



Lukas Trötzmüller, BSc

A Pipeline Approach for Viewpoint Interpolation of Panoramic Images

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to **Graz University of Technology**

Supervisors:

Dipl.-Math. Dr.techn. Torsten Ullrich

Dipl.-Ing. Ulrich Krispel

Institute of Computer Graphics and Knowledge Visualization

Graz, February 2016

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____
Date Signature

Eidesstattliche Erklärung¹

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____
Datum Unterschrift

¹Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

A main task in computer graphics is to display real scenes in a virtual environment. There exists a variety of approaches. In this thesis, a method is presented where images can be acquired using standard cameras, and new views are generated using image-based rendering.

This requires several steps, which are dependent on each other. Therefore, a complete pipeline is presented, from image acquisition to rendering.

Input images, taken with standard cameras, are stitched to panoramic images. The images are registered in 3D space to establish their location relative to each other. Using existing research, correspondences between pixels are determined (“optical flow”). New views are generated via an interpolation scheme, which takes existing images and renders new views of the scene. The interpolation scheme uses the optical flow and is based on a combination of forwards and backwards warping. It takes into account occlusions, disocclusions, and the epipolar geometry of panoramic images. Results are enhanced by generating a sparse pointcloud using structure-from-motion (SFM) tools, and using that as input for the optical flow algorithm.

The benefits of my system are low-cost acquisition, automatic registration of input images in 3D space, and a fully automatic pipeline requiring no user input.

The main contribution is the interpolation approach, which combines several existing ideas. Furthermore, a novel scheme is presented for taking into account epipolar geometry of panoramic images during interpolation. This scheme works well even if input images violate these geometric assumptions.

Kurzfassung

Eine wichtige Aufgabe in der Computergrafik ist die Darstellung von realen Szenen innerhalb einer virtuellen Umgebung. Dazu gibt es eine Reihe von Herangehensweisen. In dieser Arbeit wird eine Herangehensweise präsentiert, die es ermöglicht, die reale Szene mit normalen Kameras aufzunehmen. Mithilfe von Rendering-Verfahren, die rein bildbasiert arbeiten (Image-Based Rendering, IBR) werden neue Ansichten der Szene generiert.

Unser Verfahren erfordert eine Reihe von Verarbeitungsschritten, die aufeinander aufbauen. Die Arbeit beschreibt eine vollständige Pipeline, von der Aufnahme der Bilder bis hin zum Rendern von Ausgabebildern.

Die Eingabebilder werden mit normalen Kameras aufgenommen, und automatisch zu Panorama-Bildern zusammengefügt. Diese Panoramabilder werden im 3D-Raum lokalisiert. Mittels Verfahren aus der Computer Vision können Korrespondenzen zwischen Pixeln ermittelt werden ("Optischer Fluss"). Mithilfe dieser Korrespondenzen kann Interpolation durchgeführt und neue Bilder erzeugt werden. Dieses Interpolations-Schema basiert auf einer Kombination aus Vorwärts- und Rückwärts-Morphing. Das hier vorgestellte verfahren berücksichtigt Occlusions, Disocclusions, und die Epipolar-Geometrie der panoramischen Bilder. Zusätzlich wird mithilfe von structure-from-motion (SFM) eine Punktwolke erzeugt, die dann als Initialisierung für den optischen Fluss verwendet wird.

Die Vorteile meines Systems sind schnelle und kostengünstige Aufnahme realer Szenen, automatische Lokalisierung der Aufnahmen im 3D-Raum, und eine vollautomatische Verarbeitung.

Der wichtigste wissenschaftliche Beitrag ist das Interpolationsverfahren, mit dem aus den zuvor aufgenommenen Bildern neue 3D-Ansichten erstellt werden. Für dieses Verfahren wurden mehrere bestehende Ideen miteinander verknüpft. Außerdem wird eine neue Methode zur Berücksichtigung der Epipolargeometrie während der Interpolation vorgestellt. Diese Methode funktioniert auch dann gut, wenn gewisse Annahmen bezüglich der Geometrie der Eingabebilder verletzt werden.

Acknowledgements

This thesis would not have been possible without the following people.

First, I would like to thank Dipl.-Ing. Ulrich Krispel and Dipl.-Math. Dr.techn. Torsten Ullrich for their guidance and help while working on this thesis. Their research suggestions were very valuable, and their feedback insightful.

I would also like to thank Dipl.-Ing. Dr.techn. Clemens Arth from the Institute for Computer Graphics and Vision (ICG), who generously provided a panoramic camera. This allowed to capture large datasets quickly.

My thanks goes to the IT administrators of the Institute for Computer Graphics and Knowledge Visualization (CGV), Mr. Lars Schimmer and Mr. Wolfgang Scheicher, for providing hardware to run some of my calculations.

Finally, I would like to thank my family and friends, who always encouraged and motivated me.

Contents

Abstract	iii
Acknowledgements	vii
1. Introduction	1
1.1. Problem Statement	1
1.2. Related Work	3
1.2.1. Examples of Pipelines	5
1.3. My Approach	6
2. Pipeline Overview	7
2.1. Overview	7
2.2. Conventions	8
2.2.1. Representing 3D Rotation	8
2.2.2. Representing Direction	11
2.2.3. Coordinate Systems	11
2.3. Panoramic Image Storage	13
2.3.1. A Simple Model of Photography	13
2.3.2. Storage of Light Information	14
2.3.3. Choice of Projection for this Thesis	18
3. Image Acquisition	19
3.1. Using a Standard Camera	20
3.2. Stitching Standard Images	21
3.2.1. A Brief Overview of Automated Stitching	21
3.2.2. Used Software	22
3.3. Using a Panoramic Camera	22
3.4. Evaluation	24
4. Registration and Alignment	25
4.1. Problem Statement	25
4.2. Implementation	26
4.2.1. Registration with Standard Images	26

Contents

4.2.2.	Registration with Panoramic Input Images	26
4.3.	Alignment	28
4.3.1.	Alignment with Standard Images	28
4.3.2.	Alignment with Panoramic Images	30
5.	Interpolation	31
5.1.	Terminology and Notation	31
5.2.	Related Work	33
5.3.	Optical Flow	35
5.3.1.	Introduction	36
5.3.2.	Optical Flow Algorithm Selection	36
5.3.3.	Wrapping Considerations	38
5.4.	Our Approach	39
5.4.1.	Overview	39
5.4.2.	Optical Flow Field Generation	45
5.4.3.	Effective Flow Calculation	45
5.4.4.	Inversion of Flow and Occlusion Handling	46
5.4.5.	Disocclusion Detection and Hole-Filling	46
5.4.6.	Sampling of Input Images to generate Intermediate Images	49
5.4.7.	Blending	50
5.4.8.	Improved Warping using Epipolar Constraints	50
5.4.9.	Limitations	54
6.	Evaluation	55
6.1.	Datasets and Challenges	55
6.2.	Results	56
6.2.1.	Influence of Distance between Original Viewpoints	57
6.2.2.	Disocclusion Handling	57
6.2.3.	Influence of the Registration System	60
6.2.4.	Comparison with Geometry-Based Approach	61
6.3.	Performance Characteristics	65
7.	Conclusion	69
7.1.	Advantages, Disadvantages and Applications	69
7.2.	Future Work	70
7.3.	Benefits and Contribution	71
	Appendix A. File Formats, Directory Structure, Interfaces	75
A.1.	Dataset Directory and Input Images	75
A.1.1.	Conventions	79
A.2.	Global Configuration	80

A.3. Using different Optical Flow Algorithms	81
A.3.1. Option 1: Place files into the “flow” folder	81
A.3.2. Option 2: Replace DeepFlow	81
A.4. Using different Registration tools	82
A.5. Using different Stitching tools	82
Appendix B. How To Use	83
B.1. Parameters for the “prepare” Mode	84
B.2. Parameters for the “calc_flow” Mode	84
B.3. Parameters for the “interpolate” Mode	84
Appendix C. How To Build	87
C.1. Mandatory Configuration and System Requirements	87
C.2. On Microsoft Windows	87
C.3. Porting to other platforms	88
Appendix D. Used Components, Licenses	89
Bibliography	91

1. Introduction

This chapter gives an introduction to the topic. First, the problem of displaying real scenes inside a virtual environment is examined. Then, some existing work on the topic is presented. Finally, we will have a brief look at our solution and how it compares with other approaches.

1.1. Problem Statement

In computer graphics, it is often desired to display real scenes inside a virtual environment. This has a wide variety of applications. For example, in visual effects for movies, real scenes are augmented with computer-generated objects. Interactive simulations, on the other hand, require free navigation through a real scene with high visual fidelity. In real estate and tourism, users should be able to navigate real places from their own computer.

There exists a variety of different approaches to solve this problem. Most approaches can be summarized in three steps:

1. **Data Acquisition:** How to record raw data from a real scene.
2. **Pre-Processing to an intermediate representation:** Usually, raw data will not be used directly. Instead, it is pre-processed and stored in a specific format that lends itself to efficient rendering.
3. **Rendering:** How to generate new images from the intermediate representation.

These three steps form a pipeline: The output of one phase is used as input for the next. This is illustrated in Figure 1.1. The example images shown in Figure 1.1 refer to a simple manual geometry construction approach, which works as follows: The real scene is photographed from multiple angles (“Data Acquisition”). From these reference photographs, geometry is constructed manually, and textures are extracted (“Pre-Processing”). This geometry and texture information is stored (“intermediate representation”). It can then be used to generate new images (“Rendering”).

1. Introduction

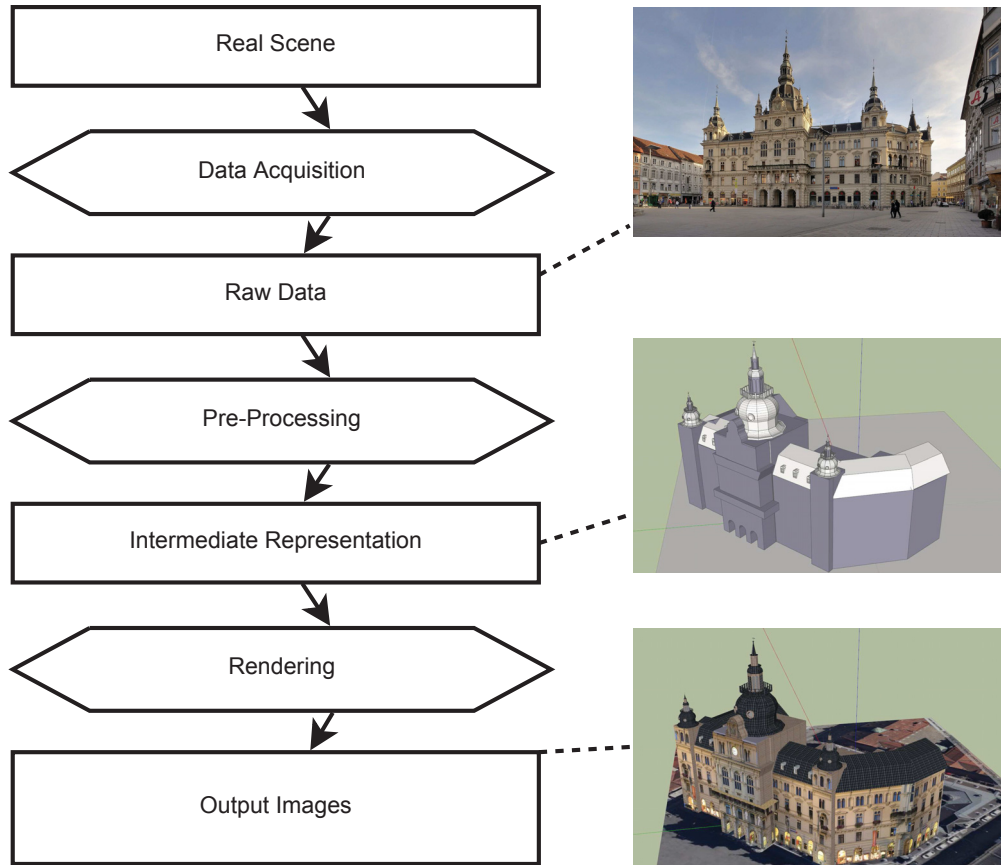


Figure 1.1.: A generalized pipeline for displaying real scenes in a virtual environment. The example images show a manual geometry construction based approach. Rectangles represent entities or data items, hexagons represent processing steps. From a real scene, raw data is captured. After some pre-processing, we arrive at an intermediate representation. The rendering process generates output images from the intermediate representation. ¹

¹ Image Credits: Photograph of Grazer Rathaus by Taxiarchos228, published at https://de.wikipedia.org/wiki/Grazer_Rathaus#/media/File:Graz_-_Rathaus2.jpg, used under CC BY 3.0 (<http://creativecommons.org/licenses/by/3.0/>). 3D Model of Grazer Rathaus, created by Gerhard A., published at <https://3dwarehouse.sketchup.com/model.html?id=a0d586afe6334a9178695d69e90202d>, used under the 3D Warehouse General Model License (<https://3dwarehouse.sketchup.com/tos.html#license>), rendered using the "SketchUp Make" software.

The choice of intermediate representation depends on the rendering approach used. There are two main paradigms for rendering: geometry-based and image-based rendering [54].

In geometry-based rendering, the intermediate representation consists of geometry and texture data. New images are generated using standard 3D rendering techniques. Data acquisition can be done in a number of ways: Geometry can be modelled by hand, or reconstructed automatically. Automatic reconstruction techniques are described by Bourke et al. [4] and Goesele et al. [18], among others.

On the other hand, in image-based rendering, output images are created from the input images, without explicit geometry reconstruction. In this case, the intermediate representation consists of images, which may be processed and augmented with additional information.

A variety of data acquisition devices may be used for both approaches, some of which are mentioned in Chapter 3.

When comparing different approaches, major considerations include rendering performance, storage requirements for the intermediate representation, and the amount of manual labour needed for data acquisition. For example, In the visual effects industry, detailed geometry for scenes is often reconstructed manually, and combined with high-resolution textures. This leads to high visual fidelity, but is very cost-intensive and requires a lot of processing time for rendering. On the other hand, for interactive online walkthroughs in real estate, the requirements are different. Data acquisition needs to be done quickly and cost-efficiently, intermediate representation data needs to be small, and real-time rendering is desirable.

1.2. Related Work

This section shows some previous work in implementing image-based rendering pipelines. Only implementations of complete pipelines are mentioned here. Additional work relating to some of the individual steps is listed in the corresponding chapters.

Yang & Crawfis implemented the “Rail-Track Viewer” [69], a system for efficient real-time navigation through virtual scenes. Instead of rendering the scene directly, they pre-render specific key points along a path. For each key point, a

1. Introduction

panoramic image with associated depth information is rendered. They reconstruct a simple mesh from the depth information, and use that to interpolate new views. This allows the user to move along a 1D path. However, their system does not allow the visualization of real scenes, and movement is restricted to that one-dimensional path.

Siu et al. presented a novel architecture for an image-based rendering pipeline [56]. They make several new contributions, including an architecture to unify several previous approaches and recovery of geometry proxies for outdoor scenes. They also present a rendering scheme which takes into account the expected distortions during interpolation, and selects the two most suitable input images to sample from.

Starck et al. created the “free-viewpoint video renderer” [58]. Like Siu et al., they create geometry proxies for the scene, however their geometry proxy changes over time. Rendering is performed on the GPU, enabling interactive frame-rates with high-resolution video streams.

Lipski et al. solve the problem of free-viewpoint video using a different approach (“virtual video camera” [32]). Instead of constructing a geometry proxy, their method is purely image-based. First, they compute the optical flow between pairs of images. Optical flow refers to the problem of finding per-pixel correspondences in a pair of images. Their method is optimized for temporal upsampling of video, therefore they use a special-purpose optical flow algorithm previously developed by Stich et al. [60]. That algorithm is optimized for small displacements, high performance, and perceptually good results. To generate new views, they move the pixels of the original image in the direction of the optical flow field (“forward warping”). The rendering is implemented on the GPU.

Zhao et al. [70] present a method based on warping a 2-dimensional triangle mesh. They use panoramic images as input. From those images, position and orientation are estimated. Then, feature points are detected. Between these feature points, a 2D triangle mesh is constructed, and warped to the correct location. All image pixels within a particular triangle follow that triangle’s motion. This stands in contrast to optical-flow-based approaches, where each pixel has an individual motion vector.

In Kolhatkar’s thesis [28], another approach based on optical flow is presented. This approach is most similar to the present thesis, but there are several differences. This thesis includes stitching and registration, whereas Kolhatkar focusses on the interpolation. The pipeline presented here uses external optical

flow software, while Kolhatkar implements a custom algorithm. Finally, the interpolation scheme, while based on optical flow in both cases, is different.

Stich et al. interpolate directly in image space, with no explicit geometry reconstruction [60]. In contrast to Starck et al., which used video streams, they work on a small set of static images that are taken at different times and different locations. Using homographies combined with an image deformation model, they achieve convincing interpolation. They also mention that on dynamic scenes, their approach outperforms optical-flow-based methods.

The ideas discussed in this thesis are closely related to new television technologies, namely 3DTV and FTV. 3DTV refers to stereoscopic image material, where two images are recorded and displayed, for the left and right eye. FTV, on the other hand, allows the viewer free viewpoint and view direction selection (within a certain range). Kubota et al, in their article, give an overview about the challenges of 3DTV and FTV, and emphasize the importance of considering the entire pipeline [30]. Many of the topics addressed in this thesis are covered in that article: The relation between capturing, preprocessing and interpolation; various capture technologies, and the distinction between geometry- and image-based rendering. The article also covers 3D video coding technologies.

1.2.1. Examples of Pipelines

Two well-known examples of image-based interpolation pipelines are Google Street View [21], and Mapillary [63].

Google Street View uses panoramic image material, taken from company-operated cars on streets around the world. The images are localized using initialization data based on GPS, with an image-based refinement scheme [27]. Their web viewer enables image-based navigation through the streets. However, their interpolation scheme is relatively crude and can give only a rough approximation of perspective.

Mapillary allows anyone to upload their own street-level images, and localizes these images in 3D space, enabling navigation through a large collection of photographs. Their interpolation scheme is more advanced than Street View, and provides more realistic transitions between images. However, the source images are not panoramic, thereby restricting the field of view.

1. Introduction

1.3. My Approach

The goal for this thesis is to implement a flexible and extensible pipeline for image-based rendering of panoramic photographs.

A decision was made to use image-based rendering (IBR) and avoid geometry-based rendering. The goal was to see how well IBR performs in practice. An advantage of IBR is predictable performance and simplicity in the algorithm. By using only images as input material, it is possible to capture scenes using a simple camera, and there is no need for expensive or heavy equipment.

Throughout the entire pipeline, panoramic images are used instead of standard photographs. Panoramas have unlimited field-of-view. This allows the viewer to rotate freely, creating an immersive effect. Panoramic images can be taken using a standard camera or special panoramic cameras. Output images can be rendered at any location between the original camera positions. This provides the user with freedom in both viewpoint and view direction.

Position and orientation of each panorama is determined from the images.

The system was implemented and evaluated. Evaluation was done with respect to rendering artifacts, output quality and performance measures. Furthermore, experiences with image acquisition are described and the pipeline is compared with a geometry-based rendering approach.

Various external software packages and libraries were used for some of the steps, and components can be exchanged individually.

2. Pipeline Overview

In this chapter, we will have a look at the structure of our pipeline. Then, coordinate systems and conventions will be discussed. Lastly, we will look at how panoramic images are stored in our system.

2.1. Overview

The process starts from a collection of photographs. The pipeline accepts two types of input images: Standard images taken with a standard camera, or 360° panoramic images. Standard images are automatically stitched into panoramic images. The **image acquisition** process is explained in Chapter 3, and the stitching in Section 3.2.

The **registration** subsystem (Chapter 4) determines 3D positions (up to a scale factor) and relative orientations for all panoramas. It may also generate sparse 3D points with visibility information. This information improves rendering quality, because the optical flow software can more precisely estimate the true motion of scene points later.

Using the registration information, all panoramas are aligned relative to each other (**alignment**, Section 4.3). This alignment is necessary to get optimal results out of the next step.

The **optical flow generation** step determines pixel-wise correspondences between pairs of images. This is explained in Section 5.3. By knowing these correspondences, pixels can be moved to simulate intermediate positions during the next step.

Finally, using the intermediate results generated so far, new viewpoints can be rendered (**interpolation**, Chapter 5).

2. Pipeline Overview

Figure 2.1 gives an overview over the pipeline. Input, output and intermediate data is stored as follows:

- **Input Images** as JPG.
- **Aligned Panoramic Images** as PNG files.
- **Position and Orientation for each Panorama, Sparse 3D Points** are written to a single JSON file per dataset.
- **Optical Flow Files** in the .flo file format, which is a de facto standard for storage of optical flow fields.

More details can be found in Appendix A.

2.2. Conventions

This section explains the mathematical conventions used in this thesis. All data retrieved from external tools is converted into this representation.

2.2.1. Representing 3D Rotation

In this thesis, we use different coordinate systems that are oriented at angles relative to each other. Therefore, we first need to discuss how to represent rotations mathematically. In this thesis, two representations are used to represent full 3D rotations: **yaw-pitch-roll**, and **quaternions**.

Yaw-pitch-roll is illustrated in Figure 2.2. This representation has the advantage of being intuitive to understand, however they have some properties that may make calculations difficult (for example, they suffer from singularities). In the literature, this representation is called “Tait-Bryan Angles” or “Euler Angles”, depending on the specific convention used.

Our Yaw-Pitch-Roll representation works as follows: We first apply yaw (a rotation around the y-axis, measured from the positive z-axis), then pitch (rotation around the x-axis, measured from the positive z-axis), and then roll (rotation around the z-axis, measured from the positive x-axis). The rotations are applied intrinsically, i.e. they are applied to the rotated coordinate system. We define the yaw range to be -180° to $+180^\circ$, pitch from -90° to $+90^\circ$, and roll from -180° to $+180^\circ$.

