FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

OpenSfM

Ein kollaboratives Structure-from-Motion System

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Matthias Adorjan, BSc
Matrikelnummer 0927290

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Michael Birsak

Wien, 24. April 2016

Matthias Adorjan

Michael Wimmer

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

OpenSfM

A collaborative Structure-from-Motion System

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Matthias Adorjan, BSc
Registration Number 0927290

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Michael Birsak

Vienna, 24th April, 2016

Matthias Adorjan

Michael Wimmer

Erklärung zur Verfassung der Arbeit

Matthias Adorjan, BSc
Heidegasse 2a
7400 Oberwart

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 24. April 2016

Matthias Adorjan

Acknowledgements

Firstly, I would like to thank everyone who supported me during my studies at the Technical University of Vienna and the work on this thesis. In particular, I want to express my gratitude to my advisors Michael Birsak and Michael Wimmer for their helpful input and support during the research and development work of my thesis.

Special thanks to Markus Schütz, who supplied me with valuable advices for the front-end development and background information about his Potree renderer, which we use in this thesis.

Furthermore, I wish to thank the technicians of the Institute of Computer Graphics at the TU Vienna and especially Stephan Bösch-Plepelits for his support on setting up the needed client-server architecture.

At last I would like to thank my family and my girlfriend Katharina, who always stand behind me and supported me patiently during my studies.

Danksagung

Ich möchte mich zunächst bei jedem bedanken der mich während meines Studiums und bei der Erstellung dieser Arbeit unterstützt hat. Besonders möchte ich mich bei meinen Betreuern Michael Birsak und Michael Wimmer für die hilfreichen Ratschläge und für ihre Unterstützung meiner Diplomarbeit und der damit einhergehenden Forschungs- und Entwicklungsarbeit bedanken.

Ein besonderer Dank gilt Markus Schütz, der mir wertvolle Tipps für die Erstellung der Frontend-Anwendung, sowie detaillierte Informationen zu seinem Potree-Renderer gab, den wir in unserer Anwendung verwenden.

Weiters möchte ich mich bei den Technikern des Computergraphik-Instituts der TU Wien, insbesondere bei Stephan Bösch-Plepelits, für seine Unterstützung beim Aufsetzen der benötigten Client-Server-Architektur bedanken.

Zu guter Letzt gilt mein Dank meiner Familie und meiner Freundin Katharina, die während meines Studiums immer hinter mir standen und mich geduldig unterstützten.

Abstract

Besides using high-cost laser scanning equipment to capture large point clouds for topographical or architectural purposes, nowadays other, more affordable approaches exist. Structure-from-motion (SfM) in combination with multi-view stereo (MVS) is such a low-cost photogrammetric method used to generate large point datasets. It refers to the process of estimating three-dimensional structures out of two-dimensional image sequences. These sequences can even be captured with conventional consumer-grade digital cameras.

In our work we aim to a establish a free and fully accessible structure-from-motion system, based on the idea of collaborative projects like OpenStreetMap. Our client-server system, called OpenSfM, consists of a web front-end which lets the user explore, upload and edit SfM-datasets and a back-end that answers client requests and processes the uploaded data and stores it in a database.

The front-end is a virtual tourism client which allows the exploration of georeferenced point clouds together with their corresponding SfM-data like camera parameters and photos. The information is rendered in the context of an interactive virtual globe. An upload functionality makes it possible to integrate new SfM-datasets into the system and improve or extend existing datasets by adding images that fill missing areas of the affected point cloud. Furthermore, an edit mode allows the correction of georeferencing or reconstruction errors.

On the other side the back-end evaluates the uploaded information and generates georeferenced point datasets using a state-of-the-art SfM engine and the GPS data stored in the uploaded images. The generated point clouds are preprocessed, such that they can be used by the front-end's point cloud renderer. After that, they are stored together with the uploaded images and SfM parameters in the underlying database.

