calculate the corresponding probability, i.e., $\exp(-E(z))$.

2.4 Normal estimation

In order to apply the Poisson reconstruction method to a point cloud, the surface normals corresponding to each point need to be estimated. Each triangulated point originates from a pixel in the main camera's frame. We can exploit this fact: assuming that the scene surface is mostly flat, points corresponding to neighboring pixels in the frame should form a mostly flat shape in the scene. A wide enough neighborhood must be considered so that it really resembles a surface to make the estimation sufficiently robust against noise. We can obtain normals for the whole point cloud by repeating the calculation for the neighborhood of each point.

Name:	the **normal estimation** problem.
Input:	a set $\{\mathbf{x}^i\} \subset \mathbb{R}^3$ of noisy points sampled across a flat surface.
Output:	estimated normal $\mathbf{n}$ of the surface.

We view this set of points as samples of a random variable and calculate its 3×3 covariance matrix Σ. This covariance matrix roughly describes the spread of the points in any given direction. If size of the neighborhood is significantly larger than the amount of noise, then the covariance will be the smallest along the surface normal. This direction is exactly the smallest eigenvector of the covariance matrix and can be easily extracted using Singular Value Decomposition. This technique of extracting eigenvectors from covariance matrices of given data is known as the *Principal Component Analysis.*

Note that the result is defined only up to a sign. In practice, we set the orientation so that the normal faces a majority of the cameras from which it is visible.

3. Heuristics

This chapter presents the novel algorithms applied by our method. They are all heuristic algorithms. The reason for this is partially that the tasks being solved are not rigorously defined from theoretical perspective. Rather, some mathematical interpretation of these tasks is first necessary to turn them into proper algorithmic problems. Furthermore, due to an inherent ambiguity in these problems, any algorithm solving these can be easily forced to fail under artificial conditions.

3.1 Camera selection

The scene updating process, as outlined in Section 1.1, compares several frames corresponding to *side* cameras to a single frame corresponding to a camera denoted as the *main* one. The effect of each such substep is local, limited to the scene structure visible to the main camera, and the accuracy and amount of information obtained depends on the spatial configuration of the cameras. The computational costs are mostly linear in the number of cameras used, dominated by calculating the optical flow.

In this section we design an algorithm to choose the cameras for all such update steps to be performed during one iteration of the main algorithm — that is, to choose several main cameras and for each of these one or more side cameras. The result of scene reconstruction is defined only up to Euclidean transformations, so this selection heuristic should be invariant under these. The camera motion during the acquisition might have been completely arbitrary (specifically, the camera might have stalled for a long time), so the heuristic should not rely on the order of frames in the sequence. Robustness against such irregularities may be formulated as the selection heuristic being invariant under permutation of frames in the sequence. Finally, it would be pleasant for the number of cameras chosen to grow sublinearly if the sequence is extended only by frames that do not provide any new visual information about the scene.

We begin by presenting a formal derivation of a heuristic scheme to choose the camera set. The scheme is subsequently modified to provide better efficiency.

Name:	the **camera selection** problem.
Input:	estimated mesh surface $\mathcal{S}$; set of cameras $\mathtt{C}^i$.
Output:	set C of one or more *reprojection pairs*, each of which contains one *main* one *side* camera.

Formally, the problem can be stated as maximization of an implicit fitness function:

$$F(C) = \frac{1}{|\mathcal{S}|}\int_{\mathcal{S}} w(\mathbf{p}, \mathcal{S}, C)\mathrm{d}\mathbf{p} - \alpha|C|, \tag{3.1}$$

where $w(\mathbf{p}, \mathcal{S}, C)$ is supposed to estimate how accurately is the surface point $\mathbf{p}$ imaged by the given set C of camera pairs, and α allows to weigh between precision of the result and speed.

The key decision is how to define w. We constrain it to the form

$$w(\mathbf{p}, \mathcal{S}, C) = \sum_{(\mathtt{C}^{\mathrm{main}}, \mathtt{C}^{\mathrm{side}}) \in C} w_s(\mathbf{p}, \mathcal{S}, \mathtt{C}^{\mathrm{main}}, \mathtt{C}^{\mathrm{side}}) \tag{3.2}$$

for some function w_s. Separating the cameras' influence in this way may bring in significant inaccuracy as cameras with similar viewpoints are highly correlated sources of information. To some extent, we try to improve on that later.

However, the problem is now much easier to handle because we can describe the improvement gained by adding a camera pair $\mathtt{C}^k, \mathtt{C}^l$:

$$F(C \cup (\mathtt{C}^k, \mathtt{C}^l)) - F(C) = \frac{1}{|\mathcal{S}|}\int_{\mathcal{S}} w_s(\mathbf{p}, \mathcal{S}, \mathtt{C}^k, \mathtt{C}^l)\mathrm{d}\mathbf{p} - \alpha. \tag{3.3}$$

This allows us to easily decide if a given pair $(\mathtt{C}^k, \mathtt{C}^l)$ should be included in the selection by testing if this improvement is positive. We can make an approximation to the integral and return all camera pairs that pass a certain threshold, namely $\alpha|\mathcal{S}|$.

An easy way to approximate a surface integral is uniformly random sampling:

$$|P|\int_{\mathcal{S}} f(\mathbf{p})\mathrm{d}\mathbf{p} = \mathbb{E}_P\big(|\mathcal{S}|\sum_{\mathbf{p}^i \in P} f(\mathbf{p}^i)\big) \tag{3.4}$$

for a random finite subset $P \subset \mathcal{S}$. Using this approximation, the constraint of positive improvement reads:

$$\frac{1}{|\mathcal{S}|}\int_{\mathcal{S}} w_s(\mathbf{p}, \mathcal{S}, \mathtt{C}^k, \mathtt{C}^l)\mathrm{d}\mathbf{p} = \frac{1}{|P|}\mathbb{E}_P\Big(\sum_{\mathbf{p}^i \in P} w_s(\mathbf{p}^i, \mathcal{S}, \mathtt{C}^k, \mathtt{C}^l)\Big). \tag{3.5}$$

A similar approach with non-uniform sampling can be applied to a product of functions of a single parameter $i \in \mathbb{N}$ as follows: $f(i)g(i) = \mathbb{E}_r\big(\mathcal{I}(r = i) \cdot N \cdot g(i)\big)$, where $N = \sum_i f(i)$ and r is sampled with probability proportional to $f(i)$, namely: $\mathrm{Prob}[r = i] = f(i)/N$. This approximation is useful when evaluating $g(i)$ is computationally expensive, but it works in the trivial case $g(i) = 1$ as well.

Therefore, we begin by separating out of w_s the component w_m not dependent on the side camera: $w_s(\mathbf{p}, \mathcal{S}, \mathtt{C}^k, \mathtt{C}^l) = w_m(\mathbf{p}, \mathcal{S}, \mathtt{C}^k)\frac{w_s(\mathbf{p}, \mathcal{S}, \mathtt{C}^k, \mathtt{C}^l)}{w_m(\mathbf{p}, \mathcal{S}, \mathtt{C}^k)}$. This clearly holds if w_m is nonzero on the whole support of w_s.

For the final formula, we pack the random points $\mathbf{p}^i$ and pairs of random indices k^i, l^i for both cameras all together into a single sampling set I. The probability distributions of k^i and l^i are designed to suit the approximation scheme:

$$\begin{aligned} \mathrm{Prob}[k^i = k] &= w_m(\mathbf{p}^i, \mathcal{S}, \mathtt{C}^k)/N_i \text{ and} \\ \mathrm{Prob}[l^i = l] &= w_s(\mathbf{p}, \mathcal{S}, \mathtt{C}^{k^i}, \mathtt{C}^l)/\big(w_m(\mathbf{p}, \mathcal{S}, \mathtt{C}^{k^i}) \cdot M_i\big), \end{aligned} \tag{3.6}$$

where $N_i = \sum_k w_m(\mathbf{p}^i, \mathcal{S}, \mathtt{C}^k)$ and $M_i = \sum_l w_s(\mathbf{p}^i, \mathcal{S}, \mathtt{C}^{k^i}, \mathtt{C}^l)/w_m(\mathbf{p}^i, \mathcal{S}, \mathtt{C}^{k^i})$.

The constraint of positive improvement is then stated as follows:

$$\frac{1}{|P|}\mathbb{E}_P\Big(\sum_{\mathbf{p}^i \in P} w_s(\mathbf{p}^i, \mathcal{S}, \mathtt{C}^k, \mathtt{C}^l)\Big) = \frac{1}{|I|}\mathbb{E}_I\Big(\sum_{(\mathbf{p}^i, k^i, l^i) \in I} \mathcal{I}(k^i = k)\mathcal{I}(l^i = l) N_i \cdot M_i\Big) > \alpha. \tag{3.7}$$

This reformulated constraint leads to the correct solution only asymptotically with growing $|I|$. Particularly, each triplet from I can lead to picking at most one camera pair, so for a C chosen this way it holds that $|C| \leq |I|$. This by-product is convenient for us, as it allows to directly limit the count of cameras chosen.

The resulting heuristic is much simpler than its derivation. It chooses a suitable sample count $|I|$, generates random triplets along the given distributions and evaluates $N_i \cdot M_{i,k}/|I|$ to check if they meet some threshold α. If they do, the corresponding camera pair $\mathtt{C}^k, \mathtt{C}^l$ is added to the output set. If they do not, we have to remember $N_i \cdot M_{i,k}/|I|$ from this attempt in case we would pick the same camera pair again; the new value is added up before the threshold check.

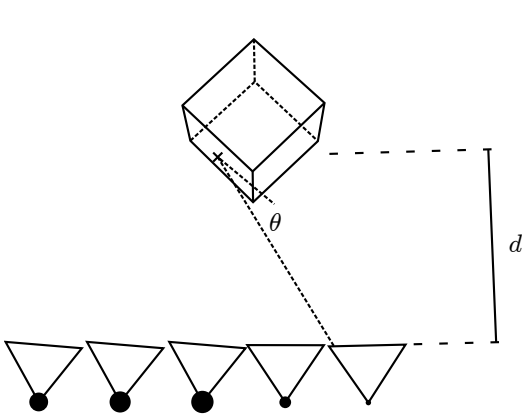
Before we deviate from this scheme, let us give formulas for w_m and w_s.