Quaternions are an extension of imaginary numbers that are well suited to describe 3D rotations. Quaternions are represented by a set of four numbers

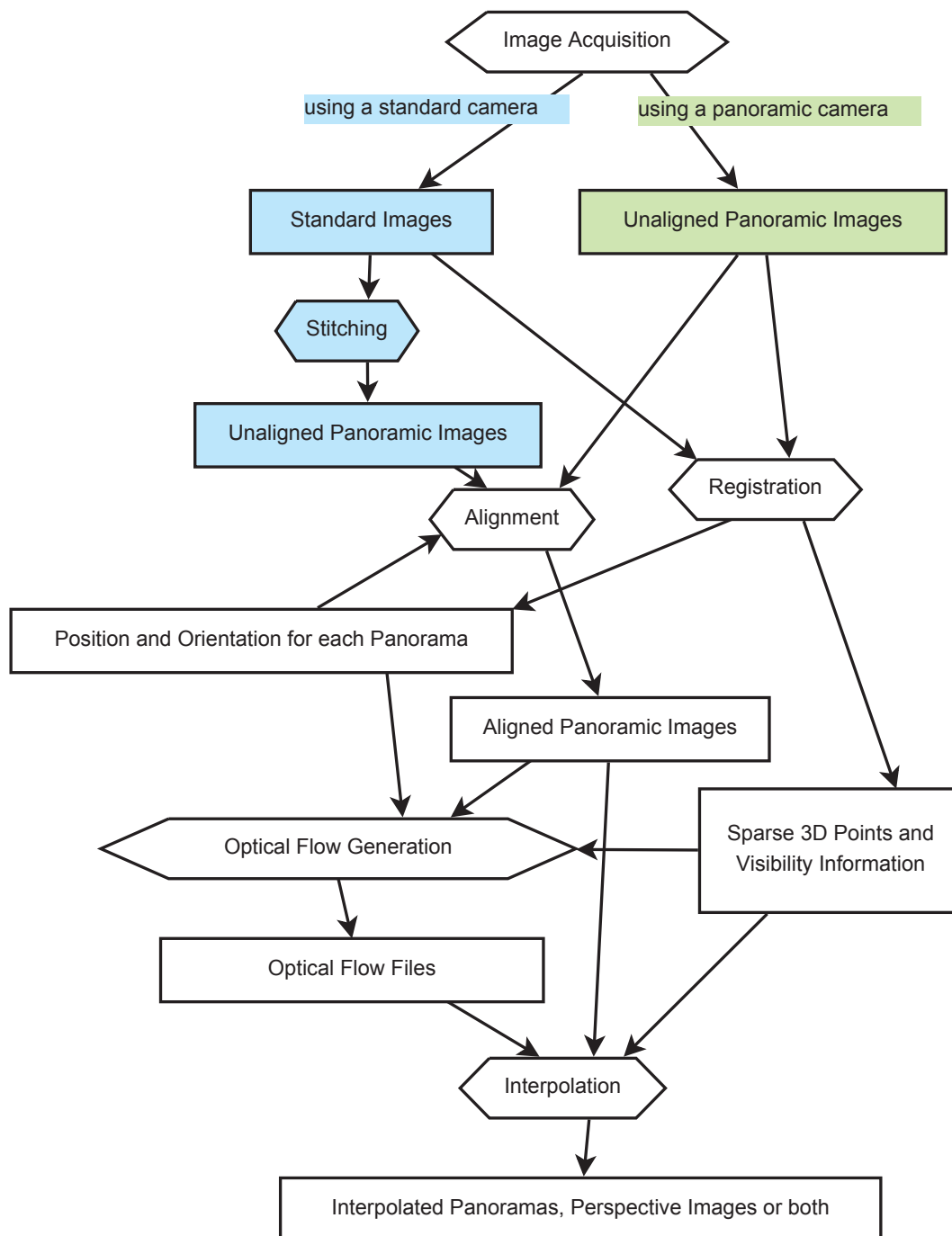


Figure 2.1.: Overview of the pipeline. Rectangles represent entities or data items, hexagons represent processing steps. The process starts with standard images, which are stitched together, or panoramic images, which are used directly. In the registration step, orientation and position of each panorama is calculated. In the alignment step, their orientation in world space is equalized. From that information, optical flow is calculated new panoramas can be generated by interpolation.

2. Pipeline Overview

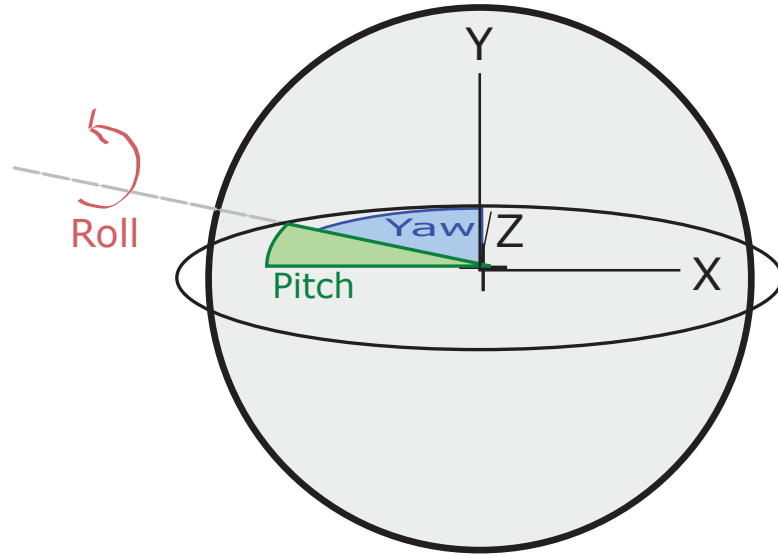


Figure 2.2.: The yaw-pitch-roll representation of rotation.

x, y, z, w . They are ideal for efficient computation. For example, to chain two rotations together, one only needs to multiply the respective quaternions. This multiplication is performed using the so-called Hamilton product, which will be written in the thesis like this: $q_1 \circ q_2$. For more details, the reader may refer to the literature [61]. An introduction to quaternions and their application to 3D geometry is given by Goldman [20]. A concise summary can be found in Schwab's article [52].

To rotate a vector $v = (v_x, v_y, v_z)$ by a quaternion $q = (q_x, q_y, q_z, q_w)$, we define a quaternion q_p :

$$q_p = (v_x, v_y, v_z, 0) \quad (2.1)$$

Then, we calculate:

$$p' = q \circ q_p \circ q^{-1} \quad (2.2)$$

The rotated vector is given by $v' = (p'_x, p'_y, p'_z)$.

Conversion between yaw-pitch-roll, quaternions, and other orientations is explained in an article by Diebel [10].

2.2.2. Representing Direction

To represent the direction of a point relative to a specific viewpoint, we use **direction vectors** and **normalized spherical coordinates**.

We define the **(normalized) direction vector** v of scene point x as seen from camera position c :

$$v = \frac{x - c}{\|x - c\|} \text{ where } \|\cdot\| : \text{Euclidean norm} \quad (2.3)$$

We can represent the same direction using **normalized spherical coordinates**. These are specified using two values, ϕ (phi) and θ (theta). ϕ has the same meaning as yaw, and θ means pitch, as shown in Figure 2.2. To convert spherical coordinates into a direction vector, yaw and pitch are applied to the z-axis vector $(0, 0, 1)$ using the same convention as for our yaw-pitch-roll representation. Obviously, since we apply the rotation to the z-axis, roll values would have no effect.

To convert a direction vector into normalized spherical coordinates, the following formula can be used:

$$\begin{aligned} \text{Direction vector } v &= (x, y, z) \\ \phi &= \text{atan2}(z, x) - \frac{\pi}{2} \\ \theta &= \arcsin(y) \end{aligned} \quad (2.4)$$

2.2.3. Coordinate Systems

We distinguish three coordinate systems: Global, local and image.

The global coordinate system is used to establish the position of viewpoints, and the position of sparse 3D points which are optionally reconstructed.

The local coordinate systems are centered on each viewpoint, at a specific rotation relative to the global coordinate systems. This rotation is identical among all viewpoints because we align the panoramic images (Section 4.3). This alignment is necessary because we send the image files to external “optical flow” tools, which work much better with alligned images. The relationship

2. Pipeline Overview

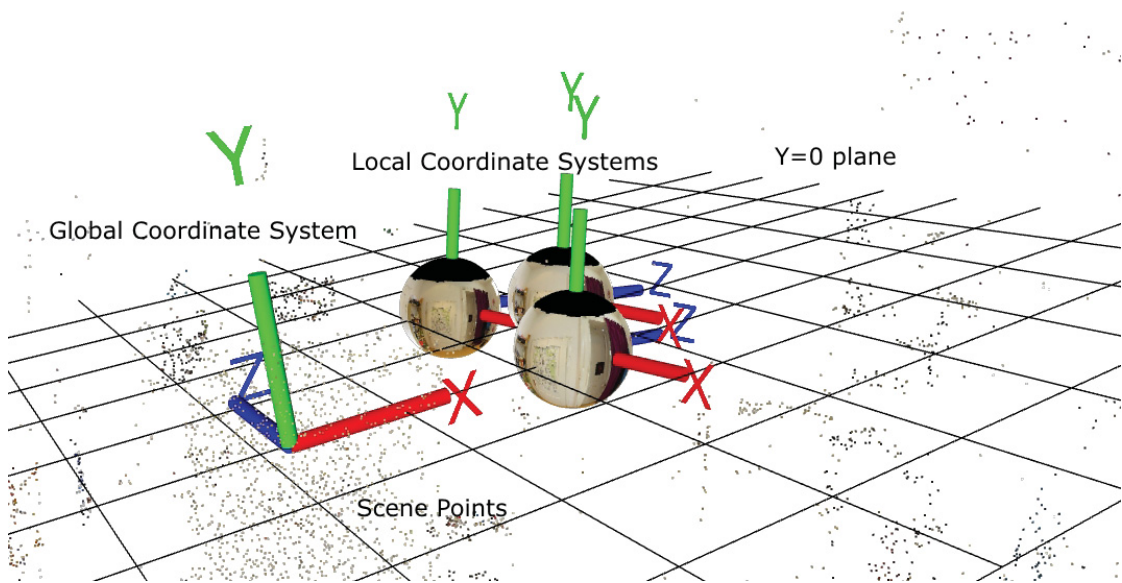


Figure 2.3.: Example of three local coordinate systems, centered at the corresponding viewpoints, in relation to the global coordinate system. Also shown is a cloud of sparse 3D points from our scene.

between global and local coordinate systems is illustrated in Figure 2.3, and defined mathematically below.

The pipeline uses left-handed coordinate systems.

The image coordinate system defines the mapping between angles of incident light rays and their position in the panoramic image. This mapping depends on the choice of projection type. For our implementation, equirectangular projection is used and this conversion is illustrated in Figure 2.7.

Transformations

Scene Point to Direction Vector

Let x be a 3D scene point in global coordinates. Let the camera position be c_1 . For most purposes, we only need to know the direction in which a scene point appears, but not its distance. Therefore, we will only consider the **normalized direction vector** to our scene point, expressed in global coordinates v_x :

$$v_x = \frac{x - c}{\|x - c\|} \text{ where } \|\cdot\| : \text{Euclidean norm} \quad (2.5)$$

Direction Vector to Local Direction Vector

Let q_{I_1} be the orientation of the local coordinate system for this camera, expressed as quaternion. To transform the direction vector in the local coordinate system, we make use of the properties of quaternions established in Section 2.2.1.

$$\begin{aligned} &\text{Let } q_{v_x} \text{ be } v_x \text{ converted into a quaternion according to Formula 2.1} \\ v_{x, \text{ local}} &= q_{I_1} \circ q_{v_x} \circ q_{I_1}^{-1} \end{aligned} \quad (2.6)$$

Local Direction Vector into Normalized Spherical Coordinates

We then convert this local direction vector into spherical coordinates (ϕ, θ) according to Formula 2.4.

Normalized Spherical Coordinates to Image Coordinates

These spherical coordinates can be converted into panoramic image coordinates. This is explained in Section 2.3, and more specifically in Formula 2.7.

2.3. Panoramic Image Storage

In this section, we will investigate how light information from our scene is captured and stored.

2.3.1. A Simple Model of Photography

When describing photography, it is helpful to imagine the following model, which is adapted from Hartley and Zisserman's classic book [23]. We choose a single, infinitely small viewpoint, the "center of projection" or "scanning position". We capture all light rays arriving at this point, with their color information and the direction they are coming from.

2. Pipeline Overview

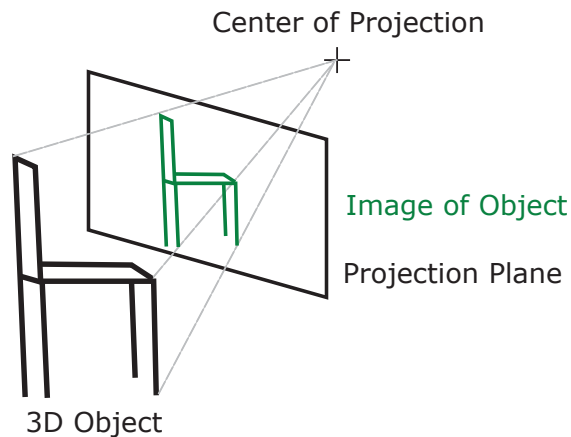


Figure 2.4.: Standard Photography: View rays are captured on a rectangular image plane, and only a portion of the scene is captured.

To capture that light information, we need to define a 2-dimensional shape onto which the image is projected (the “projection surface”). For example, Figure 2.4 shows a model of a standard camera, with projection onto a plane.

Panoramic images and standard images differ in their field of view. In panoramic photography, light rays arriving from all directions are captured, with a field-of-view of 360° horizontally and 180° vertically. On the other hand, with standard images, we only capture a subset of light rays, with a field-of-view that covers less than half of the full sphere.

2.3.2. Storage of Light Information

We saw in the previous section how light information arrives at a center of projection, and is projected onto a surface. Standard photos are usually stored as rectangular grids of pixels, so-called bitmaps. Therefore, we need to define not only the shape of our projection surface, but also a method to store light information from that surface in a rectangular bitmap.

With a standard digital camera, view rays land on a rectangular sensor. Since the bitmap and the projection surface are both rectangular, the light information arriving at the sensor may be stored directly: each sensor pixel is stored in one bitmap pixel. This is illustrated in Figure 2.4.

With panoramic images, on the other hand, the projection surface can be imagined as a sphere. This is illustrated in Figure 2.5. However, to store the

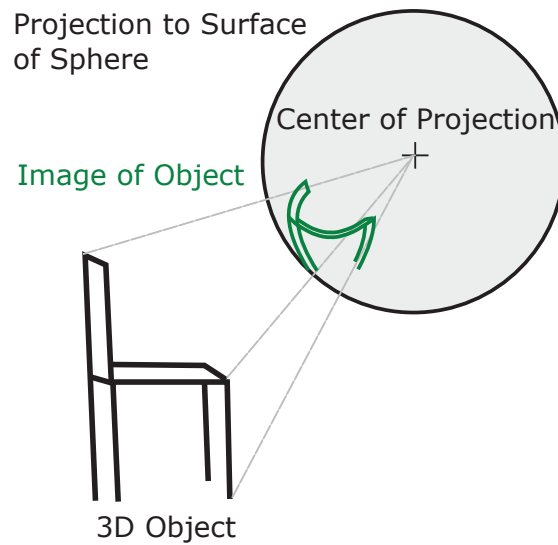


Figure 2.5.: Panoramic Photography: The entire scene is captured. This includes (theoretically) all light rays arriving at the center of projection, from all directions.

images as bitmaps, we need to define how to store points on the sphere in a rectangular image.

This problem has been well-studied by cartographers, trying to represent a spherical earth in a rectangular map. A comprehensive overview from the cartographers's perspective can be found in [57].

Hassanat [24] investigates several projection types and explains their properties in relation to virtual reality applications. For this thesis, two projection types were considered: Cubemaps and equirectangular projection. In a cubemap, the panoramic sphere is replaced by a panoramic cube, with 6 images with a 90° field of view each (see Figure 2.6). These images (left, right, front, back, up, down) can then be packed into one image, at the cost of wasted space (Figure 2.8b).

In equirectangular projection, angles are linearly mapped to pixels. For each view ray, we take its normalized spherical coordinates, expressed as a tuple (ϕ, θ) (phi and theta, corresponding to yaw and pitch). This is shown in Figure 2.8b. The inclination (-90° to $+90^\circ$) is mapped to the vertical position in the image. The radial angle (-180° to $+180^\circ$) is mapped to the horizontal position. The center of the image corresponds to spherical angles of 0° inclination and 0° radial. This can be expressed mathematically as follows:

2. Pipeline Overview

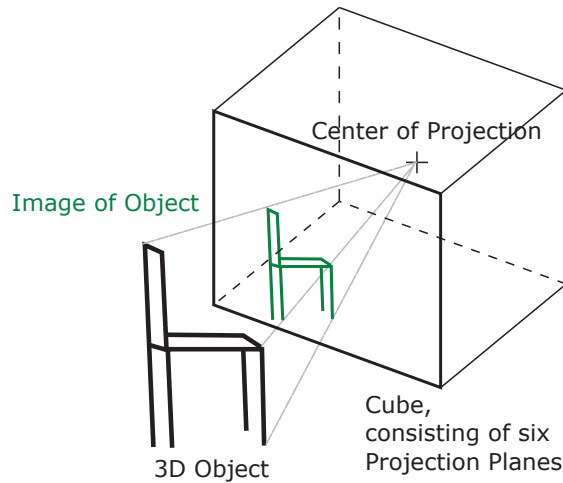


Figure 2.6.: Cubemaps can be imagined as the projection of view rays onto a cube. The sphere from Figure 2.5 is replaced by a cube, onto whose sides the rays are projected.

let (ϕ, θ) : normalized spherical coordinates of view ray

let (w, h) : Image width and height, respectively

calculate image coordinates (x, y) as follows:

$$\begin{aligned} x &= \frac{(\phi + \pi) \cdot w}{2\pi} \\ y &= \left(\theta + \frac{\pi}{2}\right) \cdot \frac{h}{\pi} \end{aligned} \tag{2.7}$$

Equirectangular projection has the problem of a very non-uniform resolution distribution: The poles are stored with much higher resolution than the equator. Cubemaps offer more uniform resolution, but they have discontinuities between the up/down and left/right/back images. These discontinuities present a problem when used with optical flow algorithms, as described in Section 5.3. Also, cubemaps waste half of the space in the bitmap. Therefore, equirectangular projection was chosen for our implementation.

An example of one panorama, displayed as cubemap and in equirectangular projection, is shown in Figure 2.8.

2.3. Panoramic Image Storage

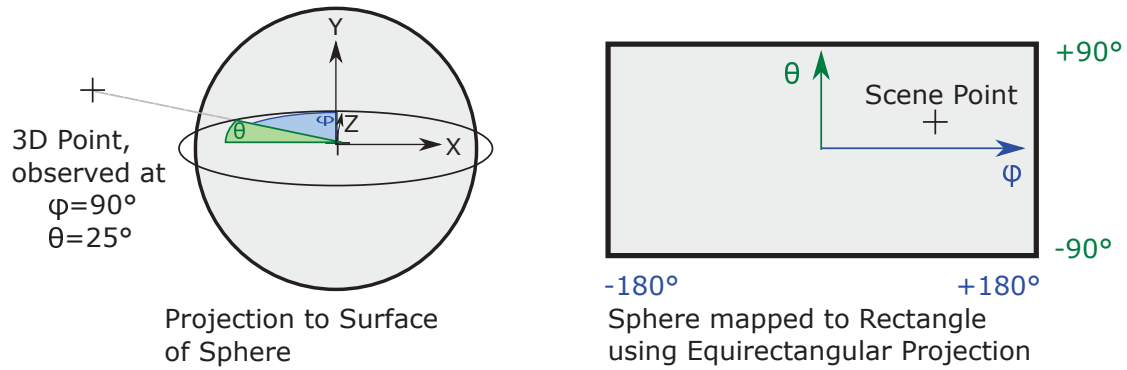


Figure 2.7.: Equirectangular Projection. For a scene point, we calculate its normalized spherical coordinates, expressed in azimuthal angle ϕ and polar angle θ . These angles are mapped to the horizontal and vertical axis in the image.



Figure 2.8.: Comparing cubemaps (a) and equirectangular projection (b). Note that the very top and bottom of the sphere were not photographed in this case, causing two holes in (a) and white borders in (b).

2. Pipeline Overview

2.3.3. Choice of Projection for this Thesis

Equirectangular projection has been selected for our implementation, for reasons presented in Section 2.3.2. Equirectangular projection is also used for most of the explanations and images in this thesis.

However, most algorithms presented are agnostic of the specific projection type used. This is because the panoramic images are only a particular representation of the light rays arriving at the viewpoint, and it is always possible to extract light rays (angle and color) from the panoramic image. Therefore, by the term “panoramic image”, we refer to a representation of the light rays, without implying a specific type of projection.

When a specific type of projection is referred to, we will use the terms “equirectangular panoramic image” or “panoramic image in equirectangular projection”.

3. Image Acquisition

In Chapter 1, we saw how the pipeline is structured, and that panoramic images are used as input data. There are several options to shoot these images. For this thesis, two approaches were evaluated:

1. Using a standard camera pointed in different directions, shooting small-field-of-view images. These images are then stitched together into a single spherical field-of-view image.
2. Using a panoramic camera, which contains several smaller cameras mounted at a fixed angle relative to each other. The images still need to be stitched, but this is done by the camera software.

These two approaches were evaluated as part of this thesis, however there are other options to shoot panoramic images. Approaches **using consumer hardware** include:

- Photographing a reflective sphere using a standard camera. By photographing just one half of the sphere, more than half of the surrounding scene can be captured in a single shot.
- Using a video camera on a tripod, then slowly rotating the camera.
- Panoramic photography apps for smartphones.