On the whole, our system allows the gathering of SfM-datasets that represent different sights or landmarks, but also just locally famous buildings, placed all over the world. Those datasets can be explored in an interactive way by every user who accesses the virtual tourism client using a web browser.

Kurzfassung

Neben der Verwendung von teuren Laserabtastgeräten zur Aufnahme von großen Punktwolken für topographische oder architektonische Zwecke existieren heutzutage leistbarerer Alternativen. Structure-from-motion (SfM) ist in Kombination mit Multi-View Stereo (MVS) so eine kostengünstige photogrammetrische Methode mit deren Hilfe man große Punktdatensätze erzeugen kann. Unter SfM versteht man die Schätzung dreidimensionaler Strukturen aus zweidimensionalen Bildsequenzen. Diese Sequenzen können auch nur mit konventionellen Digitalkameras aufgenommen werden.

In dieser Arbeit wollen wir ein freies, uneingeschränkt zugängliches Structure-from-motion System, basierend auf der Idee kollaborativer Projekte wie OpenStreetMap, aufbauen. Unser Client-Server-System mit dem Namen OpenSfM besteht aus einem Web-Frontend, das es dem Benutzer erlaubt SfM-Datensätze anzusehen, hochzuladen oder zu editieren und einem Backend, das auf Clientanfragen antwortet und die hochgeladenen Daten verarbeitet und in einer Datenbank speichert.

Das Frontend ist ein virtuelles Tourismus-System über das man verortete Punktwolken und die dazugehörigen SfM-Daten wie Kameraparameter oder Fotos erkunden kann. Die Information wird im Kontext eines interaktiven virtuellen Globus gerendert. Eine Uploadfunktionalität macht es möglich neue Datensätze in das System zu integrieren und bestehende Datensätze zu verbessern oder zu ergänzen, indem man ihnen Fotos hinzufügt, die Bereiche der Punktwolke abdecken, die noch nicht erfasst sind. Weiters erlaubt ein Editiermodus die Korrektur von Verortungs- oder Rekonstruktionsfehlern.

Am anderen Ende wertet das Backend hochgeladene Informationen aus und generiert mithilfe eines State-of-the-Art SfM-Systems und den GPS-Daten der Fotos verortete Punktwolken. Die generierten Punktwolken werden vorverarbeitet, sodass sie vom Punktwolkenrenderer des Frontend's verwendet werden können. Danach werden sie zusammen mit den hochgeladenen Fotos und SfM-Parametern in der dem System zugrundeliegenden Datenbank gespeichert.

Im Großen und Ganzen erlaubt unser System das Sammeln von SfM-Datensätzen die über die ganze Welt verteilt sind und die verschiedene Sehenswürdigkeiten oder Wahrzeichen, aber auch nur lokal bekannte Gebäude, repräsentieren. Diese Datensätze können von jedem Benutzer der das virtuelle Tourismus-System über seinen Webbrowser aufruft interaktiv erkundet werden.

Contents

Abstract **xi**

Kurzfassung **xiii**

Contents **xv**

1 Introduction **1**
1.1 Motivation and problem statement 3
1.2 Contribution 4
1.3 Structure of the work 5

2 Related work **7**
2.1 Peer-production-based projects 7
2.2 3D photo exploration applications 15

3 The reconstruction pipeline **23**
3.1 The classic reconstruction algorithm 23
3.2 Alternative approaches 35
3.3 Applications 36

4 Working with large point clouds **37**
4.1 Basic data structures 37
4.2 Large point cloud rendering 40

5 Geographic coordinate systems **47**
5.1 World Geodetic System 1984 (WGS84) 47
5.2 Local ENU reference frame 50
5.3 Coordinate conversions 50

6 Implementation **57**
6.1 System overview 57
6.2 The front-end: A virtual tourism client 58
6.3 The back-end: SfM, point cloud preparation and database interface 73
6.4 The OpenSfM database 81

7 Results **85**
7.1 System 85
7.2 Performance tests 86
7.3 The OpenSfM workflow in practice 88
7.4 Limitations 91

8 Conclusion **93**
8.1 Future work 94

Bibliography **97**

CHAPTER 1

Introduction

One of the major goals in computer graphics research has always been the creation and visualization of realistic models that describe real-world objects. With increasing CPU and GPU power, the visual realism of the digital representatives of such real-world objects has steadily improved.