Using w_m, we wish to express how well is the surface patch around $\mathbf{p}$ displayed by the given camera:

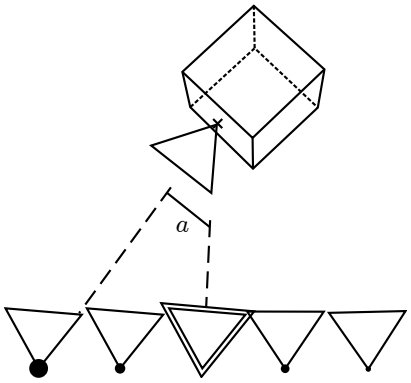
$$w_m(\mathbf{p}, \mathcal{S}, \mathtt{C}) = \frac{\cos(\theta)}{d(\mathtt{C}, \mathbf{p})^2} \cdot V(\mathcal{S}, \mathbf{p} \leftrightarrow \mathbf{c}) \cdot \mathcal{I}\big(\pi(\mathtt{C}\mathbf{p}) \in \Omega\big), \tag{3.8}$$

where $\mathbf{c}$ is the camera center of $\mathtt{C}$, θ is angle between the surface normal at $\mathbf{p}$ and the direction $\mathbf{p} \to \mathbf{c}$, and where $d(\mathtt{C}, \mathbf{p})$ is the distance of point $\mathbf{p}$ from $\mathbf{c}$ projected onto the viewing axis of $\mathtt{C}$. The visibility function $V(\mathcal{S}, \mathbf{p} \leftrightarrow \mathbf{c})$ is 1 if the open line segment between $\mathbf{p}$ and $\mathbf{c}$ does not intersect $\mathcal{S}$, otherwise 0. The indicator function $\mathcal{I}$ works similarly, in this case it tells whether $\mathbf{p}$ is projected inside of the image domain. Note that w_m is proportional to the area of a surface patch around $\mathbf{p}$ as projected by $\mathtt{C}$.

Our function w_s applies the same weighting using both cameras, but at the same time tries to (roughly) maximize the parallax between them. For that reason, we design a camera $\mathtt{P}$ centered at $\mathbf{p}$ and oriented along the surface normal, with an arbitrary focal length f. Furthermore, let us denote the centers of $\mathtt{C}^k, \mathtt{C}^l$



(a) Weights w_m for a given point; $d = d(\mathtt{C}, \mathbf{p})$



(b) Weights w_s for a given point and main camera; $a = |\pi(\mathtt{P}\mathbf{c}^k) - \pi(\mathtt{P}\mathbf{c}^l)|$

Figure 3.1: Illustration of the (unmodified) weights assigned to different cameras.

by $\mathbf{c}^k, \mathbf{c}^l$, respectively. Then,

$$w_s(\mathbf{p}, \mathcal{S}, \mathtt{C}^k, \mathtt{C}^l) = w_m(\mathbf{p}, \mathcal{S}, \mathtt{C}^k) \cdot w_m(\mathbf{p}, \mathcal{S}, \mathtt{C}^l) \cdot \frac{1}{f^2} |\pi(\mathtt{P}\mathbf{c}^k) - \pi(\mathtt{P}\mathbf{c}^l)|^2. \tag{3.9}$$

This includes the special but important case that w_s is zero if $k = l$. Note that $\mathtt{P}$ is helpful when calculating w_m as well; this is discussed in detail in Section 4.4.

The weighting is shown for a model setting on Figure 3.1. Cameras are rendered as triangles and weights assigned to them is depicted by the radius of the circle around their center. In Fig. 3.1a, weights for various possible main cameras are shown. In Fig. 3.1b, the camera drawn with an outline was chosen as the main camera, and weights are shown for choosing a corresponding side camera.

A trouble of this scheme is, as mentioned in the beginning, that we neglect correlation of frames corresponding to cameras that are close to each other. It appears difficult to design a scheme that would account for that effect correctly, as it can easily lead to solving an NP-hard graph problem. Instead, we let our scheme cut down the overall count of used cameras by weighting up those cameras that were chosen earlier, if they are applicable for the given point $\mathbf{p}$. This is the actual reason we approximate the weighting functions by random sampling. The modification is remarkably stronger for main cameras, as all side cameras assigned to it will be used in the triangulation at once: adding another side camera improves accuracy of the points without increasing their count.

We let the actual weights be

$$\begin{aligned} w'_m(\ldots) &= w_m(\ldots) \cdot \big(1 + c_m \cdot \mathrm{used}_m(k)\big) \text{ and} \\ w'_s(\ldots) &= w_s(\ldots) \cdot \big(1 + c_s \cdot \mathrm{used}_s(k, l)\big), \end{aligned} \tag{3.10}$$

where the values of c_m and c_s are adapted to the total camera count ($c_m \gg c_s$),

and where the functions used_m, used_s are indicators of whether the cameras of the given indices have been already selected. Note that the unmodified version of w_m is still used inside w'_s.

3.2 Reprojection

This section describes an image transformation that takes place before the optical flow calculation, and explains how to interpret the optical flow on the transformed image. The aim of this intermediary step is to improve the accuracy of the optical flow: in order to obtain reliable depth information, we need to use pairs of images with wide parallax. Unfortunately, both the optical flow algorithms discussed in Section 2.2 tend to perform badly if the optical flow contains large displacements.

Name:	the **reprojection** problem.
Input:	one side camera $\mathtt{C}^i$ and its frame I_i; main camera $\mathtt{C}^{\text{main}}$; estimated scene surface $\widehat{\mathcal{S}}$.
Output:	predicted image P_i; affine transformation matrix $\{\mathtt{A}^{x,y} \in \mathbb{R}^{2\times 2}\}$ for each pixel.

Our solution to this might be viewed as a pre-warping step for the optical flow based on the current estimate of the scene to be reconstructed. For a given frame I_i viewed by a side camera, we generate a prediction image P_i by back-projecting I_i onto the scene surface and rendering the textured scene from the main camera. To formally state this, we consider the raycasting function from 3D rendering literature:

Definition 5. *The raycasting function* $r(\mathtt{C}, \mathbf{x}, \mathcal{S})$ *returns the point* $\mathbf{p}$ *on scene surface* $\mathcal{S} \subset \mathbb{P}^3$ *seen by camera* $\mathtt{C} \in \mathbb{R}^{4\times 4}$ *at image coordinates* $\mathbf{x} \in \mathbb{R}^2$.

This concept is, in a sense, a counterpart to the projective division function π defined at the beginning of Section 2.3. The inversion of r with respect to $\mathbf{x}$ is the camera projection: $\pi\left(\mathtt{C} \cdot r(\mathtt{C}, \mathbf{x}, \mathcal{S})\right) = \mathbf{x}$. As long as the point $\mathbf{p}$ is visible in the camera $\mathtt{C}$, inversion in the opposite direction $r(\mathtt{C}, \pi(\mathtt{C}\mathbf{p}), \mathcal{S}) = \mathbf{p}$ also works.

The reprojection transformation produces an image P_i using the following formula:

$$P_i(\mathbf{x}) = I_i \circ \pi\Big(\mathtt{C}^i r\big(\mathtt{C}^{\text{main}}, \mathbf{x}, \widehat{\mathcal{S}}\big)\Big). \tag{3.11}$$

Note that it contains only color values from the input image, so it can be viewed as image warping by a displacement field $\text{warp}^+(\mathbf{x})$, where

$$\text{warp}^+(\mathbf{x}) = \pi\Big(\mathtt{C}^i r\big(\mathtt{C}^{\text{main}}, \mathbf{x}, \widehat{\mathcal{S}}\big)\Big). \tag{3.12}$$

The pixels of P_i with undefined value (if the result of raycasting is undefined or the source position would be outside the image domain Ω) are specifically marked, e.g., set to full transparency.

Camera projection can be locally well approximated by an affine mapping. Under the assumption that the scene surface is locally well approximated as a flat plane, we can compose the affine approximations of π, $\mathtt{C}^i$ and r to locally approximate $\mathrm{warp}^+\left((x,y)^\mathsf{T}\right)$ by an affine mapping $\mathbb{R}^2 \to \mathbb{R}^2$. The constant component of it is the actual value of warp^+ at the given pixel coordinates. The matrix $\mathtt{A}^{x,y}$ describing the linear component of this mapping is calculated for each pixel $(x,y)^\mathsf{T}$ where $P_i\left((x,y)^\mathsf{T}\right)$ is defined. These matrices will be later used to obtain covariance matrices for point triangulation.

This reprojection scheme is supposed to help calculating an optical flow between the side camera frame I_i and a main camera frame I_m by converting it to an optical flow problem between P_i and I_m. For the sake of completeness, we formulate the original flow field to be computed, assuming that the function flow relating P_i to I_m has been calculated such that $I_m(\mathbf{x}) = P_i\big(\mathbf{x} + \mathrm{flow}(\mathbf{x})\big)$. By substitution it follows that each pixel $\mathbf{x}$ in I_m corresponds to coordinates $\mathbf{s}$ in the input image I_i for

$$\mathbf{s} = \mathrm{warp}^+ \circ \mathrm{flow}^+(\mathbf{x}) = \pi\Big(\mathtt{C}^i r\big(\mathtt{C}^{\mathrm{main}}, \mathbf{x} + \mathrm{flow}(\mathbf{x}), \widehat{\mathcal{S}}\big)\Big). \tag{3.13}$$

3.3 Estimating the optical flow error

Our point triangulation function takes as an input the estimated screen-space positions along with their respective covariance matrices. These covariance matrices are not known, but their values may have significant impact on the accuracy of the triangulation. Therefore, we would like to estimate them as accurately as possible.

Name:	the **error estimation** problem.
Input:	two images I, P related by (noisy) displacement field flow; set of affine projection matrices $\{\mathtt{A}^{x,y}\}$ approximating the reprojection warping as defined Section 3.2.
Output:	set of covariance matrices $\{\Sigma^{x,y}\}$.