All cameras mentioned so far capture light rays arriving at a single point in space, with their direction and color. There is also **special hardware** which captures more information about the scene. These cameras would enable different approaches to our problem. Two examples are:

- Depth cameras, which use laser scanning or structured light to measure distance at each pixel. One example is the Microsoft Kinect, which has been used for image-based rendering by Gupta et al. [22]. Using a depth camera would allow more precise reconstruction of scene geometry, and more accurate results during interpolation (Chapter 5).
- Lightfield cameras, which acquire information about position and direction of incoming light rays. Effectively, this captures multiple viewpoints and multiple directions simultaneously. Examples include products from Raytrix GmbH [44] and Lytro Corporation [64]. Using a lightfield, it is

3. Image Acquisition



Figure 3.1.: The setup for standard image acquisition, consisting of a Powershot S100 consumer camera (a) and a Panosaurus panoramic head (b).¹

possible to extract new viewpoints with high quality, and replicate camera characteristics like depth-of-field.

3.1. Using a Standard Camera

Test images were acquired using a Canon PowerShot S100 consumer camera. The camera is set up on a tripod using a “Panosaurus” panoramic head [47]. This is a simple mechanical device enabling the camera to pivot exactly around its sensor. The setup is shown in Figure 3.1. Then, images are taken into all directions. Using automatic feature detection and matching, these images can be stitched automatically. This image-based stitching process works well most of the time, but the scene needs to have enough distinct features, and the images need to share some of the features. The stitching process is explained in detail in Section 3.2. In scenes with few features, automatic stitching may not work. In that case, it would be necessary to measure the camera orientation during shooting for each image, and enter these values into the stitching software by hand. For optimal results, the photographer should follow these guidelines:

¹Image credits: (a) Anna16, https://commons.wikimedia.org/wiki/File:Canon_PowerShot_S100.jpg, used under Creative Commons Attribution-Share Alike 3.0 Unported. (b) Gregwired Digital, <http://gregwired.com>, used with permission.

3.2. Stitching Standard Images

- The camera should pivot exactly around its sensor. This can be achieved using special panoramic mounts, e.g. Panosaurus [47].
- Adjacent images must have some amount of overlap, to allow for automated image-based stitching.
- Ideally, the camera should be operated in manual mode. This is to avoid differences in apparent illumination between the pictures. Differences in apparent illumination might stem from varying ISO, aperture, shutter timing or in-camera postprocessing. Using manual mode will, in most cameras, prevent these variations. If the camera does not have a manual mode, the stitching software will try and correct for differences in brightness within each panorama. However, these corrections are not consistent among different panoramas, and will lead to problems in later stages of the pipeline.
- White balance must be on the same setting for the entire shoot.
- The focus should be set so that the most important objects are in focus. “Important” objects are these which are most likely to be of interest to the user in the particular application.
- Aperture should be set to small, so that most of the scene is in focus.
- For optimum picture quality, a low ISO value and the use of a remote shutter or self-timer is recommended.

3.2. Stitching Standard Images

When feeding standard images into our pipeline, they are stitched automatically. This section describes briefly how automated stitching works, and the particular software used.

3.2.1. A Brief Overview of Automated Stitching

Brown and Lowe present an overview over automated stitching, and then present their own approach [5]. Their solution is fairly sophisticated. Here, we will summarize only the basic steps.

Step 1: Feature Detection. First, invariant feature points are detected in the images. “Invariant” means that the feature detector algorithm can recognize the same feature in different images in the presence of zoom, rotation, and slight changes in illumination.

3. Image Acquisition

Step 2: Matching. Secondly, matches between the features are detected across different images.

Step 3: Image Alignment. Now, the images need to be aligned using the feature matches found previously. Because of inaccuracies in feature positions and mismatches between features, no perfect solution is possible. Instead, we need to find a solution minimizing the error in feature alignment.

Step 4: Blending. Brown and Lowe explain how even after perfect alignment, overlapping pixels in multiple images will not have the same intensity. This is because of vignette effects inside the lense, changing scene illumination, noise, and other factors. Therefore, they argue that good blending is essential for high-quality results. They use multi-band blending, an approach developed previously by Burt and Adelson [6]. Effectively, low-frequency detail are blended over a large range, whereas high-frequency detail is blended over a shorter range. This prevents small registration errors from producing blurry regions in the output image.

Brown and Lowe also present some extensions to basic image stitching. For example, their method automatically detects multiple viewpoints in the unsorted collection of input images. They also mention automatic straightening and gain compensation.

3.2.2. Used Software

Existing Open Source Software produces excellent results and no further research was required. Therefore, the Hugin package [9] was simply integrated into our pipeline.

For our datasets, Hugin produced reasonably good results. All images were aligned roughly correctly. However, approximately one out of ten input images had some small errors in their alignment, leading to seams in the stitched image. Therefore, there would be potential to improve results by integrating a different stitching package into the pipeline.

3.3. Using a Panoramic Camera

Shooting standard images requires taking between 10 and 30 images per viewpoint. This has several disadvantages:

3.3. Using a Panoramic Camera



Figure 3.2.: The Ladybug 3 setup.

- (1) The process is time-consuming.
- (2) The image-based stitching may fail if not enough feature points can be detected.
- (3) Seams between individual images will appear if there is scene motion or changes in illumination.
- (4) During the long shooting process, illumination of the scene may change significantly (due clouds, motion of the sun, etc).

Using a panoramic camera solves problems 1-3, and may alleviate (4) because the shooting process takes less time, so the motion of sun and clouds will be less apparent.

The camera used for this thesis was a Ladybug 3 360 Degree Firewire Camera [40]. It contains six small cameras which cover 80% of the full sphere at a resolution of 1600x1200 per camera. [41]

Figure 3.2 shows the setup. A tripod holds the Ladybug camera, which is connected to a laptop using a 10m FireWire cable. The camera is controlled via the laptop, running the Ladybug software.

The camera is set to capture an image every second. This makes the shooting process very efficient. The photographer moves the tripod around, and leaves it standing at each location for two seconds. Since the Ladybug does not capture the area directly underneath it, the photographer can simply knee down to

3. Image Acquisition

	Ladybug 3 (Panoramic Camera)	PowerShot S100 & Panosaurus (Standard Camera with Panoramic Mount)
Number of Images taken	1	20
Field of View per Image	360° by 144°	71° by 53°
Shooting Time	10 seconds per panorama	3 minutes per panorama
Resolution	Fixed and limited, but enough for most applications.	Flexible, depending on zoom. Very high resolution possible (using a zoom lens).
Stitching Reliability	Works always, because the exact offset of the six sensors is known.	Needs distinctive image features and overlap.
Stitching Quality	Somewhat lower due to parallax effects, if cameras do not share the same center of projection.	Very high (if it works).

Table 3.1.: Comparison of two approaches for image acquisition.

avoid being captured. Later, using the same software, the correct frames can be found and exported in equirectangular projection. Equirectangular projection is explained in Section 2.7.

3.4. Evaluation

Table 3.1 compares both approaches. My recommendation is to use a panoramic camera for large scenes, because this greatly reduces the time required. This is especially important in outdoor scenes, where illumination conditions change quickly.

For indoor scenes with controlled lighting conditions, and if high resolution is important, it is preferable to use a standard camera with a panoramic mount.

4. Registration and Alignment

In this chapter, we look at the problem of registration and alignment. We want to determine the position and orientation of the acquired panoramic images relative to one another, using only the images. This information allows to align the images relative to each other, so they all share the same orientation. Additionally, during this process, a set of 3D points is reconstructed. Those 3D points can be used to improve results later in the pipeline.

4.1. Problem Statement

In the **registration** phase, locations and orientations of the panoramic images are established. Furthermore, a set of reconstructed 3D scene points is found. This is done using only the images, with no other measurements.

This information is used in several ways:

- By determining camera rotation, panorama images can be aligned relative to each other. This is important for good results during interpolation (Chapter 5).
- By knowing the camera locations, we can find the 2, 3 or 4 closest cameras to a specific point in the global coordinate system, and use these cameras for interpolation.
- Optionally, a set of reconstructed 3D scene points is used to improve results during the interpolation process (Chapter 5).¹

¹In practice, for the VisualSFM package, 3D scene points are retrieved. For the OpenSFM package, 2D correspondences are retrieved. Both representations are stored and converted into 2D correspondences before they are used.

4. Registration and Alignment

4.2. Implementation

External structure-from-motion software is integrated into the pipeline. Structure-from-motion (SFM) refers to the problem of reconstructing 3D points and camera data from a set of images. Camera data includes position, rotation and field-of-view. The images are set in relation to each other, determining camera position and rotation, up to a scale factor. The process also delivers a loose collection of 3D scene coordinates and their color information, but no triangles or other surfaces. SFM algorithms have been successfully used to reconstruct increasingly large sets of photos, approaching a million photographs [1].

Depending on the type of input images (standard or panoramic), two different processes are used.

4.2.1. Registration with Standard Images

We have seen that standard images are stitched together to panoramic images. However, for registration, the original, non-stitched images are used. This is done to minimize the influence of small errors in the image-based stitching method, which could affect the success of the SFM process.

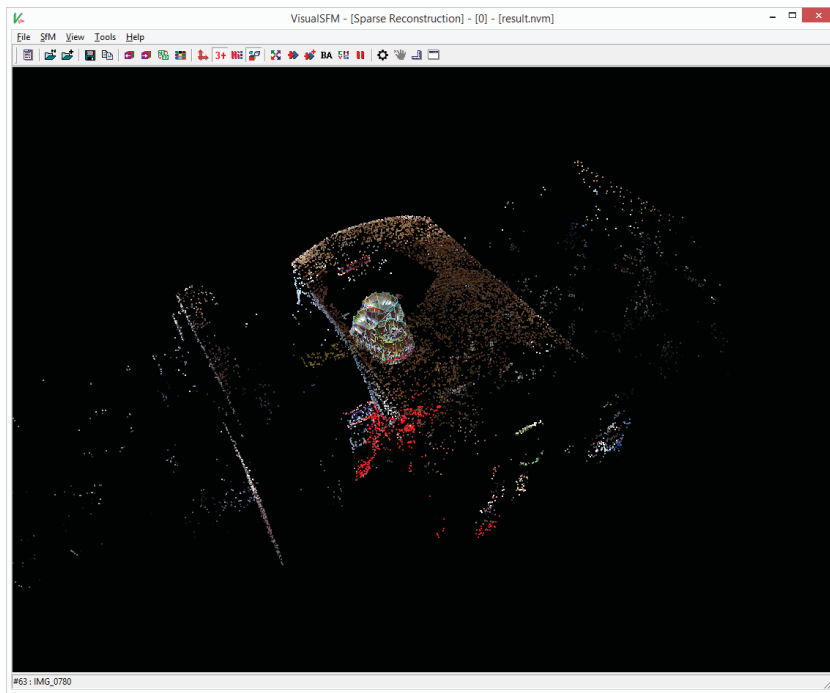
For standard images, the VisualSFM software is integrated into the pipeline [67, 68]. The software provides a graphical user interface, which can be seen in Figure 4.1a.

For registration, the original, un-stitched images are used. Therefore, the orientation of the stitched panorama can not be extracted directly from the output of VisualSFM. Instead, they must be converted. This is explained in Section 4.3.1.

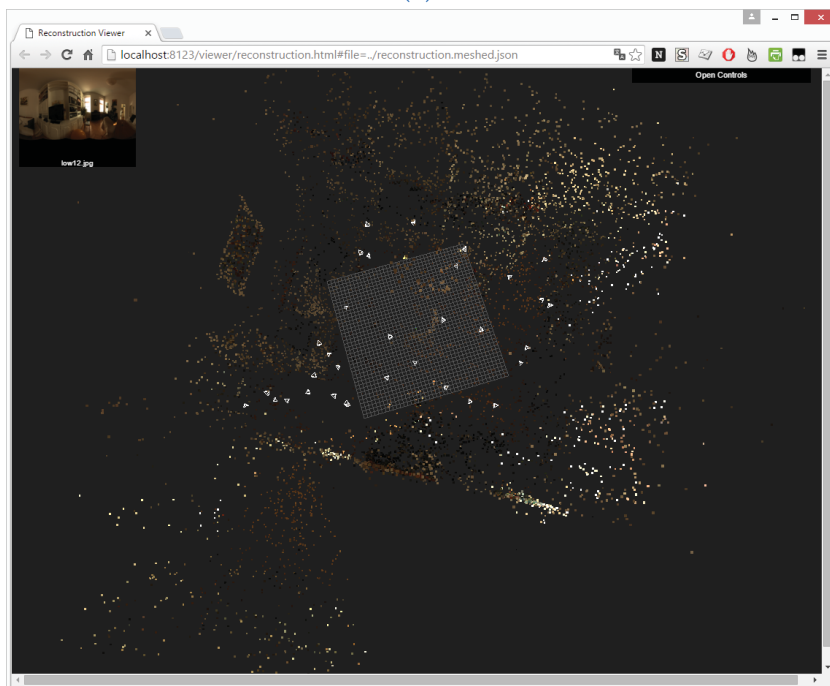
4.2.2. Registration with Panoramic Input Images

When using panoramic images, there is no stitching phase, and the images themselves are used as input for the registration. The main difficulty was finding a SFM package that allows panoramic input images. VisualSFM only supports standard images. Finally, the OpenSFM open-source package was selected [38]. It was initially created by Mapillary, a company providing crowd-sourced street-level images [63]. The software is run on the command line, and provides a web-based viewer, which can be seen in Figure 4.1b.

4.2. Implementation



(a)



(b)

Figure 4.1.: The two packages used for Registration in this thesis. (a) shows the graphical user interface of VisualSFM. The dataset “bar” is loaded, and the original (standard) input images are displayed together with a reconstructed point cloud. (b) shows the web viewer of OpenSFM. The “grabenkirche” dataset is loaded. The wireframe camera icons display the orientation of the input panoramic images. A sparse pointcloud is also shown.

4. Registration and Alignment

4.3. Alignment

We now have position and orientation for each input image. For good results, it is important to rotate the panoramas so that the same image coordinates on each panorama correspond to the same view direction in worldspace. This is necessary for the optical flow generation (Section 5.3): Optical flow algorithms perform much better when applied to the aligned images. This is because the resulting flow vectors are smaller. Large flow vectors are not a problem themselves, but they would lead to wrap-around around the edges of the equirectangular images, which we would like to minimize.

Again, two different processes are used, depending on the type of input images.

4.3.1. Alignment with Standard Images

This section is divided into two parts: First, we look at how to extract orientation values for the stitched panoramic images. (This is not straightforward, because the constituent standard images are used as input for the registration). Secondly, an extension is presented to provide tolerance against errors in registration and stitching.

Extracting orientation for stitched panoramic images

In this section, we investigate how to obtain the position and orientation of a stitched panorama, which is assembled of several constituent images. At first, the results are presented assuming correct registration information. Then, in Section 4.3.1, an extension is presented to tolerate errors.

Obtaining the position is trivial, because all constituent images will share the same center of projection. Therefore, we can use the position of any constituent image.

The orientation of a stitched panorama is obtained as follows: One of the constituent images is selected as the “anchor image”. In this section, we represent orientations as quaternions.

The symbols used are summarized in Table 4.1. We look at one stitched panoramic image I_k and want to determine its orientation in the global coordinate system. Let J_a be the anchor image for I_k . Its orientation is described by

Description	Symbol	Data Type
Standard input image with index l	J_l	Function $\mathbb{R}^2 \rightarrow \mathbb{R}^3$ (RGB color)
Stitched panoramic input image with index k , in equirectangular projection	I_k	Function $\mathbb{R}^2 \rightarrow \mathbb{R}^3$ (RGB color)
Orientation of standard input image J_l in the local coordinate system (i.e., relative to its stitched panoramic image I_k)	$q_{J_l}^*$	Quaternion
Orientation of standard input image J_l in the global coordinate system	q_{J_l}	Quaternion
Orientation of stitched equirectangular input image in the global coordinate system	q_{I_k}	Quaternion

Table 4.1.: Formula Symbols used in this Section

the quaternion $q_{J_a}^*$, and refers to the local coordinate system. This information is generated by the stitching process. Let q_{J_a} represent the orientation of the same image in the global coordinate system, obtained from the registration. The first quantity allows us to put the image into relation to its panorama. The second quantity allows us to put the image into the correct location in world space. Combining these two pieces of information, we can determine the orientation of the stitched panorama in world space, q_{I_k} .

$$q_{I_k} = (q_{J_a}^*)^{-1} \cdot q_{J_a} \quad (4.1)$$

The quaternion q is calculated for each panoramic sphere and the stitched panorama images are rotated accordingly. This is done without quality loss by re-running the stitching process ². After this process, all panoramas will have the same orientation in world space – provided the information from stitching and registration is correct.

²In my implementation, with the Hugin software, this is done as follows: My software updates the project file with the new orientation values. Then, it calls the Hugin software to produce the aligned output image.

4. Registration and Alignment

Achieving Error Tolerance

One problem with this approach is that it relies on a single anchor image, which is selected arbitrarily. If this anchor image is not correctly oriented by either registration or stitching, the panorama orientation q will be incorrect.

To solve this, we select, per stitched panorama, the most suitable image to be used as anchor image. To that end, we define an error measure err_a , shown in Formula 4.2. This value indicates, for a given selection of anchor image a , the agreement between the orientation values obtained from stitching and registration ($q_{J_l}^*$ and q_{J_l}), for all standard input images J_l belonging to this panorama.

$$err_a = \sum_{\text{Standard Input Images } l} \text{diff}(q_{J_l}^* \cdot (q_{J_a}^*)^{-1} \cdot q_{J_a}, q_{J_l})^2 \quad (4.2)$$

where $\text{diff}(q_1, q_2)$ measures the difference in rotation between two quaternions.

The function diff measures the difference in rotation between two quaternions, and can be implemented in a variety of ways. In practice, the following method worked well enough: Both quaternions are converted into their Yaw-Pitch-Roll representation, interpreted as a 3D vector $(x, y, z) = (\text{yaw}, \text{pitch}, \text{roll})$, and the Euclidean distance between them is calculated.

After evaluating err_a for all possible selections of anchor image a , the one with the smallest error score is selected.

This algorithm optimizes for agreement in orientation between registration and stitching. We assume that for this image, the position will also have been estimated correctly. Therefore, we simply use the position of the anchor image as position for the entire panorama and ignore the position values of all other constituent images.

4.3.2. Alignment with Panoramic Images

When working with panoramic images, OpenSFM gives us their orientation in world space, q_{I_k} , directly. All images can simply be rotated to match the orientation of the first image. This results in a slight quality loss, as one additional resampling step is necessary.

5. Interpolation

This chapter describes the generation of new views from the panoramic image material.

To generate a new view at a specific position c_v in the global coordinate system, the 2, 3 or 4 original viewpoints are selected which are closest to this point. c_v must be within the simplex spanned by these cameras. The following descriptions assume these viewpoints have already been selected.

Given are $2 \leq n \leq 4$ **aligned panoramic input images** I_1, I_2, \dots , taken at positions c_1, c_2, \dots in 3D space. For these explanations, we assume panoramas in equirectangular projection, but the same ideas can be used with any other projection type (e.g. cube maps or cylindrical projection). The goal is to render an **output image** at virtual camera position c_v , which can be any position in the space spanned by the original camera positions. This position can be represented by barycentric coordinates $a = (a_1, \dots, a_n)$ as follows:

$$c_v = \sum_{k=1}^n c_k \cdot a_k \text{ with } 0 \leq a_k \leq 1 \text{ and } \sum_{k=1}^n a_k = 1 \quad (5.1)$$

Using 2, 3 or 4 cameras provides 1D, 2D or 3D movement. This allows navigation along a line, on a triangle, or inside a tetrahedron, respectively.

In this thesis, only interpolation was considered (navigating inside the line, triangle or tetrahedron spanned by the cameras), and not extrapolation (navigating outside that volume).

5.1. Terminology and Notation

With “**Virtual Camera Position**” and “**Virtual Viewpoint**”, we mean a camera position that was not photographed, at which we want to render a new view. “**Original Camera Position**” or “**Original Viewpoint**” designates a camera

5. Interpolation

Table 5.1.: Notation used in this Thesis

Description	Symbol	Data Type
Pixel position	\mathbf{p}	\mathbb{R}^2
Equirectangular panoramic input images in RGB color	$I_1(\mathbf{p}), I_2(\mathbf{p}), \dots, I_n(\mathbf{p})$	Function $\mathbb{R}^2 \rightarrow \mathbb{R}^3$
Blendmask (floating-point weights per pixel and per image)	$B_1(\mathbf{p}), B_2(\mathbf{p}), \dots$	Function $\mathbb{R}^2 \rightarrow \mathbb{R}$
Optical Flow Field (a vector per pixel describing the displacement between two images)	$F(\mathbf{p})$	Function $\mathbb{R}^2 \rightarrow \mathbb{R}^2$
Original viewpoints for input images 1, 2, 3	$\mathbf{c}_1, \mathbf{c}_2, \dots$	\mathbb{R}^3
Virtual viewpoint for output image:	\mathbf{c}_v	\mathbb{R}^3
Barycentric Coordinates	$\mathbf{a} = (a_1, a_2, \dots, a_n)$	\mathbb{R}^n
Equirectangular panoramic intermediate image, generated from input image I_1 , at virtual camera position \mathbf{c}_v	$O_{I_1 \rightarrow \mathbf{c}_v}$	Function $\mathbb{R}^2 \rightarrow \mathbb{R}^3$
Equirectangular output image, generated from all input images, at virtual camera position \mathbf{c}_v	$O_{\star \rightarrow \mathbf{c}_v}$	Function $\mathbb{R}^2 \rightarrow \mathbb{R}^3$
Optical Flow calculated between two images I_1 and I_2	$F_{I_1 \rightarrow I_2}$	Function $\mathbb{R}^2 \rightarrow \mathbb{R}^2$
Effective Optical Flow generated for image I_1 with desired target warp position \mathbf{c}_v expressed in barycentric coordinates \mathbf{a}	$F_{I_1 \rightarrow \mathbf{a}}$	Function $\mathbb{R}^2 \rightarrow \mathbb{R}^2$
3D scene point	\mathbf{x}	\mathbb{R}^3
Normalized Spherical Coordinates	(θ_1, ϕ_1)	\mathbb{R}^2
Direction vector with unit length	\mathbf{v}_1	\mathbb{R}^3

Note: Normalized spherical coordinates are represented in the local coordinate system, and are used in Section 5.4.8. We use them in this section to represent the projection of a scene point onto the panoramic sphere.

position of an input image, in the global coordinate system. A “warp” is a deformation of a single image. With “interpolation” or “morphing”, we refer to the entire image rendering process.