Nowadays, real-world objects can be virtually explored from arbitrary perspective in real time. This is especially useful if they cannot be investigated in detail without causing damage to them in real world. Furthermore, computer graphics algorithms make it even possible to experience real-world events that cannot be observed in real life any more. This especially occurs in the field of archaeology or cultural heritage, where the observed items suffer from decay or other destructive forces. In order to work with data that describes such real-world objects, it has to be recorded and transformed into a computer-readable representation.

One way to define a real-world object's surface is to describe it as a *point cloud*, which denotes a set of distinct point positions in space. For capturing point clouds, several techniques exist that can be mainly divided into triangulation-based methods and methods that don't use correspondences for surface reconstruction. Additionally, a distinction between techniques that use active 3D sensors (usually laser scanning) and passive image-based photogrammetric methods can be made [25].

The principle of triangulation-based techniques is depicted in Figure 1.1. In this active approach, a laser source emits light that is reflected by the target object and detected by a laser sensor. Since the distance c and angle between the laser and sensor is known, and the incident angle α of the laser at the sensor can be recognized, the distance b from the laser to the scanned target surface can be calculated by means of triangulation. This can be done for each point on the scanned object's surface to generate its digital representation. Due to physical constraints, the accuracy of this approach decreases with increasing distance between the laser source and the target object [12][101].

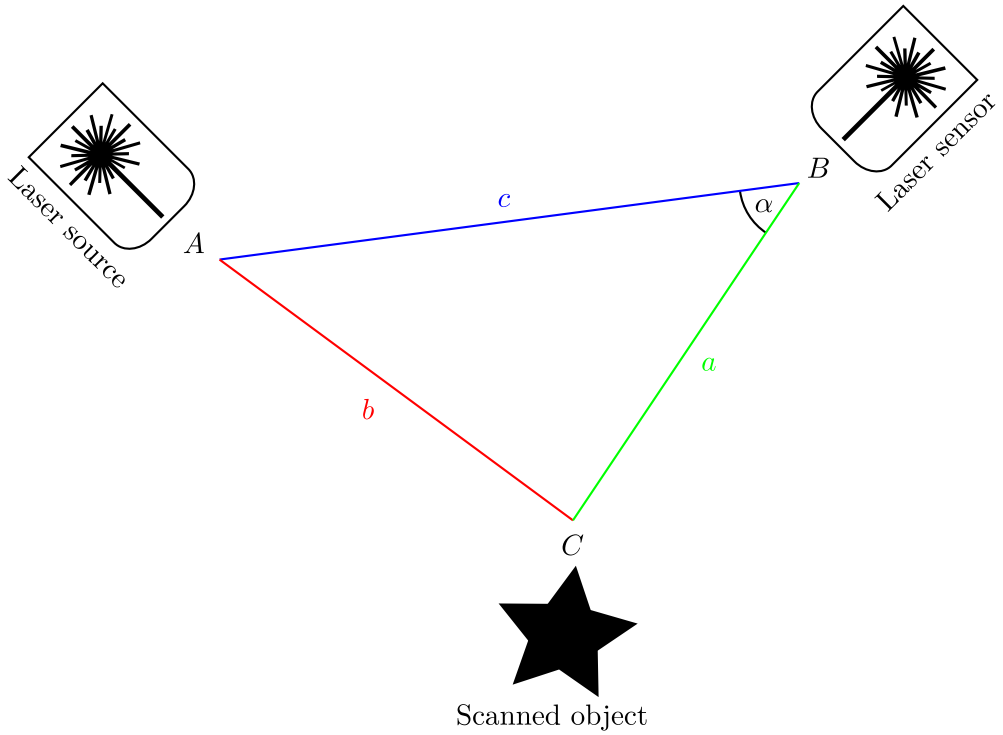


Figure 1.1: Triangulation-based laser scanning: The known positions and orientations of a laser source and its sensor are used by means of triangulation to calculate the distance between the laser source and a scanned object.