In other words, we want to approximate the probability density of the composite function $\mathrm{warp}^+ \circ \mathrm{flow}^+$ by a bivariate normal (Gaussian) distribution. The function warp^+ is produced by reprojection, as described in the previous section, and is defined by Equation 3.12. A formula for the composite function is given

by equation 3.13. The noise we consider originates from the computation of the flow function,[1] for which one of the algorithms presented in Section 2.2 is used.

To formalize the noise, we assume that $\text{flow}(\mathbf{x}) = d(\mathbf{x}) + n(\mathbf{x})$, where $d(\mathbf{x})$ is the actual exact displacement and $n(\mathbf{x})$ is for each $\mathbf{x}$ an independent random variable following a zero-mean normal distribution. This assumption is far from being realistic: the noise in each pixel of the optical flow is heavily correlated with the values in its neighbourhood. Thus, the actual distribution needs not be zero-mean and tends to produce gross outliers in the presence of repeating patterns. The zero-mean normal distribution model is chosen mainly because it is easy to deal with.

A bivariate zero-mean normal distribution is uniquely defined by its 2×2 covariance matrix $\mathtt{\Sigma}$. We firstly estimate this matrix as $\text{diag}(\sigma^2, \sigma^2)$, based on local similarity of the warped image $P \circ \text{flow}^+$ to I, and then we consider how this matrix transforms under the mapping of warp^+, to obtain probability distribution of the composite function $\text{warp}^+ \circ \text{flow}^+$.

Variance σ^2 is estimated separately for each pixel in main camera's frame as a simple L_1 norm difference between the warped image and its target. Because both the optical flow algorithms considered sequentially combine results from different levels of the scale pyramid,[2] we also estimate the difference on each level of the pyramid and sum the correctly scaled results for each pixel of the image to obtain its variance. Details about the relation between values of this norm and the corresponding variance are presented in Section 5.1.

An improvement could possibly be achieved by considering the amount of information available to the optical flow calculation as well. For example, regions of constant color do not supply any useful data, and straight lines constrain the flow only in one direction. We however did not continue in this direction since without realistic testing data it could lead to overfitting and practically useless results.

In order to keep the distribution of $\mathbf{s}^i$ normally distributed as well, the reprojection function warp^+ is locally approximated as affine mapping of the form $a(\mathbf{x}) = \mathtt{A}\mathbf{x} + \mathbf{a}^0$, where $\mathtt{A} = \mathtt{A}^{x,y}$ and $\mathbf{a}^0 = \text{warp}^+\big((x, y)^\mathsf{T}\big)$. Under such a mapping, a random vector $\mathbf{x}$ following normal distribution with covariance matrix $\mathtt{\Sigma}$ maps to a normally distributed random vector $a(\mathbf{x})$ with covariance matrix $\mathtt{A\Sigma A}^\mathsf{T}$.

The proof the last-mentioned statement is intensive on notation, but otherwise

[1] Image displacements resulting in similar noise can be introduced by lossy video compression. More details on this matter are given in Section A.3. However, we do not specifically account for such noise, as it would require (at least) to expect it in the main camera's frame too.

[2] The *scale pyramid*, or *Gaussian pyramid*, is obtained by repetitively scaling down the image by a given factor, optionally applying Gaussian blur before each scaling. Pre-calculating optical flow recursively on a downscaled version of the image, as used in Section 2.2, can be viewed as a step one level up the scale pyramid.

straightforward:

$$\operatorname{cov}\big((\mathtt{A}\mathbf{x})_i, (\mathtt{A}\mathbf{x})_j\big) = \operatorname{cov}\Big(\sum_k (\mathtt{A}_{ik}\mathbf{x}_k), \sum_l (\mathtt{A}_{jl}\mathbf{x}_l)\Big) = \sum_k \Big(\mathtt{A}_{ik} \sum_l \operatorname{cov}(\mathbf{x}_k, \mathbf{x}_l)\mathtt{A}_{jl}\Big)$$
$$= \sum_k \Big(\mathtt{A}_{ik} \sum_l \Sigma_{kl}\mathtt{A}^{\mathsf{T}}_{lj}\Big) = \sum_k \mathtt{A}_{ik}(\Sigma\mathtt{A}^{\mathsf{T}})_{kj} = (\mathtt{A}\Sigma\mathtt{A}^{\mathsf{T}})_{ij}, \quad (3.14)$$

where the second equality follows from the linearity of covariance. □

3.4 Point cloud filtering

The purpose of filtering is twofold here. Firstly, we want to detect outliers in the point cloud data and remove them. These outliers can result from several of the preceding stages and have not been dealt with in any way yet. Secondly, it is probable that some areas of the resulting point cloud will contain unnecessarily many samples. The applied polygonization algorithms may be computationally expensive, so we prefer to choose a small subset of the point cloud that represents the surface well.

Name:	the **point cloud filtering** problem.
Input:	a set $X = \{\mathbf{x}^i\} \subset \mathbb{R}^3$ of points along with their associated certainty values w_i and a filtering radius α.
Output:	a subset of X sufficiently describing the (unknown) scene surface.

Our algorithm works in two steps. Firstly, it estimates a local density around each point in the point cloud. This density is defined recursively as the sum of the densities of the points in a neighborhood weighted by a function of their distance. Such an approach is supposed to filter out even small contiguous patches of outliers. The radius of the neighborhood considered is the input parameter α.

Then, it processes all points ordered by their descending density. If the density of a point satisfies a certain threshold, it is added to the result and a subtractive penalty is assigned to the density of each of its neighbors. This process should result in choosing just a few most important points in each sphere of radius α.

Formally, the vector of densities $\mathbf{d}$ is defined as a vector that satisfies

$$\mathbf{d} = \operatorname{clamp}\left(\frac{\mathtt{A}\mathbf{d}}{\|\mathtt{A}\mathbf{d}\|}\right),$$

where

$$\mathtt{A}_{ij} = \mathtt{A}_{ji} = \begin{cases} 1 - |\mathbf{x}^i - \mathbf{x}^j|/\alpha & \text{for } i \neq j \text{ and } |\mathbf{x}^i - \mathbf{x}^j| < \alpha \\ 0 & \text{otherwise} \end{cases}$$

and

$$\operatorname{clamp}(\mathbf{d})_i = \begin{cases} \mathbf{d}_i & \text{for } \mathbf{d}_i < 2 \\ 2 & \text{otherwise.} \end{cases}$$

The above definition of $\mathbf{d}$ resembles the definition of an eigenvector of $\mathtt{A}$, except for the clamping applied. It is found by the classical power iteration method modified along this difference. We use the L_1 norm as the mapping $\|\cdot\|$. The process appears to converge quickly in practical settings. However, since the convergence is not formally ensured, we limit the number of iterations to a suitable constant.

The algorithm proceeds by assigning each point a variable 'score' s_i, initialized as $s_i \leftarrow \mathbf{d}_i$ for all i. It then processes all the points in order of their decreasing densities $\mathbf{d}_i$. In the case of equal densities, the order is randomized. If score of the point $\mathbf{x}^i$ being currently processed passes a threshold $s_i \geq t$, we add $\mathbf{x}^i$ to the output. The parameter t may be arbitrarily chosen between $0 < t < 2$; setting $t = 1$ seems to work well in practice. After processing each point, the score of all its α-neighbors $\mathbf{x}^k$ is decreased, depending on their distance and the (unmodified) density $\mathbf{d}_i$:

$$s_k \leftarrow s_k - \mathbf{d}_i \cdot \left(1 - |\mathbf{x}^k - \mathbf{x}^i|/\alpha\right)$$

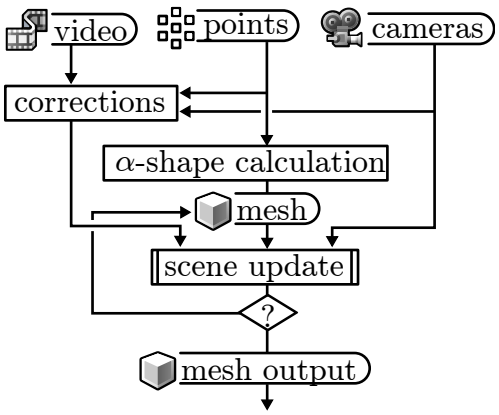
Note that the score of a point may become negative.

The process stops after processing all points with $d_i \geq t$, or optionally after generating a user-defined number of filtered points on the output.

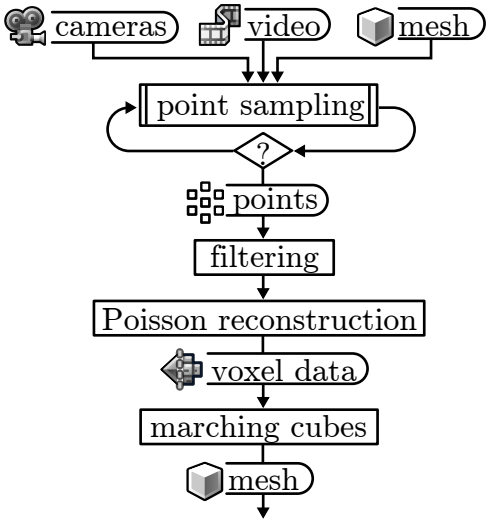
4. Implementation

We now give an overview of the resulting program, and comment on the interaction of the individual algorithms utilized in the pipeline.

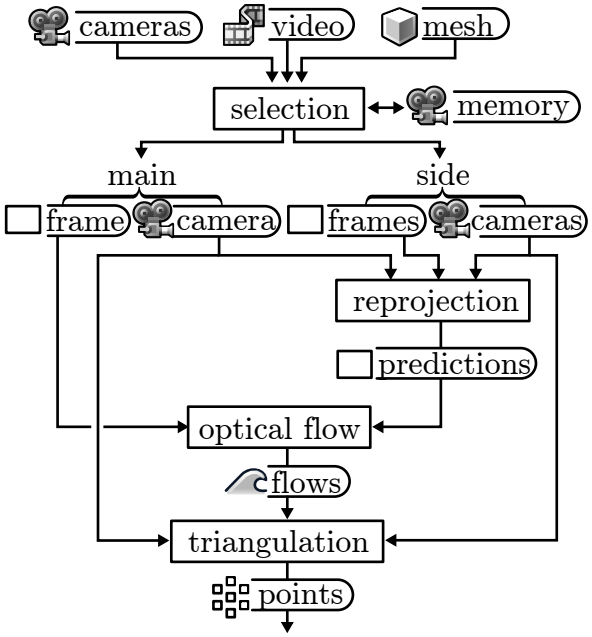
The whole reconstruction process is shown in Figure 4.1a: at the top, we have the input video sequence, the initial point cloud and the camera calibration parameters, and at the bottom, a reconstructed mesh is returned. Operations are depicted as boxes with sharp corners, data are the rounded ones. Diamond boxes with a question mark denote decisions made by a heuristic algorithm.