Occlusions and Disocclusions

In this thesis, occlusions and disocclusions are defined in relation to one of the input images, and a specific output image. An occluded region (“occlusion”) is an area of the scene that is visible in the input image, but not in the output. A disoccluded region (“disocclusion”) is visible in the output image, but not in the particular input image. This is illustrated in Figure 5.1.

5.2. Related Work

This section describes some previous results in the field. This area of research is usually called “Viewpoint Morphing”, and is considered a subset of image-based rendering (IBR). The techniques mentioned here are optimized for a static scene, medium-to-large displacements, and large regions of occlusion. An in-depth look at image-based rendering can be found in the comprehensive book by Shum et al. [54].

McMillan and Bishop outline a theoretical model for viewpoint morphing [34]. Their model defines the “plenoptic function” as the light information available at a specific viewpoint in the scene, where the viewpoint is one of several parameters to that function. In that model, image-based rendering is defined as the reconstruction of a continuous plenoptic function from a discrete set of samples.

Seitz and Dyer [53] investigated the problem of generating new viewpoints from two source images of the same object. They base their method on classic image morphing methods. They show that classical morphing, which moves each pixel linearly along its displacement, fails to produce correct perspectives for in-between steps. To solve this, they add an additional warping step before and after the morphing, and prove that this leads to correct perspective. This approach is not applicable to this thesis, as panoramic images are used.

Pollard et al. [42] start with the same principle, but generalize the algorithm to three source images. They also introduce a lookup table for efficient retrieval

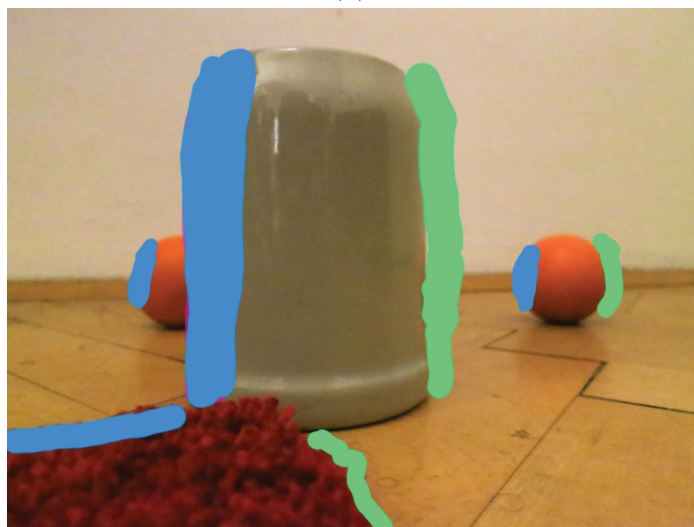
5. Interpolation



(a)



(b)



(c)

Figure 5.1.: Consider two pictures taken of the same static scene, (a) and (b). Figure (c) shows disocclusions and occlusions in (a), with respect to (b). Occlusions are scene regions in (a) that are not visible in (b), and are marked with green. Disocclusions are scene regions in (b) that are not visible in (a), and are marked with blue.

of matching edges. Lipski et al. [32] interpolate using four cameras, enabling navigation in 3D space.

Kolhatkar [28] performs optical-flow-based morphing on panoramic images, represented by a cubemap. Optical flow is explained in Section 5.3. Their main contribution is an efficient GPU implementation. My method is based on a similar scheme, but includes several improvements for occlusion handling and correct perspective.

The interpolation algorithm of Jelinek and Taylor [26] requires a series of standard images, taken from viewpoints that are relatively close together, with identical view direction. The images are placed on top of each other, to form a 3-dimensional volume. Each slice of that volume we call an “epipolar plane image”, an idea adopted from Bolles et al. [2]. Using image processing techniques, straight lines can be extracted from these epipolar plane images, and 3D scene points can be determined by analyzing the position and slope of these lines. This analysis is robust against occlusions. Finally, for rendering, the reconstructed 3D scene points are projected into each of the input images and connected with edges to form a triangulation. The resulting triangular 2D mesh can be used to warp the input images.

Lipski et al. [31] present a novel approach that uses both pixel correspondences between images and a geometric model, consisting of per-pixel depth information. That way, they are able to combine the strengths of both approaches. Their method works by estimating per-pixel correspondences, and then reconstructing approximate per-pixel depth from these correspondences. Using this depth information, the 3D scene points can be found and re-projected into any virtual camera position. The same idea was used in this thesis, as explained in Section 5.4.8.

Methods based on per-pixel depth may deliver incorrect results if the reconstructed depth information is inaccurate. Goesele et al. represent the depth uncertainty in their intermediate representation (an “ambient point cloud”), and perform some additional cleanup of the depth maps using image processing techniques [19]. Their hybrid rendering scheme uses ideas from both image-based and geometry-based rendering.

5.3. Optical Flow

My interpolation method is based on optical flow between pairs of images. Optical Flow refers to the problem of estimating per-pixel scene motion be-

5. Interpolation

tween two images. An example is shown in Figure 5.2. This section gives an introduction to optical flow in general, and explains how different algorithms can be compared to each other. Also, algorithm choice criteria for this thesis are outlined. Finally, some problems are discussed that occur when using standard optical flow algorithms on panoramic images.

5.3.1. Introduction

We learned that optical flow refers to the problem of estimating per-pixel scene motion between two images. Obviously, from the images it is not generally possible to reconstruct the true scene motion without ambiguities. Therefore, optical flow estimation is an underdetermined problem. However, it can be solved by making assumptions about brightness constancy of objects and smoothness of the resulting field. The problem was first discussed by Lucas & Kanade [33], who derived a solution based on local neighborhoods and least-squares optimization, and by Horn & Schunck [25], who presented a global optimization scheme. Newer algorithms use both colors and image gradients [13].

State-of-the-art approaches include DeepFlow [66] and EpicFlow [46]. Both methods use dense matching of image features. This provides high robustness to large displacements, which is important for this thesis. EpicFlow also includes edge detection, which helps to find precise motion boundaries.

To compare different algorithms, researchers have created several benchmarks. Sintel [7] is a 3D-rendered movie with large motions, specular reflections and other challenging features. The Middlebury dataset [49] contains stereo images of static scenes. The KITTI Vision Benchmark Suite [16] provides high-resolution stereo images taken from a moving vehicle. KITTI is not only used for optical flow benchmarking, but also for object tracking and geometry reconstruction. Of particular interest is the KITTI website [15], which lists scores for many optical flow algorithms and is continually updated.

5.3.2. Optical Flow Algorithm Selection

The interpolation method presented in this thesis can be used in conjunction with any optical flow algorithm. However, for optimal results, the optical flow needs to fulfill several criteria:

1. The ability to handle large displacements.
2. The ability to detect steep discontinuities with precision.

5.3. Optical Flow



(a)



(b)



(c)

Figure 5.2.: Images (a) and (b) are used as input for an optical flow algorithm, which calculates the flow shown in (c). Every arrow in (c) starts at a pixel position corresponding to (a) and ends at a pixel position corresponding to (b). In (c), the arrows are shown superimposed onto the image (a). In this visualization, only a small subset of optical flow vectors can be seen - in actuality, the optical flow is defined for every pixel of (a).

5. Interpolation

3. The algorithm should accept feature matches as input. This is necessary to handle large displacements. In the pipeline, 3D features identified during registration (see Chapter 4) can be used as features. These features are generated based on all images in the dataset. Therefore, this might outperform standard feature matching approaches, which only use two images.

Eventually, the DeepFlow algorithm [66] was selected. It meets all of the criteria, is reasonably fast and provides good performance on standard datasets [7, 16, 49]. The method uses deep convolutional neural networks and has two steps. First, using a novel matching algorithm (Deep Convolutional Matching, [45]), dense correspondences are computed between the images. Secondly, the optical flow field is determined using an energy minimization approach. In my pipeline, the matching algorithm is replaced by 3D features identified during registration (see Chapter 4). These 3D features also include visibility information, so 3D features are only projected into original viewpoints where they are actually visible.

Another approach worth mentioning is the work of Revaud et al., “EpicFlow” [46]. The algorithm consists of three parts: First, Deep Convolutional Matching is used [45]. Then, edge detection is performed. Finally, the optical flow is estimated using both the matching and the edges. This approach was not used because edge detection would be problematic on equirectangular images, where edges might wrap around the borders of the image. ¹

5.3.3. Wrapping Considerations

In our implementation, panorama images in equirectangular projection are used. Most optical flow algorithms are optimized for standard images and do not take the equirectangular geometry of the image into account. The most significant implication of this is that the optical flow algorithm cannot know that the equirectangular image wraps around the left/right borders.

There are some special optical flow algorithms developed for panoramic images, for example the approach of Mochizuki and Imiya [36]. However, only very few

¹This wrapping problem is not specific to edge detection, but also affects the optical flow estimation. A solution is presented in Section 5.3.3. However, that solution just makes sure that pixel-wise matches are wrapped around the edges of the image. Edges are more problematic, because they might span the entire image. Therefore, some adjustments would have to be made to the edge detection algorithm.

algorithms were designed for this purpose, so this would severely restrict the choice of optical flow algorithms available.

To solve that, the image is extended around the horizontal axis. The feature matches obtained during registration (see Chapter 4) are used to determine exactly how much extension is necessary. After the optical flow field has been estimated, the extra regions are simply thrown away. Extension around the vertical axis was not implemented in the pipeline, but could be done in a similar fashion.

5.4. Our Approach

The goal was to implement a visually convincing morphing technique that fulfills the following criteria:

- Runtime and memory requirement scaling linearly with the number of input and output pixels, irregardless of scene size.
- Can be combined with any optical flow software that generates the standard Middlebury .flo format [51]. Source code to read and write this format can be found at [50].
- Correct handling of occlusions and disocclusions.
- Arbitrary number of source images - the algorithm is exactly the same, regardless of whether we interpolate between 2, 3 or 4 images.

5.4.1. Overview

The basic approach is illustrated in Figure 5.4 and is explained in this section. The exact formulas and details for each step are described in the following sections. Highlighted terms refer to the corresponding node in the figure.

We start from 2, 3 or 4 equirectangular **panoramic input images** I_1, I_2, \dots . From these images, we want to generate a new image at virtual viewpoint c_v , expressed in barycentric coordinates $\mathbf{a} = (a_1, a_2, \dots, a_n)$. The virtual viewpoint must be inside the line, plane or tetrahedron spanned by the 2, 3 or 4 cameras respectively. As a pre-calculation step, we generate several optical flow fields $F_{I_x \rightarrow I_y}$ (one for each pair of images (I_x, I_y)).

We start the interpolation process by calculating the **effective flow** $F_{I_k \rightarrow \mathbf{a}}$ for each input image I_k . This “effective flow” field has the following meaning:

5. Interpolation

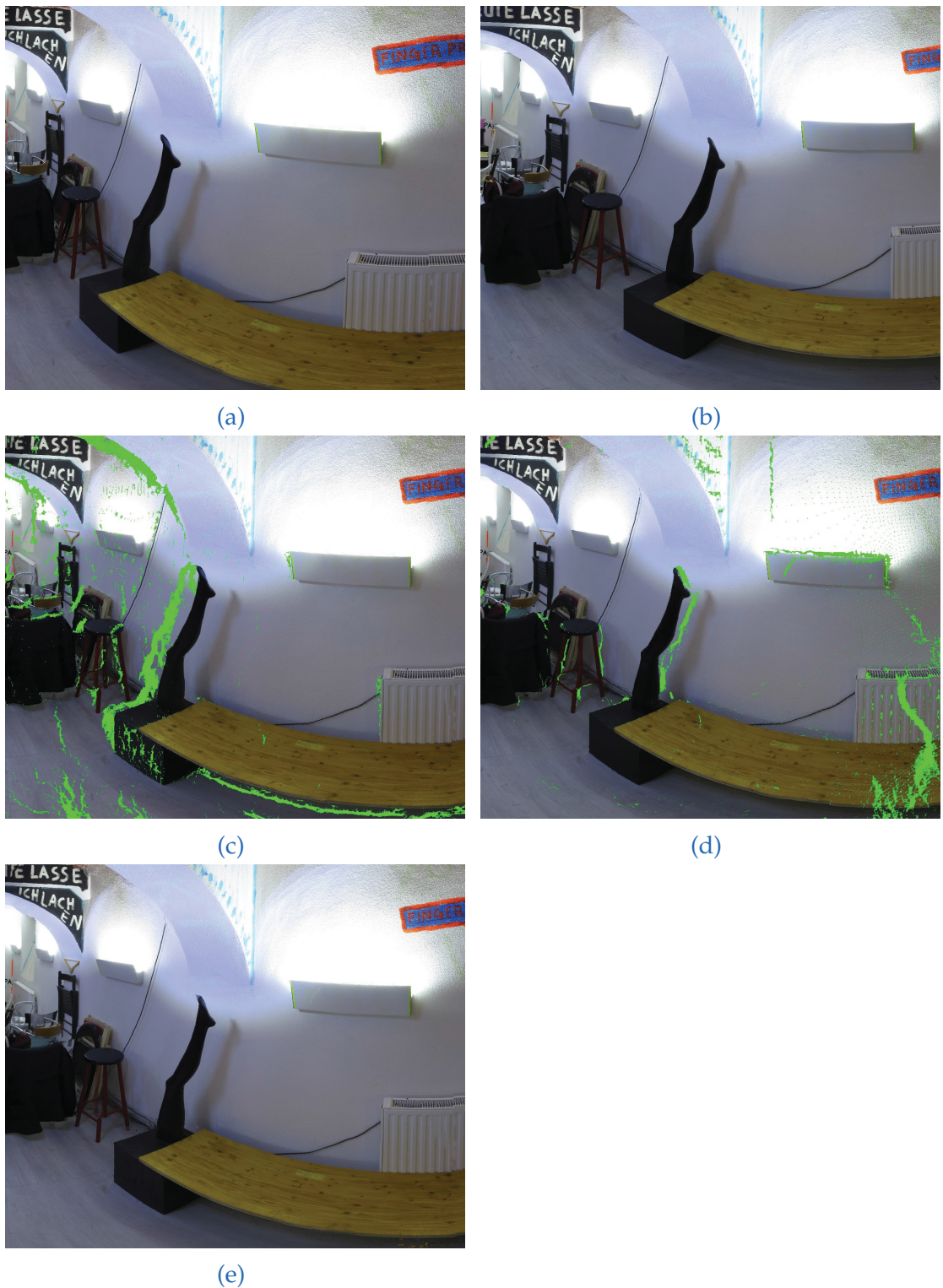


Figure 5.3.: This figure shows the equirectangular panoramic images involved in and created by the interpolation scheme. Only small cutouts of the full equirectangular images is shown, in order to better appreciate the details. We can see two input images (a) and (b), then two intermediate images generated from these, using the same virtual camera position ((c) and (d)). In the intermediate images, the estimated disocclusions are shown in green. Some of these disocclusions are estimated incorrectly, which will be explained during our Evaluation in Section 6.2.2. Image (e) shows the output image generated by blending the two intermediate images (c) and (d) together.

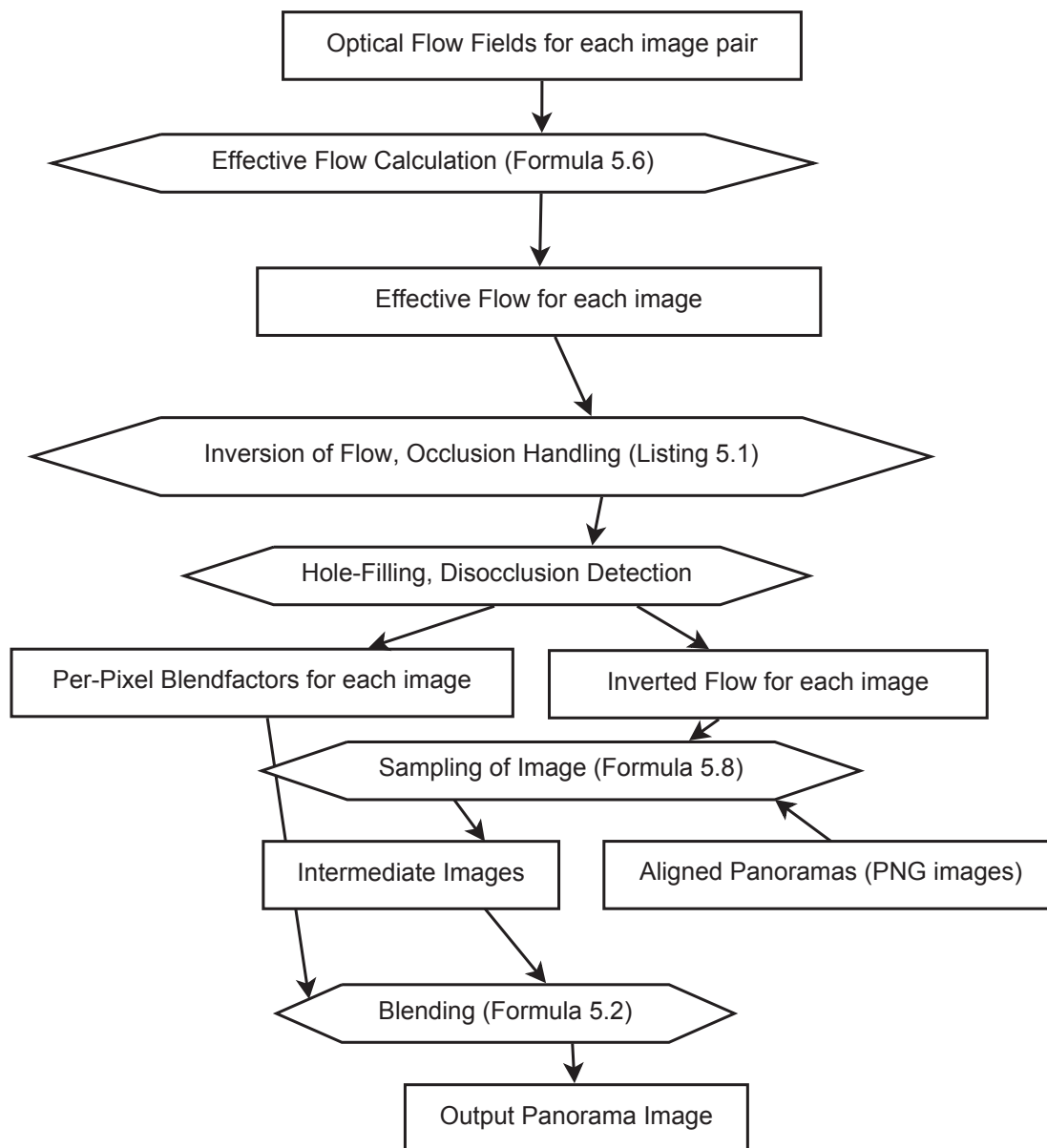


Figure 5.4.: The individual steps involved in the interpolation. In practice, most of these steps can be calculated in just one pass over the image. From optical flow fields, which have been pre-generated between pairs of images, the “effective flow fields” are calculated (one per original viewpoint). This flow field represents the per-pixel displacement which, when used to move the pixels of the corresponding image, will make it look as if taken from our target viewpoint. Instead of being used directly, this “effective flow” is inverted, and occlusions and disocclusions are handled, generating a per-pixel blendmask for each image. The input images are then sampled, generating intermediate images from each original image. These can be blended together, while taking into account the blendmask, to arrive at the final output image.

5. Interpolation

When each pixel of I_k is moved along the flow vectors, the resulting image looks as if taken at the target viewpoint c_v . This process of moving image pixels along motion vectors is called “forward morphing” and can be mathematically described as follows:

$$I_{k, \text{ forward morphed}}(\mathbf{p} + F_{I_k \rightarrow a}(\mathbf{p})) = I_k(\mathbf{p}) \text{ for each image pixel } \mathbf{p} \quad (5.2)$$

This forward morphing has several undesirable properties:

- Formula 5.2 uses all pixels of the input image I_k , but is not guaranteed to define all pixels in the output image $I_{k, \text{ forward morphed}}$. In general, in this thesis, we call pixels that are not written to by a certain formula or algorithm “undefined pixels”, and all other pixels “defined pixels”. Now, some pixels will be undefined after forward morphing, and we distinguish between two types of these pixels.
 - **Holes** arise when portions of the image are stretched, so that on some target pixels, no flow vector lands.
 - **Disocclusions** arise when objects move away, exposing background pixels that were not captured in this image. Generally, holes appear individually, whereas disocclusions appear as larger areas of undefined pixels.
- Several pixels \mathbf{p} in the target image may land on the same pixel in the destination image. We call these image pixels or regions “**occlusions**”.

These properties are illustrated in Figure 5.5. We solve these problems in separate ways:

- For the holes, we would like to fill them up with plausible color values, as those are purely artifacts of our forward morphing.
- Disocclusions, however, arise from the geometry of the scene and indicate which information was simply not captured in the respective input image. Therefore, we would like to remember these disoccluded pixels and then, in a later step, use the corresponding color values from the other image.
- To solve occlusions, we assume a static scene and a moving camera. Therefore the pixel with the largest motion vector will be closest to the camera and should overwrite pixels with smaller motion vectors. This is a common heuristic, used previously by [59], among others.

²Image Credit: textures created by Mitch Featherston, placed in public domain.