For long-range acquisition, other methods like laser pulse-based, also known as time-of-flight laser scanning is used. They send single laser pulses towards the observed object and the time it takes for a pulse to return to the source is measured. Since the speed of light is well known, it can be used together with the laser beam orientation to compute the distance between the laser source and the target object. Figure 1.2 shows the principle of time-of-flight laser scanning. First the laser source emits a laser pulse. This pulse is then reflected at the surface of the scanned object and sent back to the source. After time Δt the laser pulse reaches the source where it is captured. Δt can then be used to compute the distance to the object.

Besides using high-cost laser scanning equipment to capture large point clouds for topographical or architectural purposes, nowadays other, more affordable approaches exist. *Structure-from-motion (SfM)* in combination with *multi-view stereo (MVS)* is such a low-cost passive photogrammetric method used to generate large point datasets. It refers to the process of estimating three-dimensional structures out of two-dimensional image sequences. These sequences can even be captured with conventional consumer-grade digital cameras.

To acquire a point cloud out of a series of photographs, the photos can be fed into a

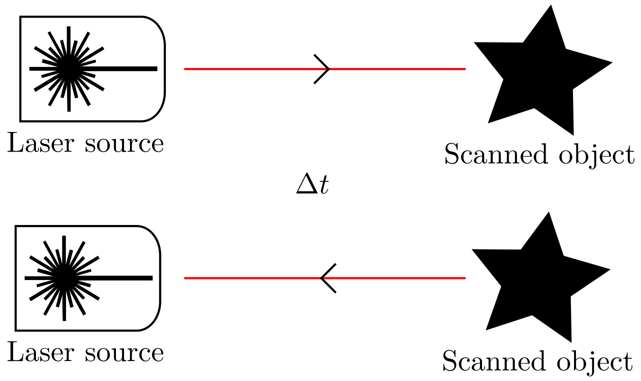


Figure 1.2: Time-of-flight laser scanning: The time Δt it takes for a laser pulse to return to its source is measured and used to calculate the distance between the laser source and the scanned object.

state-of-the-art SfM-engine. Such an engine detects corresponding image points in the given input images and reconstructs three-dimensional points by means of triangulation, as depicted in Figure 1.3. The projection matrices, representing the extrinsic and intrinsic camera parameters, needed for point reconstruction can be computed simultaneously with the reconstruction. Additionally, bundle adjustment algorithms are used to refine the outcome and minimize an appropriate cost function [81]. At last, MVS is used to generate a dense reconstruction using the camera information revealed by the previously executed SfM algorithm. There also exist MVS algorithms that use the points of the sparse reconstruction as an additional input. Details about SfM and MVS algorithms are described in Chapter 3. For the sake of convenience and for easier notation, we use the terms SfM-dataset and SfM-data in this thesis to describe a dataset consisting of extrinsic and intrinsic camera parameters obtained with the help of SfM and the dense scene reconstruction computed using MVS.

1.1 Motivation and problem statement

The resulting point clouds cannot only be used for topographical or architectural purposes, but also for interactive exploration of photo collections in 3D using point cloud rendering software. With the help of structure-from-motion and multi-view stereo, everyone can visualize his photographs in form of three-dimensional point clouds.

The problem is that these point clouds are usually only accessible by the user who created them. They are stored on his local file system, and sharing this information can be difficult, since point clouds are usually stored in very large data files, which makes copy operations time consuming. Furthermore, a lot of different, partly proprietary, point cloud file formats don’t allow easy sharing of point datasets.