(a) Overall scheme of the program



(b) Expansion of the 'scene update' stage



(c) Expansion of the 'point sampling' stage

Figure 4.1: Scheme of operation of the whole program.

We start by preprocessing the whole video sequence, as described in Section 4.2. An alpha shape is calculated for the input points and along with the cameras and processed video, they are fed as input to the 'scene update' box. The output of the scene update stage is always a new mesh estimate of the scene. The subsequent decision box considers if the amount of detail obtained or the number of iterations performed suits the user's requirements, and if not, redirects this mesh as the input of the next update iteration.

Figure 4.1b shows the update process. Firstly, a new point cloud is accumulated by joining output obtained over several runs of the 'point sampling' stage. Each run corresponds to one main camera, so their count is also chosen by the camera selection heuristic. The results are not plain 3D points; each member of the cloud is assigned a normal and a precision. Normals of the points are estimated from their neighborhood in the main camera's frame, as described in Section 2.4. The precision is calculated from the combined probability of the triangulated result. However, to obtain results fair to all samples triangulated using a varying number n of side cameras, we set the precision to n-th root of the combined probability. We join this new cloud with the current one: regions where the current cloud is incorrect should get filtered out in the upcoming step.

The point cloud is filtered for outliers using the algorithm described in Section 3.4. The necessary value of α is obtained from the initial alpha shape calculation, and is divided by a custom factor after each iteration.The filtered points, their normals and the precision are fed to the Poisson reconstruction, which can make use of all these values. The resulting volumetric data is polygonized using the 'marching cubes' algorithm.

Figure 4.1c shows the process of obtaining a point cloud. As the first step, a main camera and one or more side cameras are selected by the heuristic designed in Section 3.1, using the current scene estimate and information about which cameras were selected previously. In the practical implementation, all the cameras to be used during a scene update iteration are selected all at once in advance and later read from memory, since the result is equivalent.

The frame I_i corresponding to each selected side camera $\mathtt{C}^i$ is reprojected from the main camera using the method described in Section 3.2, obtaining an image P_i predicted by the side camera. An optical flow field is calculated for each predicted image that relates it to the frame of the main camera I_m.

The triangulation processes only pixels of the main camera that depict the surface of the current scene estimate. For any such pixel $(x, y)^\mathsf{T}$, estimated coordinates $\mathbf{s}^i$ in the frame of a side camera $\mathtt{C}^i$ can be obtained by evaluating Equation 3.13 from Section 3.2. Furthermore, a covariance matrix $\mathbf{\Sigma}^i$ corresponding to the given pixel and side camera is obtained using the process described in Section 3.3. From these values, a scene-space point $\mathbf{p} \in \mathbb{R}^3$ is triangulated, as described

in Section 2.3. The probability density of each resulting point is stored.

After triangulation is finished for all available pixels, a normal vector is estimated for each of them using the method of Section 2.4.

4.1 Polygonization methods

We use alpha shapes for the first polygonization of the input point cloud and the Poisson reconstruction in all subsequent iterations.

Alpha shapes provide an excellent tool for polygonization of points uniformly distributed over a connected volume: in such a case, the α parameter will likely be estimated correctly, and the corresponding α-shape will tightly enclose the whole volume. However, there are many reasons against using them on the data considered here.

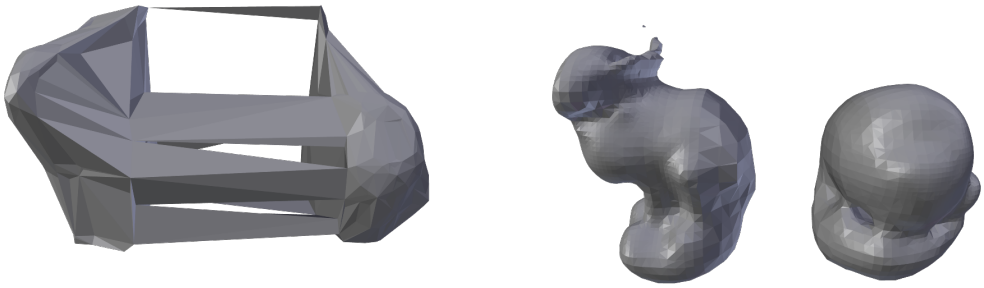
If the point cloud contains more details in a small portion of the input, it will be smoothed out due to the global character of the α value. As described in Section 2.1.1, we let the α radius be as small as possible so that all input vertices are included in one component of the resulting mesh. In particular, the result is nearly useless if the input consists of several separate parts: α must be chosen large enough to bridge them all, thus removing all details whatsoever. The same applies in the presence of outliers, so if alpha shapes would be applied for polygonization, the point filtering stage would be of enormous importance. If the initial point cloud is not polygonized by the alpha shape calculation properly, the user can bypass it by supplying the initial mesh directly.

Figure 4.2a displays the alpha shape of the Stanford bunny point set if half of it is moved apart.[1] To show the contrast, Fig. 4.2b shows the corresponding result of the Poisson reconstruction. Ears of the bunny are not reconstructed properly because normals of the point set were only inexactly estimated before the calculation.

Poisson reconstruction is well suited to processing noisy data and repeated structures that typically arise when triangulating a patch of surface independently twice. It relies on the surface normals, which is often an advantage, as can be explained by comparing the behavior of the two algorithms on real-world data. Because the triangulation produces points only on (supposed) scene surfaces, volume inside of solids will be free of any samples. In such cases, alpha shapes often produce a two-layer mesh that encapsulates the noise around the sampled surface from both sides. Poisson reconstruction usually manages to recognize all these samples as noisy representations of a single thing, and creates the output surface through the densest region.

Finally, using the Poisson reconstruction on the initial cloud is difficult, as the

[1] No bunnies have been harmed during the preparation of assets for this thesis.



(a) Alpha shape of the Stanford bunny split in half

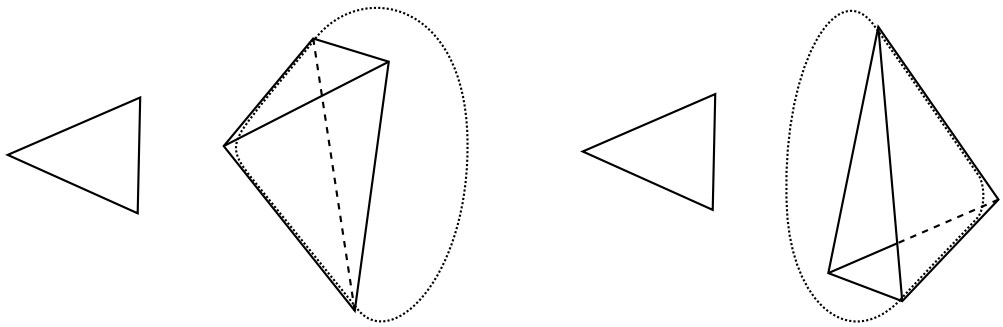
(b) Poisson reconstruction of the Stanford bunny split in half

Figure 4.2: Comparison of behavior of both polygonization algorithms on data with a gap.

input does not include any estimation of point normals. We tried to average the directions from which each point is viewed by the cameras, but such approach clearly fails if a surface is always displayed from the same direction, which is typically the case of the floor. The results of these experiments only reaffirmed the importance of correct normals, as they often did not resemble the scene at all.

A drawback of the Poisson reconstruction for our purposes is that it is designed for scenes with good volumetric representation. Such representation is rather inappropriate for reconstruction of bounded concave surfaces, such as when recording a corner of a room. In these cases, the volume to be reconstructed is sampled only from one side: no complications would arise if we tried to reconstruct the room as a whole. In regions free of any input samples, the optimal solution to the Poisson problem will simply try to minimize the surface area. Unfortunately, this may often lead to creating a surface between the cameras and the scene surface being reconstructed. Subsequent iterations of our algorithm would then obtain very misleading input.

This issue is illustrated in Figure 4.3. The cameras are depicted as triangles on the left, the actual scene is rendered in solid lines on the right. A possible surface resulting from the Poisson reconstruction is drawn in dotted smooth line. In Fig. 4.3a, the scene is a convex surface and the excessive surface is located on the back of it. Fig. 4.3b illustrates the troublesome reconstruction of a concave scene: the excessive surface that encloses the reconstructed volume is located between the camera and the actual scene, so that it will harm the subsequent iterations of the reconstruction. A similar bubble-like surface is noticeable also in Fig. 4.2b in the cut and bottom side of the bunny, as these regions do not contain any samples.



(a) Convex scene (b) Concave scene

Figure 4.3: Illustration of an undesired result of the Poisson reconstruction.

4.2 Video corrections

An operation in Figure 4.1a that was not mentioned yet is the 'corrections' box. Firstly, we correct the radial distortion, based on nonlinearity parameters that may be given in the input file. This is just another image warping transformation; its theoretical background is provided in Section A.2 in the attachments.