Input Flow:

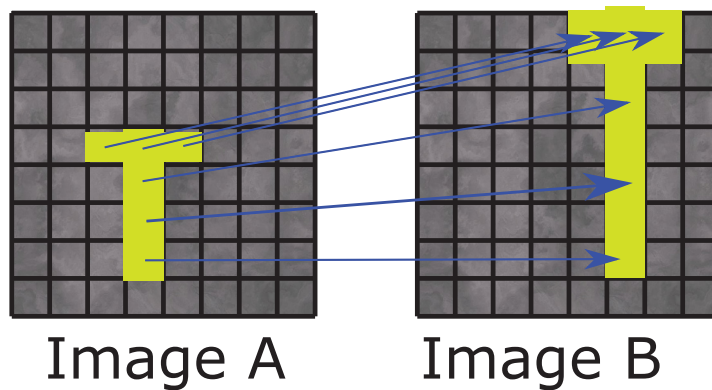
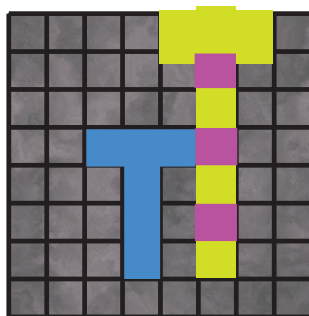


Image A, forward-morphed:



Holes

Disocclusions

Figure 5.5.: An example of holes and disocclusions. Consider two input images, showing two points in time. An object moves and stretches in front of a static background. The optical flow vectors are illustrated. When performing simple forward morphing of the first image along the flow vectors, holes and disocclusions will arise. We see that holes could be filled up with suitable color values. However, disocclusion pixels are simply not shown in the first input image - they will need to be taken from the other image. The situation presented here is a simple 2D case, but the same effects appear when forward-morphing panoramic images taken of static 3D scenes. ²

5. Interpolation

For occlusions, we simply store the length of the optical flow vectors and overwrite “slower” background objects with “faster” foreground objects. Disoccluded pixels should be left undefined. Both of these solutions can be easily applied to forward morphing.

However, hole-filling presents a problem in combination with forward morphing. For images of static scenes, the optical flow vectors will be uniform across large areas (because pixels belonging to the same object will move in the same direction). However, texture values will be much less uniform. Therefore, filling up holes in the color domain is less than ideal, and will lead to a reduction in image sharpness. A better solution would be to fill up the holes in the optical flow domain.

Therefore, we drop our previous idea of forward-morphing the color values. Instead, we perform all the filtering (occlusions, disocclusions, hole-filling) in the optical flow domain. To be able to do that, we perform **inversion of flow**. At the same time, we also do **occlusion handling** using the heuristic mentioned above (“longer motion vectors correspond to foreground objects, and should occlude background objects with shorter motion vectors”). Effectively, this means that at each position where an optical flow vector lands, the inverse vector is written, pointing back to the starting point.

We now have an inverse flow in which occlusions are already handled. This inverse flow will have some undefined pixels corresponding to disocclusions and holes, as shown in Figure 5.5. Therefore, the next step is to perform **disocclusion detection and hole-filling** (Section 5.4.5). This step distinguishes holes from disocclusions, fills up the holes and leaves disoccluded pixels undefined. The disoccluded pixels are stored separately³ using one **blendmask** B_k per image I_k .

Using the **inverted flow for each aligned input panorama** I_k , written $F_{I_k \rightarrow a}^{-1}$ and the **aligned input panoramas** I_k , we can now **sample the input images**, resulting in **intermediate images** $O_{I_1 \rightarrow c_v}$. These intermediate images will all look as if taken from the target viewpoint c_v .

This sampling is done using backwards morphing. While forwards morphing (Formula 5.2) iterates over the input pixels and moves them, Backwards morphing iterates over the pixels of the output image and samples the corresponding color value from the input image - which are easy to find by following the inverted flow.

³Note that at this point, the holes have already been filled up, so all undefined pixels correspond to disocclusions. Therefore, it would not actually be necessary to store them separately. However, this simplifies the explanations.

$$O_{I_1 \rightarrow c_v}(\mathbf{p}) = I_k(\mathbf{p} + F_{I_k \rightarrow a}^{-1}(\mathbf{p})) \text{ for each image pixel } \mathbf{p} \quad (5.3)$$

Examples of the resulting intermediate images can be seen in Figures 5.3c and 5.3d. The intermediate images are blended together, taking into account the blendmask and the distance between its original viewpoint and the target viewpoint, resulting in the **Output Panorama Image** $O_{\star \rightarrow c_v}$. An example of this can be seen in Figure 5.3e.

We will now look at each step in detail.

5.4.2. Optical Flow Field Generation

This is a pre-processing step, which only needs to be done once per pair of images. (Obviously, only pairs of images which will later be used for interpolation need to be considered). Optical Flow Generation was previously explained in Section 5.3.

5.4.3. Effective Flow Calculation

We define the effective flow field to be used for the panoramic input image I_k . This flow field could be used to forward-morph I_k to the desired target viewpoint c_v , as explained at the beginning of Section 5.4. The target viewpoint is expressed in barycentric coordinates $\mathbf{a} = (a_1, a_2, \dots, a_n)$, as explained at the very beginning of this Chapter.

$$F_{I_k \rightarrow a}(\mathbf{p}) = \mathbf{p} + \sum_{j=1}^n F_{I_k \rightarrow I_j}(\mathbf{p}) \cdot a_j \text{ where } \mathbf{p}: \text{ image coordinates of a pixel} \quad (5.4)$$

When using equirectangular projection, this linear warping does not produce physically correct intermediate images, even when the optical flow is perfect. This is because objects travelling along a straight line in 3D space will produce curved paths on the equirectangular projection. However, for most situations, the linear warping is acceptable. An improvement, taking into account the geometry of equirectangular panoramic images, is presented in Section 5.4.8.

5. Interpolation

5.4.4. Inversion of Flow and Occlusion Handling

We mentioned that it is desirable to invert the optical flow and perform some additional processing (disocclusion handling, hole-filling, occlusion handling) in the optical flow domain. Effectively, this inversion means that at each position where an optical flow vector lands, the inverse vector is written, pointing back to the starting pixel. This is illustrated in Figure 5.6.

Consider an input image I_k and the effective flow $F_{I_k \rightarrow a}$. First, $F_{I_k \rightarrow a}$ is inverted using an algorithm similar to Sánchez and Monzón [48]. What this inversion does is forward-morph the optical flow field, while simultaneously handling occlusions. The algorithm is shown in Listing 5.1.

It is worth noting that Sánchez et al. present multiple algorithms for optical flow inversions, but only one of these algorithms was used in this thesis. Therefore, improvement may be obtained by testing other approaches mentioned in their article [48].

Listing 5.1: Optical flow inversion (pseudocode).

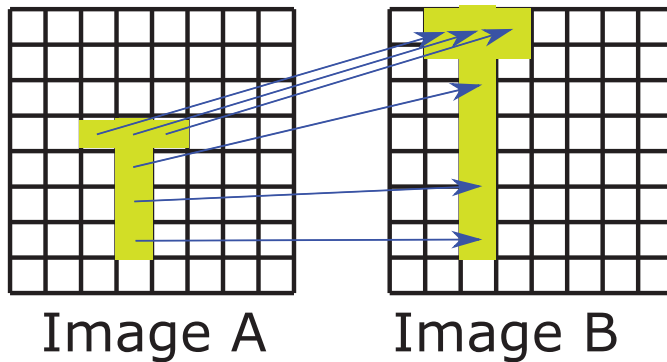
```
1 Input: flow field F,  
2   consisting of a floating-point vector (dx, dy) per pixel  
3 Output: inverted flow field FInverted  
4  
5 initialize FInverted with zero  
6 for each pixel p=(x,y)  
7   float vector targetposition = p + F(p)  
8   for each of the 4 pixels q closest to targetposition  
9     float w = EuclideanNorm(targetposition-q)  
10    // distance to the neighboring pixel  
11    if w>0.75  
12      continue  
13      // we only consider neighboring pixels  
14      // that we are reasonably close to  
15      float d = EuclideanNorm(FInverted(q)) // length of stored vector  
16      if EuclideanNorm(F(p)) > d // we found a faster-moving object  
17        // and assume it will occlude the slower-moving object  
18        FInverted(q) = targetposition - q // Point back towards pixel p
```

5.4.5. Disocclusion Detection and Hole-Filling

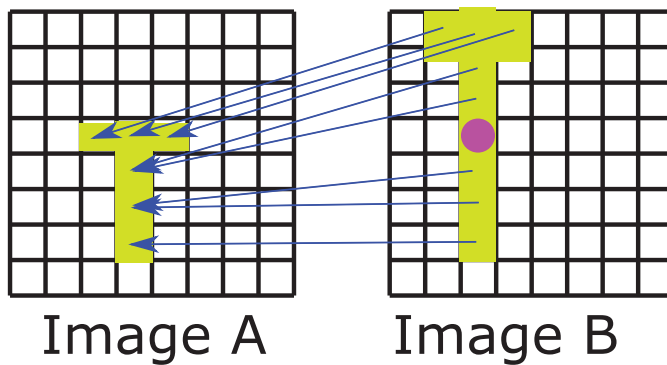
During the previous step, we performed inversion of the effective flow and handled occlusions. This leaves disocclusions and holes to be handled.

We have to discuss three problems:

(a) Input Flow:



(b) After Inversion:



(c) After Inversion and Hole-Filling:

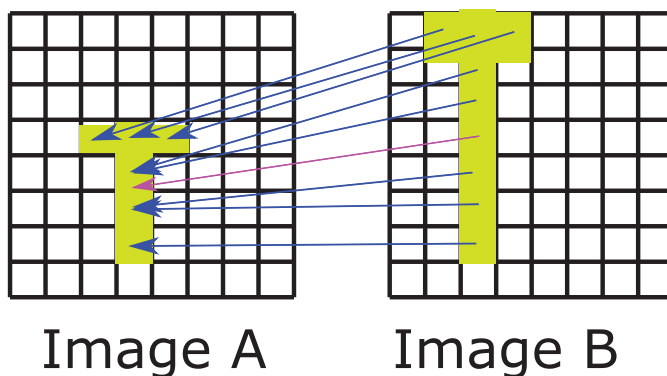


Figure 5.6.: Illustration of the hole-filling algorithm. (a) shows an optical flow field, matching pixels from the left image to locations in the right image. (b) shows the result of our optical flow inversion algorithm. The pixel with a pink circle remains undefined. (c) shows the result after the hole-filling procedure. The pixel with the pink circle has been correctly identified as a hole, because its two neighbors above and below have similar flow values. It is filled with the average of its neighbors.

5. Interpolation

- First, we need to distinguish holes and disocclusions. Both of them appear as undefined regions in the inverted flow.
- Then, we can fill up holes in the flow fields with values from the surrounding area.
- Finally, we generate a blendmask. This blendmask will be used during the final blending, in order to avoid sampling from disoccluded areas.

While holes and disocclusions both appear as undefined regions, they have different shapes: Holes appear as small regions, just one or two pixels wide. Disocclusions, on the other hand, appear as larger, contiguous areas. This is shown in Figure 5.5. A simple low-pass-filtering approach can be used to fill holes, and the algorithm is described in Listing 5.2.

We look at each undefined pixel, and consider the neighboring two pixels on opposite sides (left/right, or up/down) to make a decision. In order to be classified as a hole, two criteria have to be fulfilled. First, the two pixels on opposite sides must be defined. This is because holes mostly appear as single-pixel or two-pixel areas. Second, the flow vectors on two opposite sides must agree. This is to ensure that these two flow vectors correspond to the same object. If the two flow vectors on opposite sides do not agree, then the pixel we are looking at may be in between two objects, and actually be a very small disocclusion instead of a hole.

Listing 5.2: The hole-filling algorithm (pseudocode). The “norm” function stands for the Euclidean norm.

```
1 Input: inverted flow field F,  
2   which consists of a floating-point vector (d_x, d_y) per pixel  
3 Output: inverted flow field with holes detected and filled in.  
4  
5 initialize F_inv with zero  
6 for each pixel p = (x,y) in F  
7   if F(p) is defined  
8     continue  
9  
10  // Check left/right pixels  
11  if F(p - (1,0)) and F(p + (1,0)) are defined  
12    if norm(F(p - (1,0)) - F(p + (1,0))) < threshold // Vectors agree  
13      F(p) = average(F(p - (1,0)), F(p + (1,0)))  
14  
15  // Check up/down pixels  
16  if F(p - (0,1)) and F(p + (0,1)) are defined  
17    if norm(F(p - (0,1)) - F(p + (0,1))) < threshold  
18      F(p) = average(F(p - (0,1)), F(p + (0,1)))
```


Blendmask Generation

We generate 2, 3 or 4 blend masks for our input images I_k . The blendmask has two purposes: First, it should define linear blending of the input images depending on the distance between the new viewpoint c_v and the original viewpoints c_k . Secondly, it should mark disoccluded pixels in each image as a zero, so that these pixels taken from the other images during backwards morphing (Section 5.4.6).

$$B_k(\mathbf{p}) = \begin{cases} 0, & \text{if } I_{I_k \rightarrow c_v}(\mathbf{p}) \text{ disoccluded} \\ a_k, & \text{otherwise} \end{cases} \quad (5.5)$$

$$\text{and normalized s.t. } \sum_{k=1}^n B_k(\mathbf{p}) = 1 \quad \forall \text{ pixels } \mathbf{p}$$

where k : index of input and corresponding intermediate image

5.4.6. Sampling of Input Images to generate Intermediate Images

We now have an inverted flow field where occlusions are taken into account, and holes have been filled. We use this to backward-morph our input images, and generate one intermediate image from each input image:

$$O_{I_k \rightarrow c_v}(\mathbf{p}) = \begin{cases} \text{undefined,} & \text{if } F_{I_k \rightarrow a}^{-1}(\mathbf{p}) \text{ is undefined} \\ I_k(\mathbf{p} + F_{I_k \rightarrow a}^{-1}(\mathbf{p})), & \text{otherwise} \end{cases} \quad (5.6)$$

When sampling color values from the images, we generally do not sample at integer pixel location. This is because the optical flow vectors are non-integer. Obviously, simply rounding the pixel location to the nearest integer would discard a lot of information. Therefore, bilinear filtering is used for the color lookup. In bilinear filtering, for a given fractional pixel location, the four closest pixels are taken into account, and blended according to their distance. This improves results considerably. An introduction into bilinear filtering, as well as other filtering types, is presented by Getreuer [17].

5. Interpolation

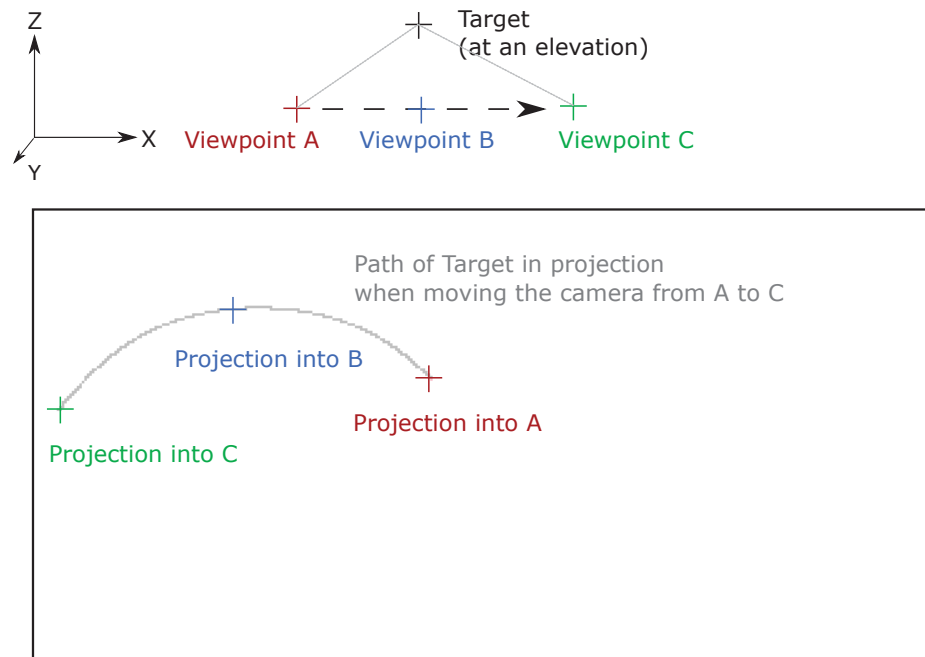


Figure 5.7.: Path of a static scene point (“Target”) when moving the camera. All camera positions are on one plane, whereas the target point is at an elevation. The image is shown in equirectangular projection ⁴.

5.4.7. Blending

To generate the output image, we perform linear **blending** of the intermediate images using the **per-pixel blendmask** $B_k(\mathbf{p})$ calculated previously.

$$O_{\star \rightarrow c_v}(\mathbf{p}) = \sum_{k=1}^n O_{I_k \rightarrow c_v} \cdot B_k(\mathbf{p}) \quad (5.7)$$

5.4.8. Improved Warping using Epipolar Constraints

The linear warping (formula 5.4) produces visually pleasant warps, especially if the displacements are small. However, in our equirectangular panoramic images, objects generally will not follow straight lines. To understand why, consider a static scene point at an elevation above some camera positions. The camera is moved along a straight line. The equirectangular projection of our scene point will follow a curve, as can be seen in Figure 5.7.

This curve represents a segment of the point’s epipolar line [23]. Theoretically, when moving between two cameras on a straight line, each image point should only move along its epipolar lines in the interpolated images. This follows from our assumption of a static scene and panoramas that are aligned relative to each other. Ideally, the optical flow algorithm should only find correspondences along the epipolar lines. However, most optical flow algorithms were developed for dynamic scenes and do not take these constraints into account. Therefore, they will not produce correspondences that match the epipolar lines exactly.

Existing Solutions

In the literature, a number of ways are discussed to address this problem. Seitz and Dyer’s classic paper [53] introduces two additional warping steps, leading to correct perspective. However, this approach is not applicable to panoramic images. McMillan and Bishop interpolate using panoramic images in cylindrical projection, and derive the epipolar geometry for that case [34]. They use a correspondence-finding algorithm that searches only along the epipolar lines. Kolhatkar [28] also investigated the panoramic case, and mentions two solutions⁵. First, we can re-project the optical flow onto the epipolar lines. However, according to Kolhatkar, this deteriorates the quality of the optical flow and leads to worse results overall. The second solution is to implement a custom optical flow algorithm, like McMillan and Bishop. However, a requirement for this thesis was to allow arbitrary optical flow software. Therefore, yet another approach was developed. The approach is similar to the interpolation technique described by Lipski et al. [31].

Our Approach

The method presented here is based on error-tolerant per-pixel triangulation of scene points. It accepts arbitrary flow fields as input, produces superior results to linear warping, and finds plausible solutions even if the flow vectors diverge from the epipolar lines.

We observe a scene point from different viewpoints c_1, c_2, \dots under different spherical angles $(\theta_1, \phi_1), (\theta_2, \phi_2), \dots$. The convention used for our spherical angles

⁴The path of the target point will be different in other projection types. For example, in cubemaps, the path will project to a line on one face of the cube, but may cross different faces.

⁵Note that Kolhatkar uses cubemaps, whereas for our implementation, equirectangular projection was used. However, this is not relevant here as the two solutions presented by Kolhatkar are agnostic of the type of input image used.

5. Interpolation

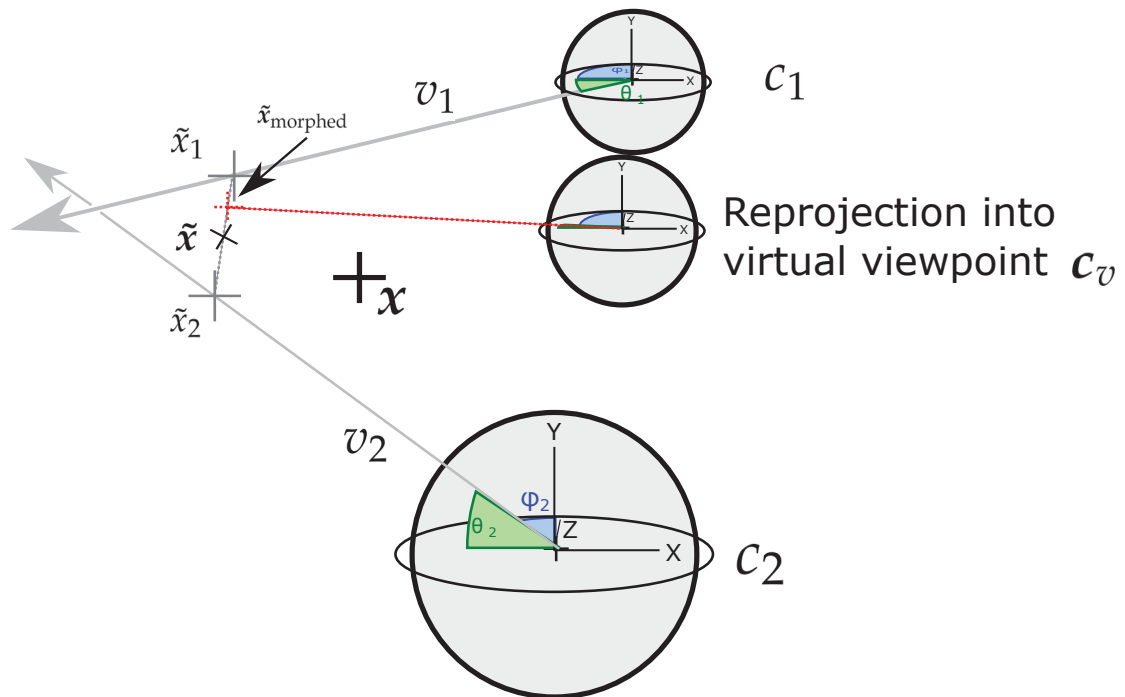


Figure 5.8: Improved Warping using Epipolar Constraints by error-tolerant per-pixel triangulation of scene points. We start with two view rays v_1, v_2 , who are both erroneous and do not intersect the true scene point x . We calculate the point closest to both lines, \tilde{x} . Then, we find the points on the view rays which are closest to \tilde{x} , and call them \tilde{x}_1, \tilde{x}_2 . Before projecting into our virtual viewpoint c_v , we generate $\tilde{x}_{\text{morphed}}$ by morphing between \tilde{x}_1, \tilde{x}_2 .

is introduced in Section 2.2.2. This is illustrated in Figure 5.8. These correspondences are trivially obtained by following the optical flow vectors. The scene point, x , is unknown. We also have the camera positions c_1, c_2, \dots for each of the observations. We are looking for the projection of x into a virtual camera at c_v .