We want to solve this problem by establishing a central database for SfM-data, which can be accessed via a front-end by everyone connected to the internet. Our front-end

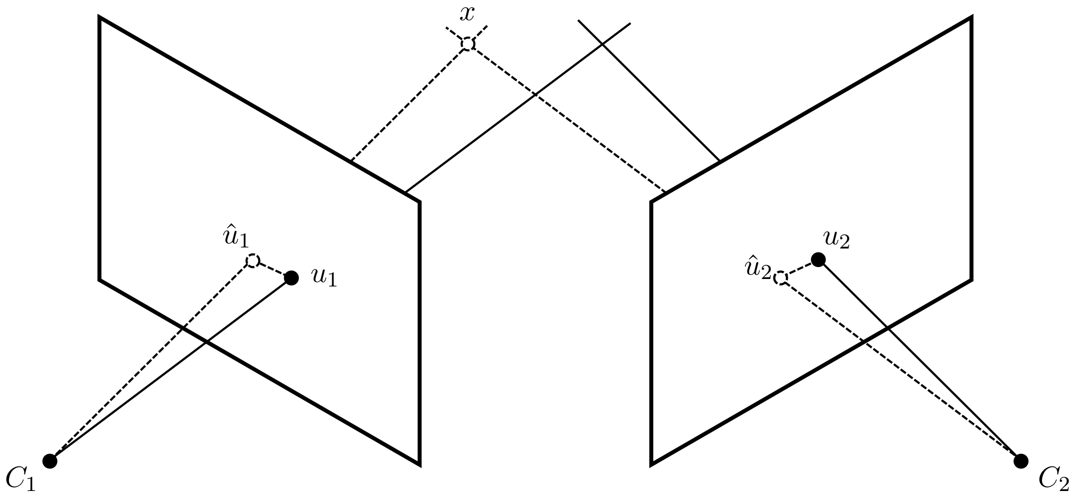


Figure 1.3: Triangulation in structure-from-motion algorithms: A 3D point x can be computed from the viewpoints $(u_1, u_2, ...)$ measured from different views $(C_1, C_2, ...)$ by intersecting their back-projected viewing rays, which are visualized as solid lines in the figure. Due to measurement errors this rays usually don't intersect exactly. Therefore x is the 3D point, which minimizes the sum of squared errors between the measured and calculated viewpoints $(\hat{u}_1, \hat{u}_2, ...)$. This figure is redrawn as seen in [81].

serves as a virtual tourism client and can be used to explore important sights and cultural heritages, but also locally famous buildings, like small old churches, mills or statues in a virtual way by using the point clouds stored in our spatial database. Additionally, a user can share his own photographs and point clouds with the help of our system in order to make them accessible to other users. Our system obviates the need for manually copying large point cloud files between different clients and allows to maintain additional information about the stored photographs and point clouds.

Another problem we target is that recent work on 3D reconstruction just results in 3D models of important buildings and landmarks (e.g. [1]). This is due to the fact that the authors usually use online photo-sharing platforms like Flickr as image source. On such sites one can generally just find photos of famous sights, because sideways and locally important buildings are rarely photographed and therefore not present. Our web front-end offers an upload possibility, where users can import photos of local regions and reconstruct them in 3D. Additionally, it makes it possible to fill holes of existing models by adding missing images to the datasets.

1.2 Contribution

The main contribution of this thesis is the development of a free, fully accessible system for efficiently storing and visualizing SfM-datasets, based on the idea of the collaborative

OpenStreetMap project [72], called *OpenSfM*.

During the work on this thesis we establish a spatial database, which stores SfM-data, like extrinsic and intrinsic camera parameters and input images, together with the resulting point clouds. A web application serves as a user interface to the underlying database and allows the exploration and modification of the stored datasets in an interactive way. Additionally, the front-end makes it possible to feed an SfM-engine at the server, which generates point clouds out of uploaded image data that are then saved to the database.