Secondly, if requested by the user, we try to revert the color adaptation performed by the camera; more details about its origin are discussed in Section A.1. Since both the optical flow algorithms considered work on grayscale images only, all we need is to stabilize the luminance of the sequence. We assume that to each frame i corresponds a linear mapping $l_i \colon \mathbb{R}^3 \to \mathbb{R}$ calculating the correct luminance from the input RGB values. We further assume that points of the initial cloud, where visible, are of constant luminance under this mapping; that is, if $\mathbf{p}^k$ is visible on frame i, then $l_i \circ I_i \circ \pi(\mathtt{C}^i\mathbf{p}^k) = c^k$ for some constant c^k. The mappings l_i can be expressed as dot products with unknown vectors $\mathbf{l}^i$:

$$l_i\big((r,g,b)^\mathsf{T}\big) = (r,g,b)^\mathsf{T} \cdot \mathbf{l}^i. \tag{4.1}$$

We solve for all $\mathbf{l}$ and c by alternating optimization, although we are interested in the vectors $\mathbf{l}$ only. To start, we assume $\mathbf{l}^i = (1,1,1)^\mathsf{T}$ for all i and obtain each c_k by averaging luminance of the given point in all relevant frames. Then, we assume the values c calculated in this way are correct, and find $\mathbf{l}^i$ for each frame that minimizes $\sum_k |c^k - I_i\big(\pi(\mathtt{C}^i\mathbf{p}^k)\big)^\mathsf{T} \cdot l_i|$ over all k such that $\mathbf{p}^k$ is not oversaturated in the given frame; this condition typically implies $I_i\big(\pi(\mathtt{C}^i\mathbf{p}^k)\big) \in \{1,\dots,254\}^3$. If there are less than three such points, we consider the oversaturated ones too. Then, we continue by re-estimating c and make a few iterations of this entire process.

4.3 Optical flow algorithms

The two algorithms for the calculation of the optical flow presented in Section 2.2 differ not only in the way of approximating the images but more importantly in what kind of smoothness they assume in the resulting flow field. The linear approximation scheme based on the approach by Horn and Schunck imposes a global smoothness constraint, whereas the result of Farnebäck's algorithm is smooth only due to fitting the result over a neighborhood (i.e., locally).

Although some sources regard the linear approximation scheme as outdated,[2] it has been shown to have a performance superior to the Farnebäck's algorithm if an adequate penalization function Ψ is applied [19]. Unfortunately, the OpenCV implementation adopted in our program sets Ψ to the identity function and so does not allow discontinuities in the flow field, which may be necessary for correct reconstruction of many scenes. Nevertheless, the linear approximation scheme may provide better results in the presence of repeated structures and textureless regions.

4.4 Application of mesh rasterization

The reprojection, and specifically the raycasting function r as described in Section 3.2 is performed by an OpenGL-faciliated offscreen rendering. We use a custom GLSL fragment shader that samples the side camera's frame in exact correspondence to the formulas stated, and anisotropic mipmapping to preserve details in the texture. The shader uses a simple shadow buffer to mask out areas occluded to the side camera. To run correctly, it requires OpenGL 3.0 support; although we expect compatibility problems, this standard should be supported even by software rendering.

In the resulting image, the masked out regions and empty areas without scene geometry are filled with the main camera's frame. We use the optical flow algorithms mostly as a black box, and we simply need to provide them with color values for the whole image. Using this background effectively suggests zero flow outside of the region of interest, which is probably the best guess we can make.

Along with the prediction frame, we render a depth map of the scene.[3] We use it to directly read off initial estimate of depth for triangulation of each pixel. Notice that although the depth map is nonlinear in its nature, its values are correct z coordinates in camera space, so we can just apply inverse camera projection

[2] In the OpenCV library version 2, the algorithm is part of the 'legacy' module and called 'obsolete' in the documentation.

[3] A *depth map*, or *Z-buffer*, depicts the same (visible) surfaces as the corresponding rendered image but its values are camera-space depth coordinates of the respective surface points.

$\mathtt{C}^{-\mathsf{I}}$ to them without sophisticated remapping.[4]

From its gradient, we directly calculate the local warping matrices $\mathtt{A}^{x,y}$. In the first glance, it would seem more proper to obtain a normal pass from OpenGL and calculate these matrices from it. Not only that using this depth map gradient is much easier, but the details of scene geometry smeared out in the process are exactly those smeared out in the predicted image by reprojection.

A depth map rendering is also used during camera selection (as described in section 3.1) to cheaply estimate the visibility function. We render a depth map of the scene using the camera $\mathtt{P}$ and check visibility of a given scene-space point $\mathbf{x}$ by comparing the projected depth of $\mathtt{P}\mathbf{x}$ to the depth image at the corresponding coordinates.

4.5 User-defined parameters

The program offers four groups of parameters for tuning the stages of the calculation that we consider the most important.

A custom model may be supplied by the user as a replacement of the initial alpha shape calculation. Clearly, the program may diverge from the correct solution or completely fail if the initial mesh reconstruction is too far from the real. If the input point cloud is too sparse or otherwise misleading to alpha shapes, providing such model is necessary.

The camera selection heuristic offers a few parameters as well. As described in the corresponding section (3.1), there is a precision parameter α and a sampling parameter to limit the maximal camera count $|I|$. Both of these are available to the user.

The optical flow algorithms have their parameters. These often need to be modified to suit the resolution of the sequence and its other characteristics. The ones of them we consider the most important are exposed to the user of our program.

Finally, the user may request to scale the whole sequence down before processing, to speed up the calculation. The program can extract only every n-th frame of the sequence and skip all others, for use on very lengthy videos.

4.6 Adopted libraries and practical remarks

The OpenCV library [7] is used for matrix arithmetic, image processing and a few other utility functions. The library also provides us with the implementation of both optical flow algorithms.

[4] Actually, a slight remapping is necessary: OpenGL defines camera space as $[-1, 1]^3$ but linearly maps the depth map values into the range $[0, 1]$. We have to map them back.

Our program uses the CGAL library [8] for calculating the alpha shape and the underlying Delaunay tessellation because by our observations, it is robust and easy to use. It is rather slow on dense data, but not significantly slower compared to other open-source libraries for Delaunay tessellations.

For the Poisson reconstruction, the Point Cloud Library (PCL, [9]) is used. Its code is mostly the original implementation by Michael Kazhdan and Matthew Bolitho, who released it under the GNU GPL license [26]. However, installing the PCL library may be difficult; because of this, the CGAL implementation of Poisson reconstruction may be used instead. Note that the CGAL implementation represents the volume using Delaunay tessellation instead of an octree, and therefore also uses a different algorithm for polygonization. We did not describe these algorithms in this text; by practical observations, they produce nicer results but are considerably slower compared to PCL.

The implementation of our method, provided as attachment to this thesis, is not designed for an end user. It is limited to Linux systems running the X window system and it has no interactive controls. We however hope that it can serve as a basis for reusing the algorithm elsewhere, or for further testing and improvements of the method.

Reconstruction of a typical video sequence — consisting of hundreds to thousands of frames — may take over an hour, depending on the particular scene and other circumstances. To a certain amount, this is reasonable, considered the amount of data to be processed and our initial aim to extract as much scene information out of it as possible. If a faster operation is desired, the parameters should allow for a speedup in orders of magnitude, at the cost of less detailed reconstruction.

5. Evaluation

5.1 Error of the optical flow

In Section 3.3, we explained how the L_1 norm is used for a posteriori error estimate of the optical flow. Here, we try to justify this choice by testing on scenes with known ground truth.

We evaluate (only) Farnebäck's optical flow algorithm on two artificial scenes. They both consist of a flat plane displaced by Perlin noise. Its texture is Perlin noise as well, with different parameters. One camera views it perpendicularly, the other from an angle. The rendered image is 640×640 pixels in size.

As already stated, we assume the optical flow is distorted by circular Gaussian additive noise with unknown variance σ^2. Then, the square of the radius of the noise values shall follow an (appropriately scaled) χ^2 distribution with variance $4\sigma^4$; it is scaled by a factor σ^2. Knowing the ground truth, we know the noise values and can make a maximum likelihood estimate of its variance.

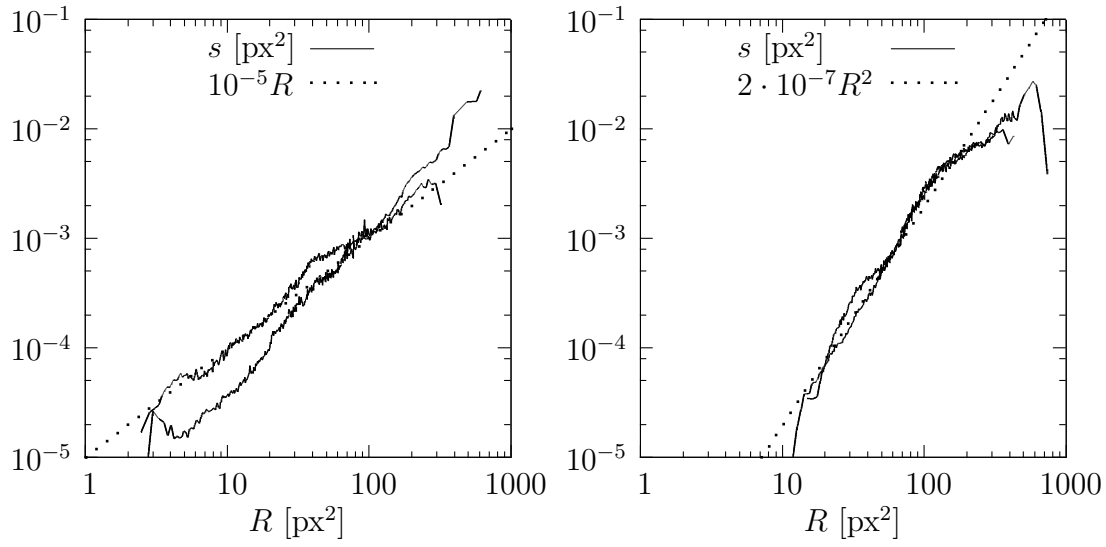
Probability density of the scaled χ^2 distribution is $p(R) = \frac{1}{2s}\exp(-\frac{R}{s})$ for $s = \sigma^2$ and $R = r^2$ being the squared radius. We find s such that the probability $P(R_1, \ldots, R_n|s)$ is maximized; it is the best option as we have no priors for $P(s)$ nor $P(R_i)$. The probability is a product of $P(R_i|s)$, i.e., of known distributions. It is maximized at

$$s = \frac{\sum_i R_i}{n}. \tag{5.1}$$

The values of s obtained this way are shown in Figure 5.1 in a log-log plot. The plot shows two solid lines, each corresponding to a set of values obtained from a scene as described, and both scenes differing only in noise seed. Each value was estimated from $n = 50$ samples, out of $3 \cdot 10^4$ samples per scene. Dotted lines show a suggested asymptotic curve.