We first assume that we have perfect optical flow. In this case, all view rays will intersect at the correct scene point x . We can find this point by triangulation. Once we have found it, we can trivially project it into any virtual camera position.

Now, we consider the situation when the optical flow is incorrect. In this case, the view rays will almost certainly not intersect. This is illustrated in Figure 5.8. We find an approximation \tilde{x} minimizing the squared distance to all of the view rays. This problem is discussed by Traa [65], and an efficient implementation was created by Eikenes [12]. We now have an approximate scene point, but it will not project back exactly into each of the observations. This is undesirable, because then the output image K_{c_v} would never equal the input image I_k , even when $c_v = r_k$. To avoid that, we calculate a different scene point \tilde{x}_k for each of the input images I_k :

$$\begin{aligned} \tilde{x}_k &= x_k \text{ projected onto ray with position } (c_k) \text{ and direction } \tilde{x} & (5.8) \\ &= c_k + v_k \cdot ((\tilde{x} - c_k) \cdot v_k) \end{aligned}$$

where v_k is (θ_1, ϕ_1) converted into a direction vector with unit length. This means, we find a point that projects exactly into our observation, while being as close as possible to the approximated scene point.

When moving the camera through the scene, we allow the scene point to move aswell. For re-projection at position c_v , represented by barycentric coordinates (a_1, a_2, \dots) (relative to Euclidean points c_1, c_2, \dots), we use $\tilde{x}_{\text{morphed}} = \sum_{k=1}^n \tilde{x}_k \cdot a_k$. Effectively, we calculate a weighted average and weigh each \tilde{x}_k stronger the closer we get to the corresponding viewpoint c_k .

To combine this approach with the interpolation method from Section 5.4.6, we only need to replace Formula 5.4. The rest of the algorithm remains unchanged.

5. Interpolation

5.4.9. Limitations

The method suffers from the following limitations:

- Reliance on a static scene.
- Reliance on optical flow - in cases where optical flow fails to provide useful information, the interpolation may not produce realistic results.
- No handling of reflections or transparency. This problem is addressed by Sinha et al. [55] and Kopf et al. [29].
- Fine-grained structures like trees and grass produce artifacts. With such structures, most pixels will lie on the edges of object. Therefore the optical flow algorithm produces imprecise results and separation of foreground and background (which is usually handled by our occlusion and disocclusion algorithms) is not cleanly possible.

6. Evaluation

In this chapter, the datasets acquired for this thesis will be presented. Results are presented to showcase specific strengths and weaknesses of the approach. The subjective quality of the output images will be evaluated, and some performance characteristics will be shown.

6.1. Datasets and Challenges

Table 6.1 shows the datasets created for this thesis, together with a list of specific challenges that each dataset presents:

- **Complex geometry** creates more disocclusions, which the system needs to handle. For an explanation of disocclusions, please refer to Figure 5.1.
- **Reflective materials:** The same object will have different color when photographed from different viewpoints. This influences Stitching, Optical Flow Generation and Registration.
- With **few distinctive features**, some images may be misaligned during stitching. Also, registration may be less precise because it relies on feature detection.
- **Fine geometry detail** presents a problem for optical flow generation, for two reasons. First, the correct optical flow would have sharp discontinuities. However, optical flow algorithms usually include a smoothing term preventing these discontinuities. Secondly, my approach relies on clear boundaries between objects. However, when geometry is very fine (as in trees), it will be blurred in the resulting image and therefore no sharp boundary is present.
- **Large displacements** presents two problems. First, it leads to large disocclusions, as mentioned previously. Secondly, many optical flow algorithms can not deal with large displacements. My approach can handle these displacements, but only if the Registration manages to find enough sparse 3D points covering the geometry (see Section 5.3.2).

6. Evaluation






	Bar	Lobby	Church	Living Room	Bridge
					
Acquisition Process	Standard Camera		Panoramic Camera		
Number of view-points photographed	3	3	27	50	16
Movement freedom	2D	2D	2D	3D	1D
Specific challenges	Complex geometry, reflective materials	Uniform textures with few distinctive features	Trees with fine geometry detail	Large displacements (geometry very close to camera)	Features directly above the camera

Table 6.1.: This table shows the datasets acquired for this thesis.

- **Features directly above the camera** can lead to large displacements in the panoramic images. This is because the equirectangular projection wraps around the poles.

6.2. Results

In this section, we will look at the results from several aspects.

First, we will investigate the influence of viewpoint distance on interpolation quality. Then, we will have a closer look at the disocclusion method, and show examples of success and failure. Next, the influence of the registration system (OpenSFM / VisualSFM) on interpolation quality will be investigated. Lastly, we will compare some of the generated images with an automated 3D reconstruction scheme obtained using third-party software.

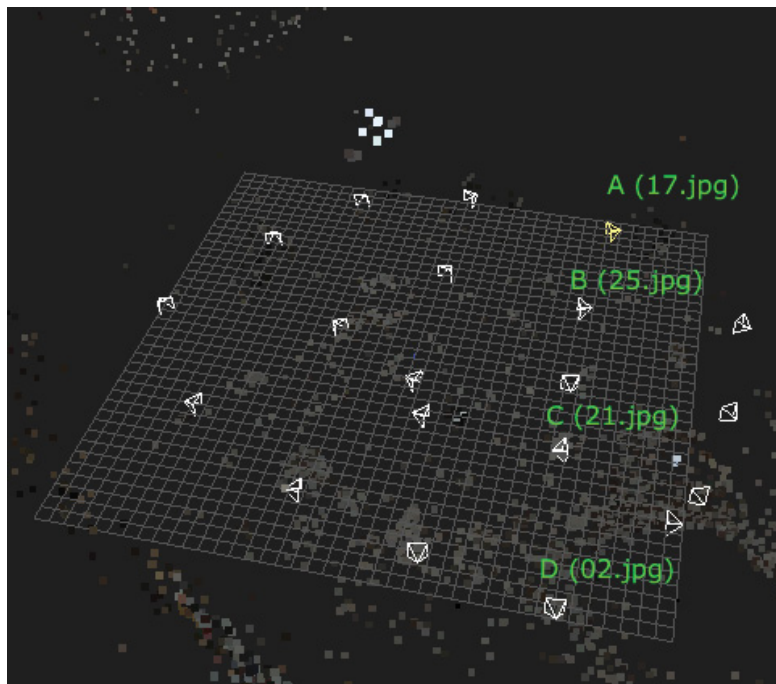


Figure 6.1.: Four original viewpoints selected for the test.

6.2.1. Influence of Distance between Original Viewpoints

In the dataset “Grabenkirche”, we have four original viewpoints A, B, C, D, that lie approximately on a line. The situation is shown in Figure 6.1. By interpolating between different pairs of images, we can evaluate the impact of distance on quality. The results are shown in Figure 6.2. We see that image quality worsens significantly with larger distance between viewpoints.

6.2.2. Disocclusion Handling

In Chapter 5, my method for handling disocclusions is presented. In this section, we look at some of the results and remaining problems. In each case, two original viewpoints are selected and a virtual viewpoint exactly halfway between them is generated. The images displayed are portions of equirectangular images.

Figure 6.3 shows successful disocclusion handling. Beneath the arch, we observe a good disocclusion mask, with some small errors. The result image (Figure 6.3e) looks very convincing.

6. Evaluation



(a)



(b)



(c)

Figure 6.2.: Influence of original viewpoint distance on output quality. (a), (b) and (c) show the output image with a distance between original viewpoints of 1, 2 and 3 units, respectively. The pairs of cameras used are A B, A C and A D from Figure 6.1, respectively. The virtual viewpoint is exactly halfway the respective original viewpoints.



(a)



(b)



(c)



(d)



(e)

Figure 6.3.: Correct disocclusion handling. (a) and (b) are the input images. (c) and (d) show the intermediate images, with disocclusions marked in green. (e) shows the blended output image.

6. Evaluation

Figure 6.6 shows an incorrect disocclusion mask. During the warping process, undefined pixels arise in the intermediate images (Figure 6.6c and 6.6d). These undefined pixels stem from two sources: due to the forward interpolation process (“holes”), and because of disocclusions. The algorithm tries to distinguish the two cases and fill the holes using a simple heuristic (see Chapter 5.4.5). In this case, the displacement was too large, and the algorithm was unable to fill all the holes correctly. The output image (Figure 6.6e) still looks very good. This is because the illumination is very similar in the input images (Figure 6.6a and 6.6b), so it doesn’t make much difference whether pixels are sampled from one image or both.

6.2.3. Influence of the Registration System

As mentioned in Section 4.2.2, panoramic images are registered using the OpenSFM package, whereas the VisualSFM package is used for standard images. In Section 5.3.2, we saw that the registration package delivers sparse 3D points plus visibility information. These are used as soft constraints for the optical flow algorithm. During testing, it became obvious that VisualSFM delivers a better 3D reconstruction than OpenSFM, with more points and better visibility information.

In this section, we will compare the quantity of matches delivered by both registration systems (VisualSFM and OpenSFM).

Quantitative Evaluation

For this test, the “Grabenkirche” dataset was considered. A small sample of 7 panoramic images is used. From each of these, 12 standard images looking into different directions are extracted, for a total of 84 test images. These standard images are sent to VisualSFM for registration. The 7 panoramic images are registered using OpenSFM directly.

The results are shown in Table 6.2. Subjectively, both systems deliver a convincing point cloud. However, we see that VisualSFM delivers more 3D points, which are each seen by more cameras. Therefore, registering with VisualSFM delivers more matches per image pair, which provides a more precise constraint for Optical Flow estimation. This justifies the selection of VisualSFM as registration for standard images, even though those could be registered by OpenSFM as well.

	OpenSFM	VisualSFM
Input	7 panoramic images	84 standard images (pinhole camera)
Runtime (approx)	2 minutes	10 minutes
Number of 3D points reconstructed	5595	9115
Number of 3D points seen from all viewpoints	126 (2.2%)	437 (4.8%)
Avg number of 3D points seen from a viewpoint	2439 (43.6%)	4341 (47.6%)
Avg number of viewpoints seeing a 3D point	3	5

Table 6.2.: Sparse 3D reconstruction comparison between VisualSFM and OpenSFM

Influence of Equirectangular Projection

In the previous section, we have seen that OpenSFM (which is fed with equirectangular images) delivers significantly less scene points than VisualSFM (which receives standard images).

The reconstruction algorithm in OpenSFM supports equirectangular images just fine, and the algorithm converges on a solution. However, feature points are only found near the equators of the images, as can be seen in Figure 6.4.

My hypothesis is that this has to do with the equirectangular projection used for the input images. These images feature significant distortion around the poles. This means, the same feature appears in different sizes in different images, and is stretched in different ways. Therefore, matching algorithm is unable to find the corresponding match in the other image. The problem does not occur at the equators, where size is more uniform and stretching is almost non-existent.

6.2.4. Comparison with Geometry-Based Approach

One of the key questions posed in Chapter 1 was how the image-based rendering approach presented here compares with geometry-based rendering. In this

6. Evaluation

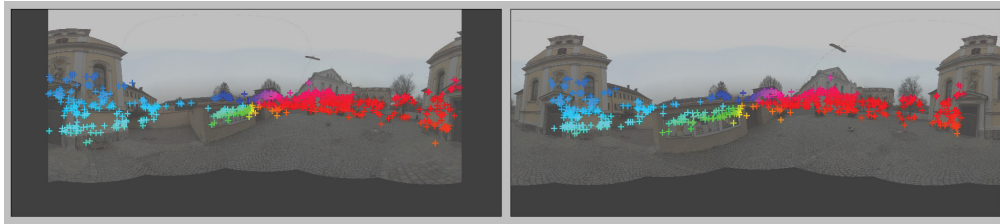


Figure 6.4.: OpenSFM matches are found almost exclusively around the equator of the image. This likely stems from the equirectangular projection, which produces distortion and makes the feature detection algorithm fail near the poles. Visualization produced using code from [45].

section, we do a brief comparison using one of our datasets.

Methodology

We use one of the datasets, “Bar”. The VisualSFM software [67] is used for registration inside the pipeline, as was explained in Chapter 4. As part of the preprocessing, VisualSFM finds camera positions and a small set of 3D scene points (a “sparse point cloud”). To compare the results with a geometry-based approach, we need to reconstruct detailed geometry from the images. This is done as follows:

Step 1: Dense Point Cloud Reconstruction. The CMVS/PMVS software is used to reconstruct a large set of 3D scene points (a “dense point cloud”). The software is available under the GPL license at [8], and a newer version is available at [14].

Step 2: Mesh Reconstruction. A mesh is reconstructed, consisting of triangles, from the dense point cloud. This is not strictly necessary, because there are methods to render point clouds directly. However, reconstructing surfaces allows using a wider range of rendering tools. Also, implicitly in the mesh reconstruction process, some automated cleanup is performed, reducing noise in the data. For this purpose, the open-source MeshLab software [35] is used. The CMVS/PMVS data can be imported into MeshLab. To reconstruct a triangle mesh, the Meshlab function “Filters → Point Set → Surface Reconstruction Poisson” is used.

Step 3: Texture Generation. Now, textures are created for the reconstructed mesh. This is done by projecting the original images onto the mesh. This functionality is also included in the MeshLab software, under the menu item

Filters → Texture → Parametrization + Texture from registered Rasters. In practice, MeshLab simply picks a subset of the original input images, packs them together into a single large texture, and generates texture coordinates to reference that large texture. The size of the large texture is adjustable. For example, for a size of 4096 by 4096 pixels, the resulting PNG image has 17 MB.

Step 4: Rendering. The mesh can now be rendered, using the generated textures, to produce new images. Usually, for high-quality rendering, it would be preferable to use special rendering software that calculates illumination in a realistic fashion. However, in this case all textures are reconstructed from photos. Therefore, everything is already lit, and sophisticated rendering software is not required. A simple rendering can be exported using MeshLab.

Results

Figure 6.5 shows the results from geometry-based reconstruction, when compared with the pipeline presented in this thesis.

We observe the following artifacts in the image-based result. The numbers refer to Figure 6.5. **(1)** Noise due to an erroneous disocclusion mask, in combination with varying illumination. This was explained in Section 6.2.2. **(2)** Blur due to imprecise matches between the three input images. **(3)** Disocclusion mask with slight errors at the border.

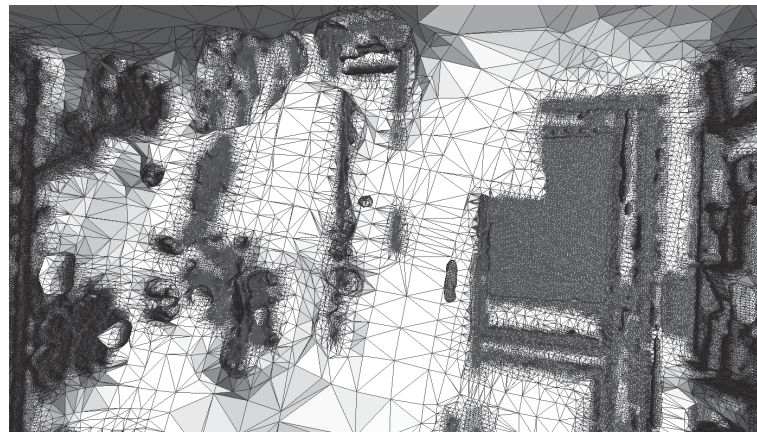
In the geometry-based result, we observe the following artifacts: **(4)** Incorrect geometry at occluded regions. In this scene, these regions are problematic to reconstruct because they are seen by only one or two cameras. **(5)** Bubbles erroneously reconstructed in the middle of the room. These bubbles could be removed by some additional processing. **(6)** Fine geometric details present a problem for the reconstruction. **(7)** Incorrect texture projection, and noticeable seams due to changes in illumination.

Table 6.3 shows the performance characteristics of both approaches (runtime, disk space). We see that both approaches perform similarly in this case. However, this comparison is just a rough example, as we are only comparing two specific implementations with each other. Both the pointcloud and the optical flow fields could be compressed, leading to reductions in storage requirements.

We arrive at the following conclusions:

- Subjectively, both results look acceptable.
- The image-based approach produced less artifacts and looks better overall. The geometry-based approach produces sharper results in some regions.

6. Evaluation



(a)



(b)



(c)

Figure 6.5.: Comparing the pipeline with geometry-based rendering. (a) shows the reconstructed mesh. (b) shows the same mesh with a texture applied. (c) shows the result from the pipeline. The scene was photographed from three viewpoints. Visual artifacts are numbered and referenced in the text.

6.3. Performance Characteristics

	CMVS & Meshlab	My Approach
Preprocessing Time	40 minutes	55 minutes
Rendering Time per frame	100ms (GPU rendering of geometry)	10 seconds (CPU-based)
Storage Requirements on Disk	200 MB (a single PNG image & geometry data)	740 MB (3 PNG images & 6 uncompressed optical flow fields)

Table 6.3.: Comparison of geometry-based rendering (using CMVS & MeshLab) and image-based rendering (using my pipeline). The output image size is 5000x2500px. The dataset “bar” was used for this test, and the measurements are based on three viewpoints.

- The mesh contains several obvious errors. This is not surprising, as only three viewpoints are used for the reconstruction. These mesh errors could be resolved using automatic cleanup techniques, or by simply shooting more viewpoints.
- If the viewpoints were photographed under different lighting conditions, both methods will produce artifacts. As a workaround, the exposure of the images could be adjusted as a pre-processing step.
- The performance characteristics of both approaches are similar on this dataset. When scaling to a large number of viewpoints, the image-based method would have one advantage: Only 3 or 4 viewpoints need to be considered at one time, therefore rendering time is independent of scene size. With geometry-based rendering, it would be much harder to guarantee stable rendering times. Sophisticated occlusion detection and level-of-detail schemes would need to be implemented to render large scenes at predictable framerates.

6.3. Performance Characteristics

Table 6.4 shows runtime and memory usage for the “bar” dataset (see Table 6.1). This dataset consists of 3 viewpoints consisting of 20 4000px by 3000px source images each. They are stitched to a 5000px by 2500px panoramic image. Registration of the 3 · 20 source images is performed by VisualSFM. All software is running on a PC with an Intel i7-4600U CPU, 8GB of RAM and an NVidia GT 730M. The graphics card is relevant for VisualSFM only.

6. Evaluation

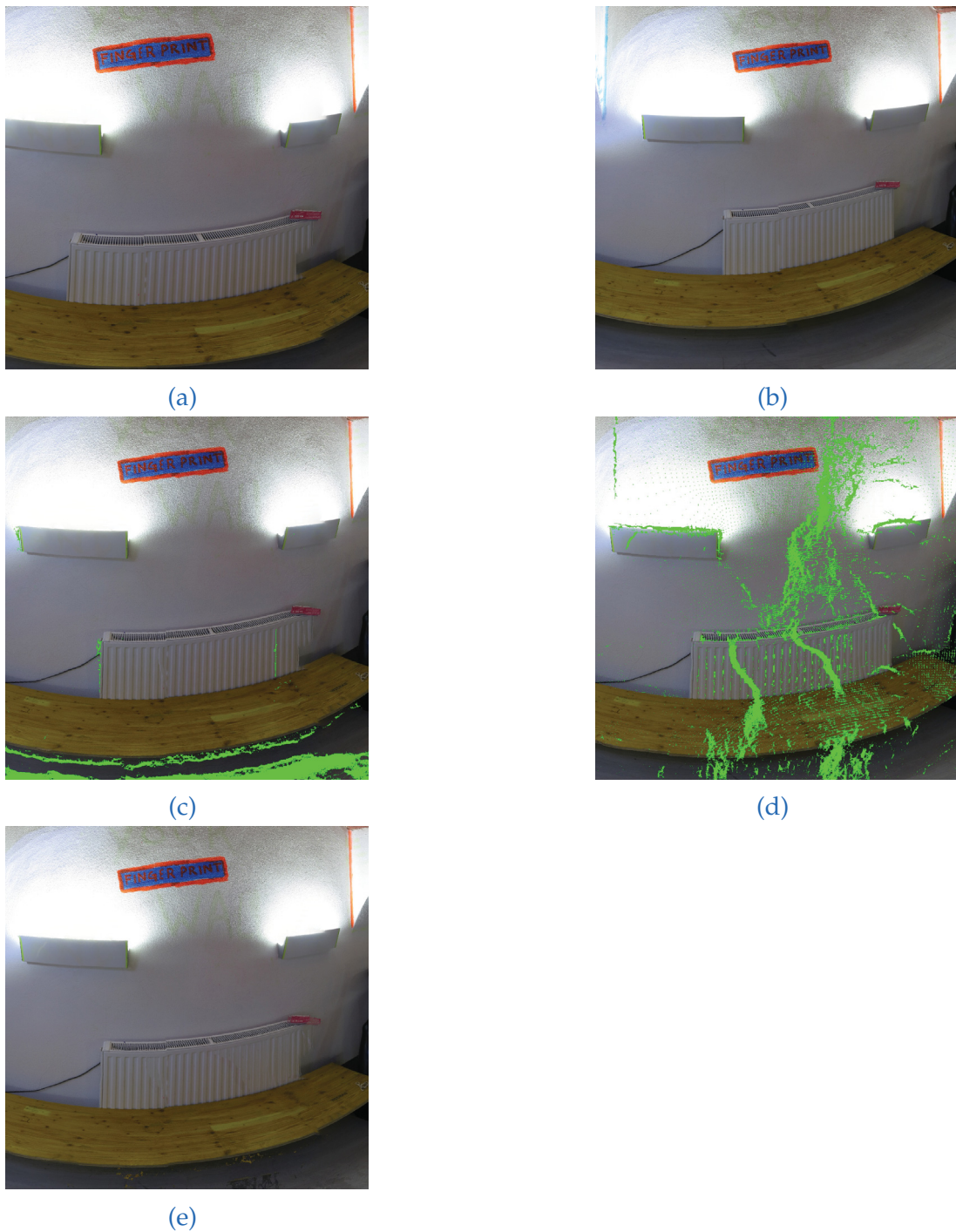


Figure 6.6.: Incorrect disocclusion mask on stretched regions of the image. (a) and (b) are the input images. (c) and (d) show the intermediate images, with disocclusions marked in green. The problem can be seen on (c), where the image is stretched. We would expect no disocclusions to be detected on the wall, because the hole-filling algorithm would have detected any undefined pixels as “holes”. Instead, what we see is that the hole-filling algorithm does not perform well under large stretching, and that the pixels are incorrectly identified as disocclusions. (d), on the other hand, is not stretched, therefore the problem does not appear. (e) shows the blended output image. Note that the seam in image 6.6b is due to an error in stitching and does not influence the problem discussed here.