For a small viewing angle (Fig. 5.1a), the dependency seems to be linear. However, as the angle increases (Fig. 5.1b), it deteriorates to some extent to a quadratic dependency. The change may be caused by the prevailing occlusions.

In real scenarios, we do not know the viewing angle, but expect it to be rather small. Since a linear factor of the mapping cancels out for the purposes of the minimization, it is justified to use the L_1 norm directly.



(a) View from 7.5°

(b) View from 30°

Figure 5.1: Estimated values of $s = \sigma^2$ for varying squared radii $R = r^2$ of noise.

5.2 Testing sequences

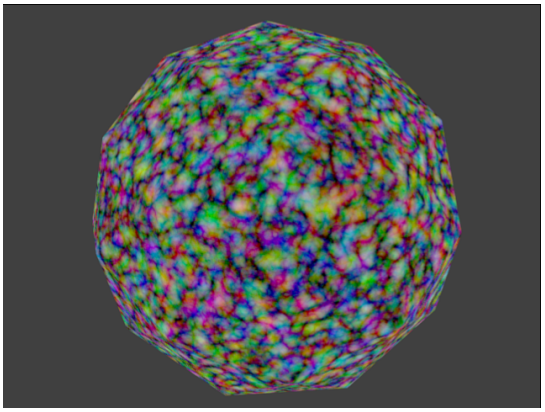
5.2.1 Perlin sphere

We executed the algorithm on a rendered sequence of a subdivided icosahedron with Perlin texture. The sequence consists of 30 frames at resolution 640×480 during which the camera moves over a 108°-arc centered at the sphere. An example frame of the sequence is shown on Figure 5.2a. There are no occlusions on the front side of the model. Its convex shape is perfectly suited for Poisson reconstruction.

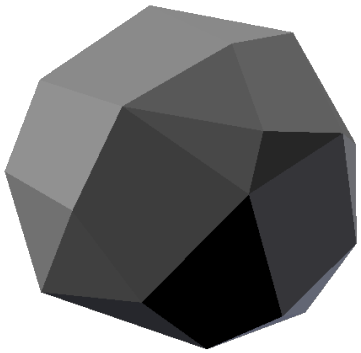
The resulting mesh reconstructed from this model sequence is indeed accurate. Figure 5.2b shows the initial scene estimate, generated as alpha-shape from 21 points. Figures 5.2c and 5.2d show the result after the first iteration and the ground truth scene, respectively. Farnebäck's optical flow algorithm was used since the scene provides enough texture. Also, the camera selection threshold was increased from the default value. The triangulated point cloud consisted of over 500,000 points, out of which about 80,000 passed the filtering stage. The reconstruction took 6 minutes on an Intel Core2 workstation.

5.2.2 Stone relief

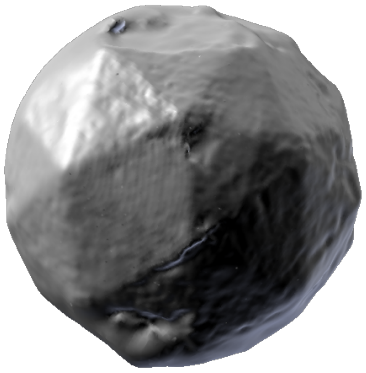
This sequence consists of 230 images of a stone relief in the portal of a cathedral. An example frame of the sequence is shown in Figure 5.3a. The image is slightly blurry, but the diffuse stone surfaces nevertheless provide very good texture for the optical flow calculation. The Farnebäck's algorithm has been used.



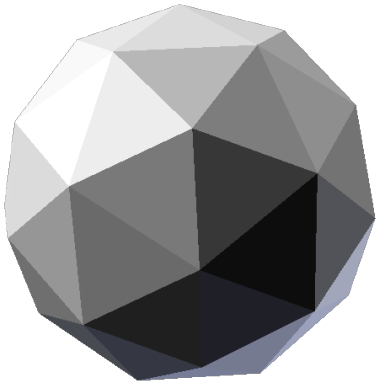
(a) A frame of the sequence



(b) Initial scene estimate



(c) Reconstruction after the first iteration



(d) Ground truth

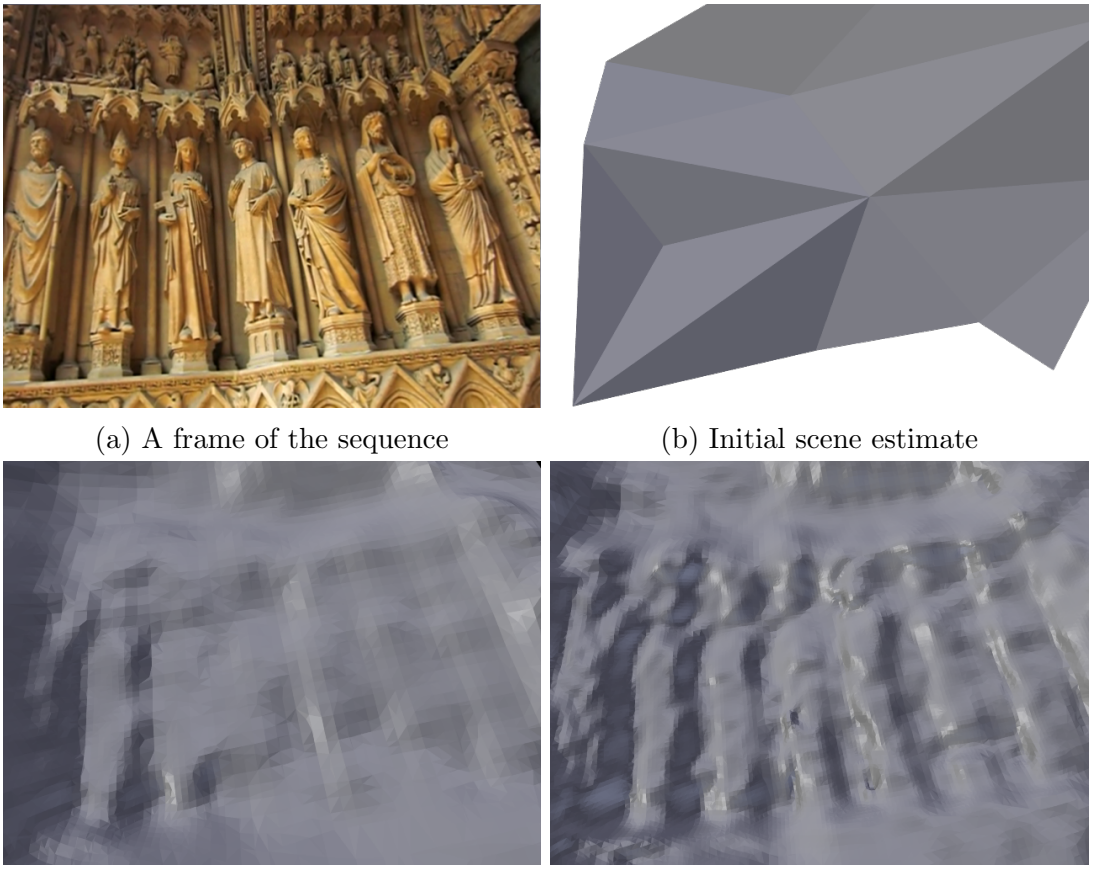
Figure 5.2: Reconstruction of the 'Perlin sphere' synthetic sequence.

The sequence was taken at high noon and the camera applied a rather strong color adaptation during the recording. Therefore, the exposure normalization feature of our program had to be enabled during the reconstruction. Figure 5.3b shows the initial scene estimate, which is an alpha shape of 23 points. Figures 5.3c and 5.3d display the result of reconstruction after the first and the second iteration, respectively. Clearly, the result improves, which can be seen as a justification of our iterative reconstruction approach. Each of these iterations took 10 and 16 minutes, respectively, on an Intel Core2 workstation.

5.2.3 Glossy car

This sequence consists of 293 frames depicting a static Opel Meriva car on a mostly sunny day (one of these is shown in Figure 5.4a). It is provided as an example of failure of the assumptions: reflections on the surfaces of the car are much more prominent than their actual texture. Because of this, the optical flow calculation cannot provide any good results and no useful data is extracted from the video.

To exaggerate the effect, we supplied the program with a model of the car



(a) A frame of the sequence (b) Initial scene estimate

(c) Reconstruction after the first iteration (d) Recon. after the second iteration

Figure 5.3: Reconstruction of the 'stone relief' sequence (Cathedral Saint-Étienne de Metz, France).

(a) A frame of the sequence (b) Initial scene estimate (c) Reconstruction after the first iteration

Figure 5.4: Reconstruction of the 'Opel Meriva' sequence.

(shown in Fig. 5.4b). The scene reconstruction diverges from the (mostly correct) initial estimate, especially on the windows where clouds are reflected with high contrast. A result after the first iteration is shown in Figure 5.4c.

Triangulating the geometry of glossy surfaces seems intuitively possible, but it would perhaps require a vastly different approach than the ours.

6. Conclusion

We have presented an algorithm for iterative estimation of scene geometry from a calibrated video sequence. Thanks to utilizing the camera calibration and a priori scene estimate, it is robust against rough camera movement, such as camera shake.

The algorithm is however very unstable if the initial scene estimate is far from being accurate. In such a case, camera selection routine will give nearly random results. Also, triangulation may diverge from the optimum if its starting depth values are too inaccurate.

Such failure is often caused by the alpha shape calculation. Its performance could probably be improved by using another heuristic for guessing the right value of α, e.g., by maximizing the total face count. We did not make any experiments with that; however, our program allows the user to bypass the initial tessellation and supply a mesh directly on input. Another possible solution might be to estimate a varying value of α on a local basis [17].

In many practical settings, it is necessary for the user to supply the mesh scene estimate directly. However, we believe this is not a real issue, as the mesh needs not be detailed. Simple solids or a subset of the convex hull of the initial point cloud usually supply sufficient initial estimates.

The triangulation is imprecise if the camera motion does not provide views with a wide parallax. This instability is inherent to the triangulation problem, since a slight amount of noise can substantially degrade the result. Specifically, the reconstruction from tracks such as the Yosemite sequence yields bad results because the camera moves only in the forward direction.

The proposed point cloud filtering algorithm appears to serve its purpose well, as long as it is supplied with meaningful data.

The threshold for the camera selection heuristic must be often altered by the user, to obtain a solution in given time limits. The computational time can be roughly guessed from the number of the main and side cameras selected for the first iteration. In the following iterations, these numbers should not substantially change. Apart from this choice of parameters, the algorithm runs without any user interaction. This property allows it to be used for batch processing, possibly on a remote server.

6.1 Future work

Accuracy of the optical flow calculation may be highly improved by imposing the *epipolar constraint*: since the camera calibration is known, valid positions of each remapped pixel are limited to a line in the image space. By further

extending this idea, the optical flow calculation may be incorporated deeper into the triangulation process. The triangulation could process image data directly, minimizing the energy as defined in the Horn-Schunck scheme for all images at once. We have tested this only briefly without any smoothness term, but the results seemed surprisingly good.

The performance of the algorithm may be improved by outputting the triangulated data in a format better suited for the subsequent Poisson reconstruction. Instead of managing a point cloud, the triangulated results could be directly saved into the octree, if its dimensions and depth could be reliably estimated.

Another speed gain could be rather easily attained by using downsampled versions of the sequence in the first iterations of the algorithm. After this initial phase, the resolution could be gradually increased to obtain a detailed scene estimate. Furthermore, the resolution could be different for each set of selected cameras, depending on their distance from the surface.

Bibliography

[1] *Bundler: Structure from Motion for Unordered Image Collections.* Source code available. (retrieved 2013-07-16) `http://www.cs.cornell.edu/~snavely/bundler`

[2] *PMVS2: Patch-based Multi-view Stereo Software.* Source code available. (retrieved 2013-07-16) `http://www.di.ens.fr/pmvs`

[3] *Blender 3D.* Source code available. (retrieved 2013-07-15) `http://blender.org`

[4] *Adobe After Effects CC.* (retrieved 2013-07-15) `http://adobe.com/products/aftereffects.html`

[5] *Cinema 4D Studio.* (retrieved 2013-07-15) `http://www.maxon.net/products/cinema-4d-studio`

[6] *SynthEyes.* (retrieved 2013-07-15) `http://www.ssontech.com`

[7] *OpenCV: Open Computer Vision Library.* Source code available. (retrieved 2013-07-15) `http://opencv.org`

[8] *CGAL: Computational Geometry Algorithms Library.* Source code available. (retrieved 2013-07-15) `http://www.cgal.org`

[9] *PCL: Point Cloud Library.* Source code available. (retrieved 2013-07-15) `http://pointclouds.org`

[10] *Microsoft PhotoSynth.* (retrieved 2013-07-16) `http://photosynth.net`

[11] *Libmv: A Structure from Motion Library.* Source code available. (retrieved 2013-07-16) `http://code.google.com/p/libmv`

[12] *Voodoo Camera Tracker.* (retrieved 2013-07-15) `http://www.digilab.uni-hannover.de/docs/manual.html`

[13] Newcombe, Richard A.; Davison, Andrew J.: *Live Dense Reconstruction with a Single Moving Camera.* IEEE CVPR, 2010.

[14] Newcombe, Richard A.; Lovegrove, Stephen J.; Davison, Andrew J.: *DTAM: Dense Tracking and Mapping in Real-Time.* IEEE ICCV, 2011.

[15] Agarwal, S.; Snavely, N.; Simon, I.; Seitz, S.; Szeliski, R.: *Building Rome in a Day.* IEEE ICCV, 2009

[16] Bujňák, Martin: *Dense Reconstruction from Uncalibrated Video.* Diss., Bratislava, 2005.

[17] Cazals, F.; Giesen, J.; Pauly, M.; Zomorodian, A.: *Conformal Alpha Shapes.* Eurographics Symposium on Point-Based Graphics, 2005.

[18] Edelsbrunner, Herbert; Kirkpatrick, David G.; Seidel, Raimund: *On the shape of a set of points in the plane.* IEEE Trans. Information Theory 29 (4), 1983: p. 551–559.

[19] Brox, T.; Bruhn, A.; Papenberg, N.; Weickert, J.: *High accuracy optical flow estimation based on a theory for warping.* Computer Vision-ECCV 2004: p. 25-36, Springer Berlin Heidelberg, ISBN 978-3-540-21981-1.

[20] Farnebäck, Gunnar: *Two-Frame Motion Estimation Based on Polynomial Expansion.* Image Analysis, 2003, ISBN 978-3-540-40601-3.

[21] Farnebäck, Gunnar: *Polynomial Expansion for Orientation and Motion Estimation.* Diss., Linköping, 2002, ISBN 91-7373-475-6.

[22] Hartley, Richard; Zisserman, Andrew: *Multiple View Geometry in Computer Vision.* 2nd edition, 2003, Cambridge University Press, ISBN 0521540518.

[23] Horn, Berthold K. P.; Schunck, Brian G.: *Determining optical flow.* Artificial Intelligence, Vol. 17, 1981: p. 185–203.

[24] Kazhdan, Michael; Bolitho, Matthew; Hoppe, Hugues: *Poisson Surface Reconstruction.* IEEE Symp. Geometry Processing, 2006.

[25] Lucas, Bruce D.; Kanade, Takeo: *An Iterative Image Registration Technique with an Application to Stereo Vision.* International Joint Conference on Artificial Intelligence, 1981: p. 674–679.

[26] *Screened Poisson Surface Reconstruction.* (retrieved 2013-07-17) `http://www.cs.jhu.edu/~misha/Code/PoissonRecon`

[27] Liu, Yuanxin; Snoeyink, Jack: *A comparison of five implementations of 3D delaunay tessellation.* Combinatorial and Computational Geometry, 2002: p. 439–458, ISBN 9780521848626.

[28] Lorensen, William E.; Cline, Harvey E.: *Marching cubes: A high resolution 3d surface reconstruction algorithm.* SIGGRAPH 1987: p. 163–169.

[29] Matoušek, Jiří: *Lectures on Discrete Geometry.* 1st edition, 2002, Springer, ISBN 0387953744.

[30] Parsonage, Phil; Hilton, Adrian; Starck, Jon: *Efficient Dense Reconstruction from Video.* IEEE CVMP, 2011.

[31] Snavely, Noah; Seitz, Steve; Szeliski, Richard: *Photo Tourism.* SIGGRAPH, 2006.

[32] *Photo Tourism: Exploring Photo Collections in 3D.* (retrieved 2013-07-16) `http://phototour.cs.washington.edu`

[33] Wendel, A.; Maurer, M.; Graber, G.; Pock, T.; Bischof, H.: *Dense Reconstruction On-the-Fly.* IEEE CVPR, 2012.

Attachments

A Common types of video degradation

This appendix describes some of the most important aspects of video degradation induced by typical consumer-grade cameras.

A.1 Colorimetric issues

Photographic sensors such as CCD[1] are mostly insensitive to color. Typically, a grid of color filters is placed in front of the sensor, so that each sensor cell receives only a part of the incoming spectrum. If white light shines on cells of a single color, the observed image will be tinted. This is noticeable in the presence of fine straight line patterns in the scene, and is referred to as *color moiré*. Some video camera designs overcome this problem by using a beam splitter and three separate sensors for the individual color channels.

The amount of light incident on a sensor cell obeys the cosine law. Detected values far from the image center are therefore darker. This fact is known as *vignetting*.

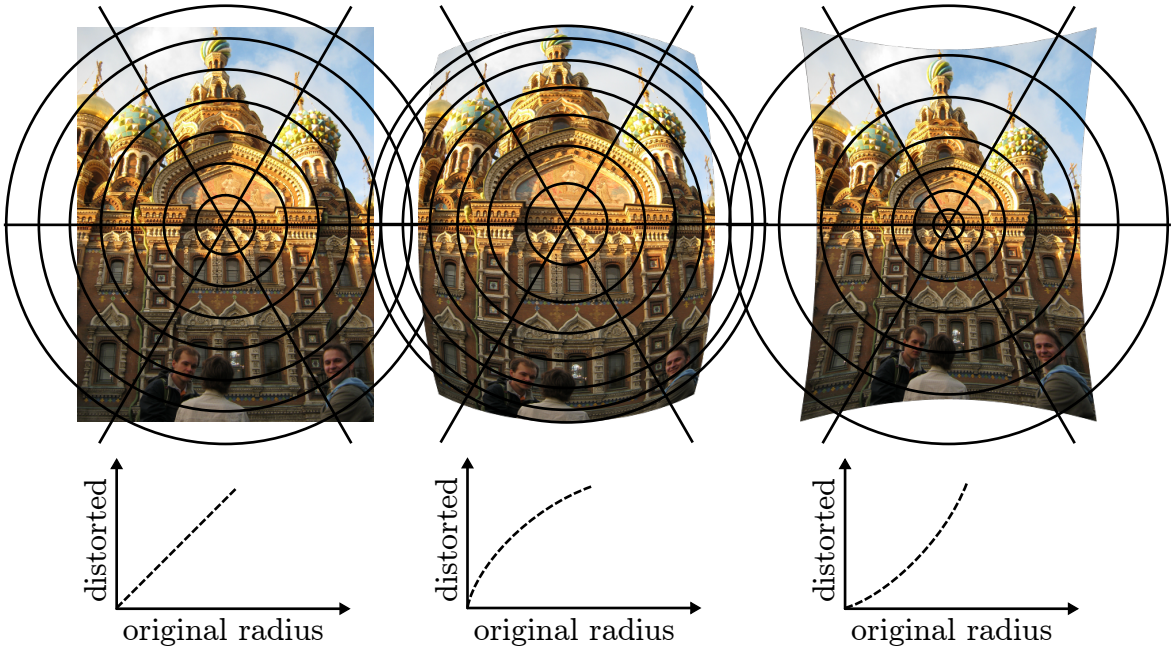
Output of the camera sensor is usually linear in the amount of incident light (within the range we are interested in). Many cameras support direct storage of this data in the so-called *raw* format, each sample being a floating point value. However, only professional-grade cameras allow to record video sequences in this format, as it is too large and difficult to process by the end user.