Task	Runtime	Memory requirements (max.)	Re-	Notes
Registration	8 minutes	150 MB		Unlike the rest of the pipeline, VisualSFM is GPU-accelerated. Therefore, some amount of GPU memory is required aswell.
Stitching and Alignment	4 minutes	700 MB		
Optical Flow Generation	10 minutes per pair of images	5000 MB		Only flow fields between pairs of adjacent viewpoints need to be generated.
Interpolation (3 viewpoints)	Interpolation along straight line: 15 seconds. Interpolation along equirectangular path (Section 5.4.8): 2 minutes	1150 MB		

Table 6.4.: Runtime and memory usage for the “bar” dataset

7. Conclusion

The goal of this thesis was to implement a pipeline for viewpoint interpolation of panoramic images. The pipeline was presented step-by-step, and the results were evaluated.

We saw that realistic and visually pleasing results were obtained for a variety of scenes, even in the presence of complex geometry and occlusions.

7.1. Advantages, Disadvantages and Applications

In Chapter 6, the pipeline was evaluated under different criteria. Based on this knowledge, recommendations about possible applications can be made.

We saw in Chapter 3 that data acquisition takes between 10 seconds and 3 minutes per panorama, depending on the camera used. Chapter 4 explains how panoramic images are registered in 3D space and aligned relative to each other, without any user input. Therefore, the data acquisition process is very fast and can be done by untrained users, using only standard cameras.

Because of the fully automatic processing and easy acquisition process, my approach would be ideally suited for a **smartphone app**. The user would take standard images and upload them to an online service, where they are stitched and registered. Optical flow fields and panoramic input images could then be downloaded onto the smartphone, or displayed on web pages using an interactive viewer. This would present challenges in the area of stitching (because the user may not be able to pivot the smartphone camera around its optical center) and data compression (because mobile bandwidth is limited).

A main obstacle to developing such an interactive viewer would be the size of the intermediate representation, consisting of panoramic images and flow fields. The images are reasonably small, because of JPEG compression. However, optical flow fields are currently uncompressed and require 64 bit per pixel. In order to reduce bandwidth usage, a good optical flow compression algorithm

7. Conclusion

would need to be implemented. A large compression factor could potentially be achieved, since those flow fields are quite smooth.

In Chapter 1, **interactive simulations** were mentioned as a possible application. These simulations require free navigation through large outdoor scenes with high visual fidelity. Furthermore, acquisition should be as easy as possible. I conclude that the pipeline is ideally suited for this purpose, for the following reasons: First, image acquisition turned out to be very fast by using a panoramic camera. Secondly, my interpolation scheme performs well in outdoor scenes. Thirdly, since only 2-4 cameras are used for interpolation, very large scenes can be rendered with predictable performance. However, the problem of loading flow fields in and out of memory dynamically would need to be addressed. And lastly, while the intermediate representation is quite large, this is usually not a problem in offline simulation applications.

Virtual reality applications would also benefit from my technology, for the following reasons. First, users expect a high degree of visual fidelity from such applications. The image-based approach can provide that. Secondly, virtual reality applications require high and predictable framerates for good results. My image-based approach fulfills this criteria aswell - the rendering performance is independent of scene complexity. Thirdly, by using the registration information, it would be relatively easy to insert virtual objects into the real scenes.

7.2. Future Work

For each step in the pipeline, there are specific opportunities for improvement. Improving any of the steps will have a direct impact on the quality of the final output.

The **Registration** detects sparse 3D features. These features are used to initialize the optical flow. By finding more features, more accurate results would be obtained during interpolation. To find more accurate features, a different registration framework could be used. Alternatively, using the results of the registration and under the assumption of a static scene, image-based matching could be performed along epipolar lines.

Optical Flow Generation is an area of active research and steady improvements are made [15]. As the state-of-the-art advances, the pipeline will benefit from better optical flow algorithms.

Optical Flow is calculated between pairs of images. It would be interesting to devise an optical flow approach that uses information from multiple images at the same times.

The **Interpolation** algorithm could be implemented on the GPU, using programmable shading and multiple passes per image.

Reflections and transparency can lead to artifacts in some situations. This problem is addressed by Kopf et al. [29], and their approach could be integrated into the pipeline.

The hole filling algorithm (part of **Occlusion Handling**) could be improved. By incorporating ideas from image processing, holes and disocclusions could be distinguished more reliably (Section 5.4.5).

It would be interesting to combine the image-based rendering approach presented here with geometry-based rendering, and to place virtual objects into photographed scenes.

7.3. Benefits and Contribution

The benefits of this pipeline are low-cost acquisition, automatic registration of input images, and a fully automatic pipeline requiring no user input. This enables users without training to acquire data rapidly, without needing to take any measurements during shooting.

The main scientific contribution is the interpolation approach (Chapter 5), which combines several existing ideas, and the scheme for taking into account epipolar geometry of panoramic images (Section 5.4.8).

In conclusion, the image-based rendering pipeline delivered promising results. The interpolation scheme is relatively simple, but produces high-quality results. The approach scales well because interpolation runtime is independent of the size of the scene. Furthermore, the components are exchangeable. Therefore, as the state-of-the-art improves in structure-from-motion and optical flow algorithms, new results can be integrated into the pipeline quickly, and output quality can be improved further.

Appendices

A. File Formats, Directory Structure, Interfaces

In Figure 2.1 we saw an overview over the pipeline, with all the individual steps and intermediate results. In this Chapter, the data structure, as used in the implementation, is specified in more detail.

A.1. Dataset Directory and Input Images

Each dataset must be in a separate directory. In these explanations, “dataset/” refers to the root of that directory.

Input images are expected to be in JPG format, and have identical sizes. Panoramic input images in equirectangular projection must be placed in a directory “dataset/equirectangular”. Standard input images must be sorted by viewpoint into subfolders “p1, p2, p3” (or o1, o2, o3, or 001, 002, 003, ...). At the moment, it is not possible to mix panoramic and standard input images in the same dataset.

Stitched images are stored as “dataset/out/input_dir_name.png”, where “input_dir_name” refers to the directories “p1”, “p2”, ...

When using OpenSFM to register panoramic input images, the package must be run manually via the command line. The result files must be placed in the “dataset/opensfm” directory. Only the files “reconstruction.json” and “tracks.csv” are required. For VisualSFM, this is not necessary as VisualSFM is run automatically by the pipeline.

Registration data and references to the input files are stored in a JSON file “dataset/data.json”. The exact format is shown in Listing A.1.

Optical Flow Files are stored in “dataset/out/flow”.

Output Images are saved under “dataset/out/morph”.

A. File Formats, Directory Structure, Interfaces

Listing A.1: The JSON format used for intermediate data. Comments are indicated by “//” - this is not valid JSON, and the comments must not be present in the actual file.

```
1 {
2 "world_data": {
3   "panorama_in_world": {
4     // The orientation of a panorama in the world.
5     // As we are using aligned panoramas,
6     // this is valid for all panoramas.
7     "Yaw": -1.3182254494071435,
8     "Pitch": 0.19422825436870126,
9     "Roll": -0.04407010612122689
10    },
11    "world_in_panorama": { // The inverse of panorama_in_world.
12      "Yaw": 1.3230713120961948,
13      "Pitch": -0.005527006790184677,
14      "Roll": 0.19902803394009256
15    }
16  },
17  "p1": { // Name of viewpoint (referring to input directory)
18    "sourceFiles": [ // List of input files
19      // —> several for standard input images,
20      // —> one for panoramic input images
21      "./dataset/p1/IMG_0713.JPG",
22      "./dataset/p1/IMG_0714.JPG",
23      ...
24    ],
25    "filename": "./dataset/out/p1.png", // filename of the aligned
26    // panorama image
27    "orientationsFromAssembler": [ // orientations for each image
28      // determined during stitching (for standard input images only)
29      // or empty (for panoramic input images)
30      // this is used for the alignment.
31      {
32        "Yaw": -3.1414328495460435,
33        "Pitch": -0.18578072950462699,
34        "Roll": -1.5690956860311017
35      },
36      {
37        "Yaw": -2.401452035118999,
38        "Pitch": -0.18385896298492289,
39        "Roll": -1.5593538379044747
40      },
41      ...
42    ],
43    "rawOrientationsFromRegistration": [
44      // Raw orientations determined from the registration,
45      // as quaternions,
46      // in the original coordinate system used by the registration.
47      // Each item refers to one input image.
```

A.1. Dataset Directory and Input Images

```
48 // Therefore, for panoramic input images,
49 // this array has only one entry.
50 {
51     "x": -0.644167131431,
52     "y": -0.671162689575,
53     "z": -0.310235339756,
54     "w": 0.19582381553
55 },
56 {
57     "x": -0.493124572437,
58     "y": -0.556997806558,
59     "z": -0.514582631658,
60     "w": 0.426365138122
61 },
62 ...
63 ],
64 "orientationsFromRegistration": [
65     // Same as rawOrientationsFromRegistration,
66     // but converted into our coordinate system
67     // and expressed as Yaw/Pitch/Roll
68     {
69         "Yaw": -3.1414328495460435,
70         "Pitch": -0.18578072950462699,
71         "Roll": -1.5690956860311017
72     },
73     {
74         "Yaw": -2.4119936391460206,
75         "Pitch": -0.18267925831078486,
76         "Roll": -1.5562355457611219
77     },
78 "anchorImage": 0, // The 0-based index of the anchor image.
79 // (always 0 for OpenSFM)
80 "position": {
81     // The position of the panorama in global coordinates
82     "X": 0.016754085198,
83     "Y": 0.0995249673724,
84     "Z": 0.0884485244751
85 },
86 "matches_in": [
87     // List of matches retrieved from OpenSFM,
88     // in image coordinates.
89     // The IDs allow matching over several viewpoints.
90     {
91         "ID": 8,
92         "X": 0.238553,
93         "Y": 0.114778
94     },
95     {
96         "ID": 12,
```

A. File Formats, Directory Structure, Interfaces

```
197     "X": -0.24033,
198     "Y": 0.0259952
199   },
200   ...
201 ],
202 "matches_in_3d": [
203   // List of matches retrieved from VisualSFM, in world coordinates
204   .
205   {
206     "ID": 1,
207     "X": 0.882973492146,
208     "Y": -1.13085711002,
209     "Z": 1.27483558655
210   },
211   ...
212 ],
213 "source": 0, // Input image format:
214 // 0 for standard images, 1 for equirectangular panoramic images
215 "registration": 2, // The registration system used.
216 // 1 for OpenSFM, 2 for VisualSFM.
217 "positionsFromRegistration": [
218   // The positions, in global coordinates, retrieved from the
219   // registration.
220   {
221     "X": -0.694735527039,
222     "Y": -0.0020846221596,
223     "Z": -0.521440088749
224   },
225   {
226     "X": -0.700594961643,
227     "Y": -0.00269711762667,
228     "Z": -0.516122758389
229   },
230   ...
231 ]
232 },
233 "p2": { ... },
234 "p3": { ... }
235 }
```

All elements of the JSON must be present, however, not all values are used for all stages of the pipeline.

- The following elements are used for flow generation and interpolation:
 - world_data
 - position
 - matches_in

- `matches_in_3d`
- The following elements are used for all steps:
 - `source_files`
 - `filename`.
- The following elements are written to the JSON file during registration, stitching and alignment. They are not required for flow calculation or interpolation. In the actual implementation, the three steps registration, stitching and alignment are run with just one call of the pipeline. Therefore the values remain in memory and the values from the JSON are never used.
 - `orientationsFromAssembler`
 - `rawOrientationsFromRegistration`
 - `orientationsFromRegistration`
 - `anchorImage`
 - `source`
 - `registration`
 - `positionsFromRegistration`

A.1.1. Conventions

All data in the JSON file uses the coordinate system described in Section 2.2 (“standard coordinate system”), with the following exception: “`rawOrientationsFromRegistration`” contains the original rotation, as received from OpenSFM or VisualSFM, in the original coordinate system used by the respective software.

Values received from other software (Hugin, VisualSFM, OpenSFM) are converted into the standard coordinate system for the pipeline. The conventions used by these other software packages are:

Hugin: Rotation is described as Yaw, Pitch, Roll. The rotation order is the same as described in Section 2.2.1. Positive yaw values mean a rotation to the right, negative yaw values to the left. Positive pitch values mean upwards, negative pitch values downwards. Positive roll values represent clockwise rotation.

VisualSFM: X-axis points right, Y-axis points downward and Z-axis points forward. Note that the Y-axis is inverted when compared to our standard coordinate system. Rotation is represented as quaternion in that coordinate system, however the rotation describes the rotation of the world, and not the orientation of the camera. Therefore, to get the orientation of the camera, the

A. File Formats, Directory Structure, Interfaces

quaternion needs to be inverted. Camera position is represented in the global coordinate system and can be used after inverting the Y-axis.

OpenSFM: X-axis points right, Y-axis forward, Z-axis points up. Rotation is represented as a rotation vector x, y, z , which works as follows: The vector describes the axis of rotation, and its magnitude describes the angle. Camera position is described in the rotated coordinate system, so to get the camera position in world space, the position must be transformed using the inverse of the rotation.

A.2. Global Configuration

An optional “global_conf.json” file can be placed in the application working directory, the format of which is described in Listing A.2. If the file exists, all values must be present. Some values correspond to command line parameters. In these cases, the command line parameters take precedence and override the values from the configuration file.

Listing A.2: The JSON format used for the global configuration. Comments are indicated by “//” - this is not valid JSON, and the comments must not be present in the actual file.

```
1 {
2   "config": {
3     "hugin_path": "tools/hugin/",
4     "visalsfm_path": "tools/visalsfm/",
5     "deepflow2_path": "tools/deepflow2/",
6     // Height of the generated equirectangular panoramic image.
7     // Same as command line -s.
8     "equi_height": 2500,
9     // Same as command line -f.
10    "output_file_format": "jpg",
11    // Same as command line -e.
12    "improved_equirectangular_warping": true,
13    // Whether or not to write the intermediate images to disk
14    "write_intermediate_images": true,
15    // Target filename for the output panoramic image.
16    // If empty, a default value is used.
17    "interpolation_target_filename_without_extension": "",
18    // The scene is rendered to a 3D visualization
19    // "visualization.html" for testing purposes.
20    // This parameter determines the resolution
21    // at which panoramic images are drawn into this visualization.
22    "x3dom_panorama_resolution": 30
23  }
```


A.3. Using different Optical Flow Algorithms

To integrate new optical flow algorithms, there are two possibilities:

A.3.1. Option 1: Place files into the “flow” folder

Calculate the flow between pairs of aligned equirectangular images, and put the resulting .flo files into the folder

“sequencename/out/flow/flow_p1_p2_deepflow2_assisted_1.flo”,
 where “p1” and “p2” stand for the aligned equirectangular input images
 “sequencename/out/p1.png” and
 “sequencename/out/p2.png”, respectively.

The .flo file format was used by Scharstein et al. for their optical flow benchmark and is explained in [51]. Source code to read and write this format can be found at their website [50].

A.3.2. Option 2: Replace DeepFlow

First, create a wrapper program that takes the same command line parameters as Deepflow2. Then, name it “deepflow2.exe”, and refer to its folder in the “global.conf.json” file.

The command line parameters for Deepflow2 are:

deepflow2.exe image1 image2 outfile -match matchfile, where:

- image1, image2: Filenames of the input images.
- outfile: Filename of the output .flo file.
- matchfile: Filename of an input text file containing a list of matches in the following format:

Each line refers to a match.

Each line has the format x_a y_a x_b y_b quality counter, where x_a , y_a , x_b , y_b are the coordinates of the feature in the first and second image, respectively. “quality” is a confidence measure with which the feature is weighed during the optimization. This information is not currently

A. File Formats, Directory Structure, Interfaces

available in the pipeline, therefore this is always set to 1. “counter” is a 0-based index of the match.

The file ends with one empty line.

A.4. Using different Registration tools

To replace the registration tool (currently VisualSFM or OpenSFM), there are two options:

- Create a new class `MyRegistration` derived from the `Registration` base class.
- Create the required files in the same format as OpenSFM. Please refer to a sample OpenSFM output for the details of that format. The pipeline uses only the `reconstruction.json` and `tracks.csv` files. From `reconstruction.json`, camera location and position are read. From `tracks.csv`, 2D image features are extracted. The file `tracks.csv` can be empty, but then the optical flow algorithm will have to work without being assisted by matches, which is not recommended.

A.5. Using different Stitching tools

Currently, Hugin is used for stitching. Hugin consists of many different programs, performing isolated functions and operating on project files. The following programs are used by the pipeline:

- “`pto_gen`” to generate a project file and reference the input images.
- “`cpfind`” to find features in images.
- “`pto_var`” to mark specific variables for optimization (in our case, yaw, pitch and roll).
- “`autooptimiser`” to optimise yaw, pitch and roll so that overlapping areas of images are aligned. The information is written to the project file.
- “`nona`” to transform each image to the location in the output image, and save it individually.
- “`enblend`” to blend these individual transformed images together.

To replace Hugin, there are two options: Either you write wrapper programs for each of the executables mentioned above. Alternatively, you can replace the class `HuginAssembler` with your own implementation.

B. How To Use

Follow the following steps to register a scene and generate interpolated views.

1. Please adjust the global configuration file, install the required libraries, and make sure your system fulfills the requirements, as described in the previous Sections. This guide assumes running on Microsoft Windows, and using OpenSfM on Linux.
2. Create a new folder "myscene".
3. If your input consists of standard images:
 - 3.1. Sort the images by viewpoints into folders "myscene/p01", "myscene/p02", ...
4. If your input consists of equirectangular panoramas:
 - 4.1. On the Windows machine, put all panoramas into a folder "myscene/equirectangular"
 - 4.2. On a Linux machine, make sure OpenSfM is installed, e.g. under "/source/OpenSfM"
 - 4.3. Copy the images to "/source/OpenSfM/myscene/images"
 - 4.4. Open a terminal, run the following commands:

```
cd /source/OpenSfM/  
./bin/focal_from_exif myscene  
"vi ./myscene/camera_models.json", change "perspective" to  
"equirectangular", type ESCAPE w q  
./bin/detect_features myscene  
./bin/match_features myscene  
./bin/create_tracks myscene  
./bin/reconstruct myscene  
./bin/mesh myscene
```
 - 4.5. Copy back the files "reconstruction.json" and "tracks.csv" into your folder "myscene" on the Windows machine.
5. Run "panomorph.exe prepare myscene_absolute_path/". Make sure the path is specified using slashes as directory separators, with a terminating slash, and as absolute path.

B. How To Use

6. Run “panomorph.exe interpolate myscene_absolute_path/ -b 1,1,1 -i p01,p02,p03” to prepare optical flow files (if not yet done) and render a new image between camera positions p01, p02 and p03. (Replace p01, p02 and p03 by the names of your images if you used equirectangular input images).

B.1. Parameters for the “prepare” Mode

Passing “prepare” as first parameter runs through stitching (if applicable), registration, and alignment. It generates a “data.json” file in the sequence directory, which must be specified as second parameter.

Parameters available are:

- “-s 2500”: Sets the height of the generated equirectangular panoramas to 2500.
- “-c 0,0,2480,4000”: Sets a crop rectangle to be used for stitching of standard images. This is useful if parts of the panoramic head are visible in each standard image at the same location.

B.2. Parameters for the “calc_flow” Mode

Passing “calc_flow” runs optical flow generation between a specific pair of panoramic images. This is optional, as optical flow generation is automatically run during interpolation if the required flow files cannot be found.

The usage is `panomorph.exe calc_flow sequenceFolder p1 p2`, where `sequenceFolder` refers to the folder containing the dataset, and `p1` and `p2` refer to the pair of images between which optical flow should be generated. Stitching, registration and alignment must already have been run before calling this mode.

B.3. Parameters for the “interpolate” Mode

Passing “interpolate” runs optical flow generation (if the files have not already been generated), and then performs the interpolation. Optical flow files are cached in the folder “sequencename/out/flow” and optical flow generation is only performed if no cached file has been found.

B.3. Parameters for the “interpolate” Mode

Parameters available are:

- “-b 1,1,1 -i p01,p02,p03” interpolates between camera positions specified with “-i” using the barycentric coordinates specified with “-b”
- “-p 2.5,5.5” interpolates at “2.5, 0, 5.5” in the global coordinate system and performs interpolation between three cameras selected using triangulation of the original viewpoints. If the position is outside the space spanned by the original viewpoints, no output is generated. “-p ...” and “-b ... -i ...” are mutually exclusive.
- “-f png” sets the output format to png. Default is jpg. All image formats supported by OpenCV can be specified here.
- “-v y,p,r” extracts a standard image facing the direction specified using yaw, pitch, roll values in radians, e.g. “-v 0.1,0.5,0.05”. “-s width,height” specifies the size of this standard image.

C. How To Build

This chapter explains the build process, both for Microsoft Windows (where the pipeline was originally developed) and for other platforms.

C.1. Mandatory Configuration and System Requirements

The system requires a reasonable amount of RAM, mainly for optical flow generation. For example, 5000x2500px output images require at least 8GB of system RAM. VisualSFM requires a graphics card. The pipeline was only tested on English versions of Windows. On German versions, there may be problems parsing command line arguments, as “,” is used to separate different decimal values from each other, as in “1.23,5.91”.