Instead, the values are discretized and clamped, typically into the range of $\{0, \ldots, 255\}$. To make the image more distinctive and visually pleasant, a custom *camera response function* is applied to the values before rounding them off. This function is designed to compress the range of bright colors, mimicking classic film cameras. Further, a *white balance* mapping is applied to all the three channels at once, trying to emulate human eye's color constancy. The transformation is sometimes expressed by a matrix multiplication, but the camera manufacturer may apply a more sophisticated mapping.

A.2 Optical distortions

Real camera projection exhibits nonlinearities, which can be diminished to some extent by using more lenses and by precise lens manufacturing. Another option is to restore the image by estimating and inverting the distortion.

[1] *Charge Coupled Device* (CCD) is the most common type of camera sensor at the time of writing.



(a) Original image (b) Barrel distortion (c) Pincushion distortion

Figure 6.1: Illustration of radial distortions on a photo (Church of the Savior on Blood, Saint Petersburg).

The most prominent is the *radial distortion*, commonly called either barrel (Figure 6.1b) or pincushion (Fig. 6.1c) distortion, depending on its shape. It is best expressed as a mapping in polar coordinates, since it affects only the radius of each projected point. We approximate the scaling of the radius as a cubic function of the radius squared: $r_{\text{distorted}}(r) = r \cdot (1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$. It is preferred to remap the input images in the beginning; although it induces slight blur, it saves us from inverting this function during projection.

A.3 Lossy video formats

Although a range of lossy video codecs are available, only few standards defined by Motion Pictures Experts Group appear to be used by camera manufacturers in practice. Here, we discuss the basic aspects of MPEG-4 AVC (also known as H.264), because it is the newest and perhaps the most advanced of them. However, most of these concepts are common to all of the codecs.

The sequence is saved frame-by-frame, with each frame further divided into *macroblocks* of 16×16 pixels. The content of a macroblock is a sum of its *prediction* and its *residual signal*. These values need not be stored explicitly. Before using a frame in any way (such as displaying it), a *deblocking filter* is applied to reduce grid-like artifacts caused by the compression.

Perhaps the strongest tool of the compression is the *motion estimation*. In the case of motion-estimated macroblocks, the prediction values are copied (in-

terpolated) from one or two other frames as encoded in the sequence. The source area is of the same size, but displaced by a vector expressed up to a quarter-pixel precision. A macroblock may be further subdivided into up to 16 parts with distinct motion vectors.

If a macroblock is not motion estimated, each pixel of its prediction is a weighted average of pixels neighboring on the top and left edge of the macroblock. The weights are one of a few predefined simple schemes. Some frames of a sequence do not contain any motion estimated macroblocks at all, so that they can be decoded directly.

A macroblock may be *skipped*, meaning that no more information is stored. Such macroblock is filled using motion estimation: its motion vector is averaged from its neighbors and its residual signal is set to zero.

The pixel data is processed in YCbCr color space. An integer-valued (and thus reversible) approximation of Discrete Cosine transform is applied to each channel in 4×4 pixel blocks. Each of the chroma channels has half resolution in both dimensions, and its missing values are interpolated. This implies that the residual signal of a macroblock has 16 luminance blocks and $4 + 4$ chroma blocks in total.

The deblocking filter is strictly defined and mandatory, because its result is used by other motion-estimated frames. It is basically an adaptive smoothing filter applied to an 8-pixel row across the boundary of two neighboring pixel blocks, and then again in columns.

B Sparse scene reconstruction

The goal of *sparse reconstruction* is to obtain camera calibration from a set of images displaying an unknown scene from different viewpoints. Of interest for this thesis is the specialized case when the images form a video sequence acquired by a single moving camera. As a side product of the classical approach, a sparse set of points on estimated scene surface is generated during the computation.

B.1 Overview

Sparse reconstruction programs typically proceed in two stages. In the first step, a sparse set of *feature points* is identified on the input video frames by considering their visual appearance. Each of these points correspond to a patch of surface in the scene, and there are at least two points (in different frames) corresponding to the same surface patch. The second stage processes only the image-space positions of these points; it assigns an estimated scene-space position to each of them and external camera calibration for each frame.

Two main approaches to identifying feature points are commonly used, which are discussed in the following two sections.

B.2 Feature tracking

In the feature tracking method, a set of tens to hundreds of interest points is identified in a single frame chosen by the user. Each point's *feature area* is typically a square 20–50 pixels in width with the interest point as its center. This feature area should provide enough texture in order to be easily localized. They can be manually chosen by the user if necessary.

The tracking then proceeds either to the previous or to the next frame in the sequence. If the camera motion is reasonably slow, all the interest points will stay roughly in their positions, and the texture around them stays very similar. Thanks to this, we can locally search for the feature area in the new frame.

One of the most common local image search techniques is the Kanade-Lucas-Tomasi tracker [25]. It uses a linear approximation of the image and iteratively searches for the feature area across the individual levels of the detail pyramid.

If the expected motion is slow enough, it is even feasible to use a brute force search. However, in order to get an accurate reconstruction, we want to identify the feature point position with subpixel precision. Therefore, the brute force search is usually followed by a nonlinear optimization technique (such as Gauss-Newton) on subpixel scale.

Nonlinear optimization can be also used to solve for rotational change of the feature area, as well as other types of projective mappings. It can solve for slow exposure changes between the frames. Considering these usually improves robustness of the algorithm.

The tracking then continues to another neighbor frame in the sequence. A decision must be made which frame (of the already tracked ones) to take the representative feature area from.

This approach is, for example, used in programs Adobe After Effects [4], Blender [3] and Voodoo Tracker [12].

B.3 Feature matching

A perhaps more sophisticated approach is commonly used to allow for a full automation of the task. A set of feature points is selected in each frame of the sequence, and their amount ranges to thousands per frame. The main concern during the feature point selection is repeatability, so that the same scene point is selected many times throughout the sequence. Many algorithms have been proposed for this task; most of them are based on finding maxima in partial derivatives of different scales of the image (i.e., its gradient or Laplacian).

The feature area of each of the points detected is then described by a *feature vector*. This description should be invariant to the expected image distortions, so that the feature vector is the same when its source image is, e.g., rotated. Many feature descriptors are also nearly invariant under global illumination changes. The feature vectors have typically tens to hundreds of dimensions.

All these feature vectors are fed to a nearest-neighbor search structure. We expect that feature vectors corresponding to the same features in the scene across different frames will be noticeably close to each other, and ask the search structure for a set of such putative matches.

Some of these will be of course matched incorrectly. Thanks to the fact that correctly matched points between two frames are related by a projective mapping, an outlier filtering scheme such as RANSAC can be used to separate the correct matches [22, p. 117–121].

This approach is used for example in Cinema4D [5] and Syntheyes [6].

B.4 Camera calibration

The known position $\hat{\mathbf{f}}^{i,k}$ of feature point k in the i-th frame is related to its unknown scene-space position $\mathbf{p}^k$ by camera projection:

$$\mathbf{f}^{i,k} = \pi(\mathtt{C}^i \mathbf{p}^k), \tag{6.1}$$

where $\mathtt{C}^i$ is an unknown camera matrix corresponding to the i-th frame. The vector $\mathbf{f}^{i,k}$ is an exact value of the projected position, since the measured vectors $\hat{\mathbf{f}}^{i,k}$ typically contain a substantial amount of noise. We can state such equation for each i and k if feature point $\mathbf{p}^k$ was successfully identified in frame k. This all in all makes a huge nonlinear system, in a typical case vastly overdetermined. We can see it as a minimization problem and solve it as such, but to ensure convergence, we need an initial approximate solution.

Estimate of the internal camera calibration can be either extracted from the sequence metadata, or is supplied by the user. Without loss of generality, we can set external calibration for one camera to an arbitrary value, which effectively equals to translating and rotating the resulting scene.

To make a bootstrapping step, we need two frames depicting at least 8 common feature points. It allows us to solve for the so called *fundamental matrix* that relates points between these two frames. One of the cameras can then be moved to a default position, and relative position of the other is determined by the fundamental matrix.

Knowing a point's screen-space position in at least two frames, we can perform triangulation similarly as described in Section 2.3 to get its position in scene space. Knowing the scene-space position of at least 6 points depicted on a single

frame,[2] we can solve for the corresponding camera matrix [22, p. 181]. The initial approximate solution is obtained by a greedy repetition of these two tasks, until all scene points and cameras are assigned a value.

Then, we define the energy to be minimized as the total reprojection error

$$E(\{\mathtt{C}^i\}, \{\mathbf{p}^k\}) = \sum_{i,k} w(i,k) \|\hat{\mathbf{f}}^{i,k} - \pi(\mathtt{C}^i \mathbf{p}^k)\|,$$

where the weighting function $w(i,k)$ is zero if the corresponding screen-space position $\hat{\mathbf{f}}^{i,k}$ is not known. Depending on parametrization of the cameras, their internal calibration can be minimized per-frame or for a single optimum throughout the whole sequence. Note that such non-linear refinement can be performed earlier on an unfinished approximate solution, if the precision does not seem sufficient.

C Directory structure of the attached data

- The directory `bin/` contains a binary of the program built for Linux x86-64. Most libraries are however dynamically linked, so that the CGAL and PCL shared objects must be installed and available at runtime.
- `doc/install.html` and `doc/usage.html` is the user documentation of the program. The file `doc/code.html` contains developer documentation and overall notes about the structure of the code.
- `extern/` contains the less common external libraries that are needed to compile and run the program. Furthermore, this directory contains the current source code of the program Blender 2.68.
- `src/` contains the whole source code of the program. Apart from it, a Python script `src/io_export_tracks.py` is provided that serves for exporting the camera calibration conveniently from Blender as input to our program. Further files for testing of the source code are provided in `src/test/`.
- The directory `test/` contains data for testing of the program. Each testing set consists of a video sequence, a configuration file for our program, the Blender file used for tracking and optionally a suitable initial mesh estimate. The names of files within a set differ only by their extension.
- Finally, the file `thesis.pdf` is the electronic version of this document.

[2] Fewer points may be sufficient if we impose a priori constraints. For example, we can assume that the internal calibration does not change.