C.2. On Microsoft Windows

Use Visual C++ 2013 and the provided solution file. The following components need to be installed (obviously, the paths can be changed in the project properties)

- OpenCV 3.0 binaries under `c:/lib/downloaded-build/x64/vc12/bin`
- OpenCV 3.0 include files under `c:/lib/opencv/build/include`
- Boost under `c:/lib/boost`. The following Boost components are used: `filesystem` and `algorithm/string`.
- Eigen under `c:/lib/eigen`.

Deepflow2 is already included, pre-compiled for Windows, and including some DLLs from the Cygwin project that enable it to run. OpenSFM needs to be called manually. It is recommended to use OpenSFM under linux, as it is very hard to compile on Windows.

C.3. Porting to other platforms

The pipeline was developed on Microsoft Windows using the Visual C++ 2013 compiler. However, most code is platform-independent. To build on a different platform, please take the following considerations into account.

All platform-dependent code is in the file `util/Platform.cpp`.

It is necessary to use a reasonably modern C++ compiler, as the source code uses some C++11 features. Some parts of the code are parallelized. If the compiler supports OpenMP, a speedup can be achieved.

Currently, OpenSFM is called manually, because it is very hard to compile on Windows. If compiling on Linux, OpenSFM could be compiled easily and called automatically (`OpenSFMRegistration.cpp`).

D. Used Components, Licenses

Table [D.1](#) shows the components used in the pipeline. Only components used per default, in the final version of the pipeline, are shown. Omitted are some optical flow algorithms that were used for preliminary experiments only.

The table also omits some code snippets that were adapted from various resources. These code snippets concern routine tasks (Windows system calls, string operations) and certain mathematical operations. The corresponding resources are mentioned in the source code.

The license texts for integrated components can be found in the respective folders in the source tree. For the licenses to external libraries, please refer to the corresponding distributions.

OpenCV is used to read and write optical flow files and images. Most, but not all of that functionality is encapsulated in the “MyImage” class.

D. Used Components, Licenses

Component	Purpose	Type of Integration	Author
VisualSFM	Registration	External program	Changchang Wu [67, 68]
OpenSFM	Registration	External program (* see below)	Mapillary corporation & contributors [38]
Hugin	Stitching	External program	Pablo d'Angelo & contributors [9]
Eigen	Math	Linked library	Benoît Jacob, Gaël Guennebaud & contributors [11]
TCLAP	Command Line Parsing	Sourcecode integrated	Michael E. Smootin & contributors [62]
OpenCV 3.0	Image Processing	Linked library	OpenCV Developers Team [37]
Optical Flow Inversion Code	see Section 5.4.5	Sourcecode integrated and heavily modified	Sánchez, Javier and Monzón, Nelson [48]
DeepFlow2	Optical Flow Generation	External program (** see below)	Weinzaepfel et al. [66]
rapidJSON	JSON reading and writing	Sourcecode integrated	THL A29 Limited, a Tencent company, and Milo Yip [43]
boost	C++ library for standard tasks	External library	Contributors [3]
Delaunay triangulation code	Used for selecting cameras to interpolate new views at specific world positions.	Sourcecode integrated	Wael El Oraiby [39]

Table D.1.: Components used in the pipeline. (*) Needs to be called manually, as Windows was used for development, where OpenSFM is very hard to compile on. (**) Compiled via Cygwin, as the code uses some linux-specific C libraries which are not straightforward to port to Windows.

Bibliography

- [1] S. Agarwal, Y. Furukawa, N. Snavely, I. Simon, B. Curless, S. M. Seitz, and R. Szeliski. “Building Rome in a day”. In: *Communications of the ACM* 54 (2011), pp. 105–112 (cit. on p. 26).
- [2] R. C. Bolles, H. H. Baker, and D. H. Marimont. “Epipolar-plane image analysis: An approach to determining structure from motion”. In: *International Journal of Computer Vision* 1.1 (), pp. 7–55. ISSN: 1573-1405. DOI: [10.1007/BF00128525](https://doi.org/10.1007/BF00128525). URL: <http://dx.doi.org/10.1007/BF00128525> (cit. on p. 35).
- [3] *Boost Library*. URL: <http://www.boost.org/> (cit. on p. 90).
- [4] P. Bourke. “Automatic 3D reconstruction: An exploration of the state of the art”. In: *The GSTF Journal on Computing (JoC)*, Vol. 2, No. 3. (2012) (cit. on p. 3).
- [5] M. Brown and D. G. Lowe. “Automatic Panoramic Image Stitching using Invariant Features”. In: *International Journal of Computer Vision* 74.1 (2006), pp. 59–73. ISSN: 1573-1405. DOI: [10.1007/s11263-006-0002-3](https://doi.org/10.1007/s11263-006-0002-3). URL: <http://dx.doi.org/10.1007/s11263-006-0002-3> (cit. on p. 21).
- [6] P. J. Burt and E. H. Adelson. “A Multiresolution Spline with Application to Image Mosaics”. In: *ACM Trans. Graph.* 2.4 (Oct. 1983), pp. 217–236. ISSN: 0730-0301. DOI: [10.1145/245.247](https://doi.org/10.1145/245.247). URL: <http://doi.acm.org/10.1145/245.247> (cit. on p. 22).
- [7] D. J. Butler, J. Wulff, G. B. Stanley, and M. J. Black. “A naturalistic open source movie for optical flow evaluation”. In: *European Conf. on Computer Vision (ECCV)*. Ed. by A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, and C. Schmid. Part IV, LNCS 7577. Springer-Verlag, Oct. 2012, pp. 611–625 (cit. on pp. 36, 38).
- [8] “Clustering Views for Multi-view Stereo / Patch-based Multi-view Stereo (CMVS / PMVS)” software, distributed under GPL license. URL: <http://www.di.ens.fr/cmvs/> (cit. on p. 62).
- [9] P. d’Angelo and contributors. *Hugin - Panorama photo stitcher*. 2015. URL: <http://hugin.sourceforge.net/> (cit. on pp. 22, 90).

Bibliography

- [10] J. Diebel. *Representing Attitude: Euler Angles, Unit Quaternions, and Rotation Vectors*. 2006 (cit. on p. 10).
- [11] *Eigen Library*. URL: http://eigen.tuxfamily.org/index.php?title=Main_Page#Credits (cit. on p. 90).
- [12] A. Eikenes. *Intersection point of lines in 3D space (Matlab Script)*. 2012. URL: <http://www.mathworks.com/matlabcentral/fileexchange/37192-intersection-point-of-lines-in-3d-space/content/lineIntersect3D.m> (cit. on p. 53).
- [13] D. Fleet and Y. Weiss. "Optical flow estimation". In: *Mathematical models for Computer Vision: The Handbook* (2005) (cit. on p. 36).
- [14] Y. Furukawa and P. Moulon. *Clustering Views for Multi-view Stereo / Patch-based Multi-view Stereo (CMVS / PMVS) (modified version)*. URL: <https://github.com/pmoulon/CMVS-PMVS> (cit. on p. 62).
- [15] A. Geiger, P. Lenz, and R. Urtasun. *Optical Flow Evaluation (Online Database)*. 2015. URL: http://www.cvlibs.net/datasets/kitti/eval_stereo_flow.php?benchmark=flow (cit. on pp. 36, 70).
- [16] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. "Vision meets Robotics: The KITTI Dataset". In: *International Journal of Robotics Research (IJRR)* (2013) (cit. on pp. 36, 38).
- [17] P. Getreuer. "Linear Methods for Image Interpolation". In: *Image Processing On Line* (2011). DOI: [10.5201/ipol.2011.g_lmii](https://doi.org/10.5201/ipol.2011.g_lmii) (cit. on p. 49).
- [18] M. Goesele, N. Snavely, B. Curless, H. Hoppe, and S. Seitz. "Multi-View Stereo for Community Photo Collections". In: *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*. 2007, pp. 1–8. DOI: [10.1109/ICCV.2007.4408933](https://doi.org/10.1109/ICCV.2007.4408933) (cit. on p. 3).
- [19] M. Goesele, J. Ackermann, S. Fuhrmann, C. Haubold, R. Klowinsky, D. Steedly, and R. Szeliski. "Ambient Point Clouds for View Interpolation". In: *ACM Trans. Graph.* 29.4 (July 2010), 95:1–95:6. ISSN: 0730-0301. DOI: [10.1145/1778765.1778832](https://doi.org/10.1145/1778765.1778832). URL: <http://doi.acm.org/10.1145/1778765.1778832> (cit. on p. 35).
- [20] R. Goldman. "Understanding Quaternions". In: *Graph. Models* 73.2 (Mar. 2011), pp. 21–49. ISSN: 1524-0703. DOI: [10.1016/j.gmod.2010.10.004](https://doi.org/10.1016/j.gmod.2010.10.004). URL: <http://dx.doi.org/10.1016/j.gmod.2010.10.004> (cit. on p. 10).
- [21] *Google Street View*. URL: <https://www.google.com/maps/streetview/> (cit. on p. 5).

- [22] M. Gupta, R. Kettler, K. Ninomiya, and B. Niu. *Interpolating Depth Panoramas Captured With Kinect for the Oculus Rift*. School of Engineering and Applied Science (cit. on p. 19).
- [23] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Second. Cambridge University Press, ISBN: 0521540518, 2004 (cit. on pp. 13, 51).
- [24] F. A. Hassanat. "A Lightweight, Cross-Platform System for an Immersive Experience in Virtual Exploration of Remote Environments". PhD thesis. University of Ottawa, 2014 (cit. on p. 15).
- [25] B. Horn and B. Schunck. "Determining optical flow". In: *Artificial Intelligence, vol 17* (1981) (cit. on p. 36).
- [26] D. Jelinek and C. J. Taylor. "Quasi-Dense Motion Stereo for 3D View Morphing". In: *International Symposium on Virtual and Augmented Architecture (VAA01)*. 2001, pp. 219–229 (cit. on p. 35).
- [27] B. Klingner, D. Martin, and J. Roseborough. "Street View Motion-from-Structure-from-Motion". In: *Proceedings of the International Conference on Computer Vision*. 2013 (cit. on p. 5).
- [28] S. Kolhatkar. "Real-Time Virtual Viewpoint Generation on the GPU for Scene Navigation". PhD thesis. University of Ottawa, 2010 (cit. on pp. 4, 35, 51).
- [29] J. Kopf, F. Langguth, D. Scharstein, R. Szeliski, and M. Goesele. "Image-Based Rendering in the Gradient Domain". In: *SIGGRAPH Asia* (2013) (cit. on pp. 54, 71).
- [30] A. Kubota, A. Smolic, M. Magnor, M. Tanimoto, T. Chen, and C. Zhang. "Multiview Imaging and 3DTV". In: *IEEE Signal Processing Magazine* 24.6 (Nov. 2007), pp. 10–21 (cit. on p. 5).
- [31] C. Lipski, F. Klose, and M. Magnor. "Correspondence and Depth-Image Based Rendering a Hybrid Approach for Free-Viewpoint Video". In: *Circuits and Systems for Video Technology, IEEE Transactions on* 24.6 (2014), pp. 942–951. ISSN: 1051-8215. DOI: [10.1109/TCSVT.2014.2302379](https://doi.org/10.1109/TCSVT.2014.2302379) (cit. on pp. 35, 51).
- [32] C. Lipski, C. Linz, K. Berger, A. Sellent, and M. Magnor. "Virtual Video Camera: Image-Based Viewpoint Navigation Through Space and Time". In: *Computer Graphics Forum* 29.8 (Dec. 2010), pp. 2555–2568 (cit. on pp. 4, 35).

Bibliography

- [33] B. D. Lucas and T. Kanade. “An Iterative Image Registration Technique with an Application to Stereo Vision”. In: *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2. IJCAI’81*. Vancouver, BC, Canada: Morgan Kaufmann Publishers Inc., 1981, pp. 674–679. URL: <http://dl.acm.org/citation.cfm?id=1623264.1623280> (cit. on p. 36).
- [34] L. McMillan and G. Bishop. “Plenoptic Modeling: An Image-based Rendering System”. In: *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques. SIGGRAPH ’95*. New York, NY, USA: ACM, 1995, pp. 39–46. ISBN: 0-89791-701-4. DOI: 10.1145/218380.218398. URL: <http://doi.acm.org/10.1145/218380.218398> (cit. on pp. 33, 51).
- [35] MeshLab Contributors. *MeshLab - an open source, portable, and extensible system for the processing and editing of unstructured 3D triangular meshes. (A tool developed with the support of the 3D-CoForm project)*. URL: <http://meshlab.sourceforge.net/> (cit. on p. 62).
- [36] Y. Mochizuki and A. Imiya. “Computer Analysis of Images and Patterns: 14th International Conference, CAIP 2011, Seville, Spain, August 29-31, 2011, Proceedings, Part II”. In: ed. by P. Real, D. Diaz-Pernil, H. Molina-Abril, A. Berciano, and W. Kropatsch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. Chap. Multiresolution Optical Flow Computation of Spherical Images, pp. 348–355. ISBN: 978-3-642-23678-5. DOI: 10.1007/978-3-642-23678-5_41. URL: http://dx.doi.org/10.1007/978-3-642-23678-5_41 (cit. on p. 38).
- [37] *OpenCV Library*. URL: <http://opencv.org/about.html> (cit. on p. 90).
- [38] *OpenSfM*. URL: <https://github.com/mapillary/OpenSfM> (cit. on pp. 26, 90).
- [39] W. E. Oraiby. *Relatively Robust Divide and Conquer 2D Delaunay Construction Algorithm*. URL: <https://github.com/eloraiby/delaunay> (cit. on p. 90).
- [40] Point Grey Research, Inc. *Ladybug 3 360 Degree Firewire Camera*. URL: <https://www.ptgrey.com/ladybug3-360-degree-firewire-spherical-camera-systems> (cit. on p. 23).
- [41] Point Grey Research, Inc. *Ladybug 3 Specification*. URL: <https://www.ptgrey.com/support/downloads/10149> (cit. on p. 23).
- [42] S. Pollard, S. Hayes, M. Pilu, and A. Loruss. “Automatically Synthesising Virtual Viewpoints by Trinocular Image Interpolation”. In: (1997) (cit. on p. 33).
- [43] *RapidJSON*. URL: <https://github.com/miloyip/rapidjson> (cit. on p. 90).
- [44] *Raytrix - 3D lightlight camera*. URL: <http://www.raytrix.de/> (cit. on p. 19).

- [45] J. Revaud, P. Weinzaepfel, Z. Harchaoui, and C. Schmid. “Deep Convolutional Matching”. In: *Computing Research Repository* abs/1506.07656 (2015). URL: <http://arxiv.org/abs/1506.07656> (cit. on pp. 38, 62).
- [46] J. Revaud, P. Weinzaepfel, Z. Harchaoui, and C. Schmid. “EpicFlow: Edge-Preserving Interpolation of Correspondences for Optical Flow”. In: *Computer Vision and Pattern Recognition*. 2015 (cit. on pp. 36, 38).
- [47] G. Rubottom. *The Panosaurus Panoramic Tripod Head*. 2011. URL: <http://gregwired.com/pano/Pano.htm> (cit. on pp. 20, 21).
- [48] J. Sánchez and N. Monzón. “Computing Inverse Optical Flow”. In: *Pattern Recognition Letters* 52 (2013), pp. 32–39 (cit. on pp. 46, 90).
- [49] D. Scharstein, H. Hirschmüller, Y. Kitajima, G. Krathwohl, N. Nsic, X. Wang, and P. Westling. “High-resolution stereo datasets with subpixel-accurate ground truth”. In: *German Conference on Pattern Recognition (GCPR 2014)* (2014) (cit. on pp. 36, 38).
- [50] D. Scharstein, H. Hirschmüller, Y. Kitajima, G. Krathwohl, N. Nsic, X. Wang, and P. Westling. *Middlebury Optical Flow Data and Source Code Download*. 2014. URL: <http://vision.middlebury.edu/flow/data/> (cit. on pp. 39, 81).
- [51] D. Scharstein. *README.TXT for Middlebury Optical Flow Benchmark*. URL: <http://vision.middlebury.edu/flow/code/flow-code/README.txt> (cit. on pp. 39, 81).
- [52] A. L. Schwab. “Quaternions, Finite Rotation and Euler Parameters”. In: *unpublished* (2002) (cit. on p. 10).
- [53] S. M. Seitz and C. R. Dyer. “View Morphing”. In: *SIGGRAPH Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 1996, pp. 21–30 (cit. on pp. 33, 51).
- [54] H.-Y. Shum, S.-C. Chan, and S. B. Kang. *Image-Based Rendering*. Springer US, 2007 (cit. on pp. 3, 33).
- [55] S. Sinha, J. Kopf, M. Goesele, D. Scharstein, and R. Szeliski. “Image-Based Rendering for Scenes with Reflections”. In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2012)* 31.4 (2012) (cit. on p. 54).
- [56] A. M. K. Siu, A. S. K. Wan, and R. W. H. Lau. “Modeling and rendering of walkthrough environments with panoramic images.” In: *ACM Symposium on Virtual Reality Software and Technology (VRST)*. Ed. by R. W. H. Lau and G. Baciú. ACM, Mar. 2, 2006, pp. 114–121. ISBN: 1-58113-907-1. URL: <http://dblp.uni-trier.de/db/conf/vrst/vrst2004.html#SiuWL04> (cit. on p. 4).

Bibliography

- [57] J. P. Snyder. *Map Projections: A Working Manual*. U.S. Geological Survey, 1987. URL: <http://pubs.usgs.gov/pp/1395/report.pdf> (cit. on p. 15).
- [58] J. Starck, J. Kilner, and A. Hilton. “Free-viewpoint video renderer”. In: (2008) (cit. on p. 4).
- [59] T. Stich, C. Linz, C. Wallraven, D. W. Cunningham, and M. A. Magnor. “Perception-motivated interpolation of image sequences.” In: *TAP 8.2* (2011), p. 11. URL: <http://dblp.uni-trier.de/db/journals/tap/tap8.html#StichLWCM11> (cit. on p. 42).
- [60] T. Stich, C. Linz, G. Albuquerque, and M. Magnor. “View and Time Interpolation in Image Space”. In: *Computer Graphics Forum Volume 27, Issue 7* (2008) (cit. on pp. 4, 5).
- [61] G. Taubin. “3D Rotations”. In: *IEEE Computer Graphics and Applications, vol.31, no. 6, pp. 84-89* (2011) (cit. on p. 10).
- [62] *TCLAP - Templated Command Line Parser Library*. URL: <http://tclap.sourceforge.net/> (cit. on p. 90).
- [63] *Tech Sneak Peek: Navigating Your Photos in 3D*. Mapillary. URL: <http://blog.mapillary.com/update/2014/12/15/sfm-preview.html> (cit. on pp. 5, 26).
- [64] *The Lytro Immerge professional light field solution*. URL: <https://www.lytro.com/immerge> (cit. on p. 19).
- [65] J. Traa. “Least-Squares Intersection of Lines”. In: *unpublished* (2013). URL: http://cal.cs.illinois.edu/~johannes/research/LS_line_intersect.pdf (cit. on p. 53).
- [66] P. Weinzaepfel, J. Revaud, Z. Harchaoui, and C. Schmid. “DeepFlow: Large displacement optical flow with deep matching”. In: *IEEE International Conference on Computer Vision (ICCV)*. Sydney, Australia, Dec. 2013. URL: <http://hal.inria.fr/hal-00873592> (cit. on pp. 36, 38, 90).
- [67] C. Wu. “Towards Linear-Time Incremental Structure from Motion”. In: *Proceedings of the 2013 International Conference on 3D Vision. 3DV '13*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 127–134. ISBN: 978-0-7695-5067-1. DOI: 10.1109/3DV.2013.25. URL: <http://dx.doi.org/10.1109/3DV.2013.25> (cit. on pp. 26, 62, 90).
- [68] C. Wu, S. Agarwal, B. Curless, and S. Seitz. “Multicore Bundle Adjustment”. In: *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*. 2011, pp. 3057–3064. URL: <http://grail.cs.washington.edu/projects/mcba/pba.pdf> (cit. on pp. 26, 90).

- [69] L. Yang and R. Crawfis. “Rail-Track Viewer – An Image-Based Virtual Walkthrough System”. In: *Eighth Eurographics Workshop on Virtual Environments* (2002) (cit. on p. 3).
- [70] Q. Zhao, L. Wan, W. Feng, J. Zhang, and T.-T. Wong. “Cube2Video: Navigate Between Cubic Panoramas in Real-Time”. In: *Multimedia, IEEE Transactions on (Volume:15 , Issue: 8)* (2013) (cit. on p. 4).

List of Figures

1.1. A generalized pipeline for displaying real scenes in a virtual environment.	2
2.1. Overview of the pipeline.	9
2.2. The yaw-pitch-roll representation of rotation.	10
2.3. Local and global coordinate systems	12
2.4. Standard Photography	14
2.5. Panoramic Photography	15
2.6. Cubemaps	16
2.7. Equirectangular Projection	17
2.8. Cubemaps and Equirectangular Projection	17
3.1. Powershot S100 & Panosaurus.	20
3.2. The Ladybug 3 setup.	23
4.1. VisualSFM and OpenSFM user interfaces	27
5.1. Occlusions and Disocclusions explained	34
5.2. Optical Flow	37
5.3. Input images, intermediate images and the output image.	40
5.4. Diagram showing the steps involved in the interpolation	41
5.5. Holes and Disocclusions, simplified example.	43
5.6. The optical flow inversion and hole-filling algorithms illustrated.	47
5.7. Path of a static scene point in equirectangular projection when moving the camera.	50
5.8. Improved Warping using Epipolar Constraints.	52
6.1. 'Grabenkirche' dataset, showing four viewpoints selected for evaluation.	57
6.2. Influence of original viewpoint distance on output quality	58
6.3. Disocclusion Handling Results	59
6.4. OpenSFM Matches found along the equator only.	62
6.5. Comparing the pipeline with geometry-based rendering.	64
6.6. Example for incorrect disocclusion handling	66