1.3 Structure of the work

This thesis is built up of different chapters. After the introducing words of this chapter there are following upcoming parts:

Chapter 2: Related work
This chapter discusses various free-access, free-content and collaborative systems, which serve as a model for our idea of such a system for SfM-datasets, like *OpenStreetMap* [72]. Furthermore, some existing structure-from-motion based 3D photo-gallery approaches are reviewed in more detail.

Chapter 3: The reconstruction pipeline
In this section the basic reconstruction pipeline is introduced together with some alternative approaches.

Chapter 4: Working with large point clouds
This chapter deals with different methods for storing and rendering large point clouds, especially those that are used in our framework implementation.

Chapter 5: Geographic coordinate systems
Since our application georeferences the 3D reconstructions and presents them on a globe, one must know the fundamentals about geographic coordinate systems to understand the implementation details described in the following chapter. This section covers the basics of map algebra concerning the algorithms used in our implementation.

Chapter 6: Implementation
Here the implementation of our framework is described in more detail and an overall view on the code structure is given.

Chapter 7: Results
This part of the thesis shows the results of our implementation. The front- and back-end features are described in more detail and executed performance tests are presented here.

Chapter 8: Conclusion

Finally, the conclusion sums up the thesis and introduces some suggestions on what can be done in future works regarding the implemented framework.

CHAPTER 2

Related work

Since this thesis affects various areas of computer science, a broad theoretical background knowledge is needed to understand it. This chapter gives an overview of other projects that are based on the same idea of a so-called peer production model, where a community provides freely usable and editable data. Furthermore, other photo-gallery systems based on structure-from-motion and/or multi-view stereo are introduced and reviewed. Reconstruction and point cloud specific details are described in Chapter 3 and Chapter 4.

2.1 Peer-production-based projects

The new possibilities that came with the introduction of the Internet allow the collaboration of people across the world on very large virtual projects. Compared to collaborative works in the Pre-Computer-Age, those virtual projects are novel in terms of scope and space. While important projects in human history like the Egyptian pyramids or the Great Wall of China were realized by thousands or ten-thousands of workers, large virtual projects nowadays have millions of contributors. To implement such large projects successfully, a new working process is needed. Yochai Benkler, Professor of Law at the Yale Law School, calls this process *peer production* and describes it as follows [10]

Definition 1 *Peer production describes a process by which many individuals, whose actions are coordinated neither by managers nor by price signals in the market, contribute to a joint effort that effectively produces a unit of information or culture.*

Science can be defined in exactly the same way. Scientists usually don't follow direct research orders by their Dean or work according to market signals. They work independently on a specific problem and finally bring their solutions together. The difference between scientific work and peer-production projects is the importance of the resulting product to the broad mass of people [10].

Free software based on peer-production is widespread over the Internet. The nowadays most frequently used web server, the Apache web server, is a product of collaborative software development [70]. Not just free software is based on peer-production, most of the content accessible via the Internet can be seen as the result of a very large project involving a collaborative community. While free software is implemented by people who share their thoughts through the Internet, user-generated content, which makes up the World Wide Web, is created directly on it. Generally, such collaborations can be classified into three types: *Social spaces*, *Content spaces* and *Cosmographies* [39].

Social spaces are mainly social networks that have emerged in the last few years. People that join such networks can upload and share personal information, videos or music clips with their friends, family or colleagues. The biggest and most popular social network, in terms of the number of contributors, is Facebook with an average of 1.39 billion monthly active users in 2014 [26]. Besides social networks, there are other systems that are focused on establishing platforms for social interaction. These include online forums and chatrooms, but also massively multiplayer online role-playing games (MMORPGs), like World of Warcraft, which reached its highest subscriber count of 12 million players in 2010 [13]. In such games, users can communicate with each other in an online universe with the help of controllable avatars.

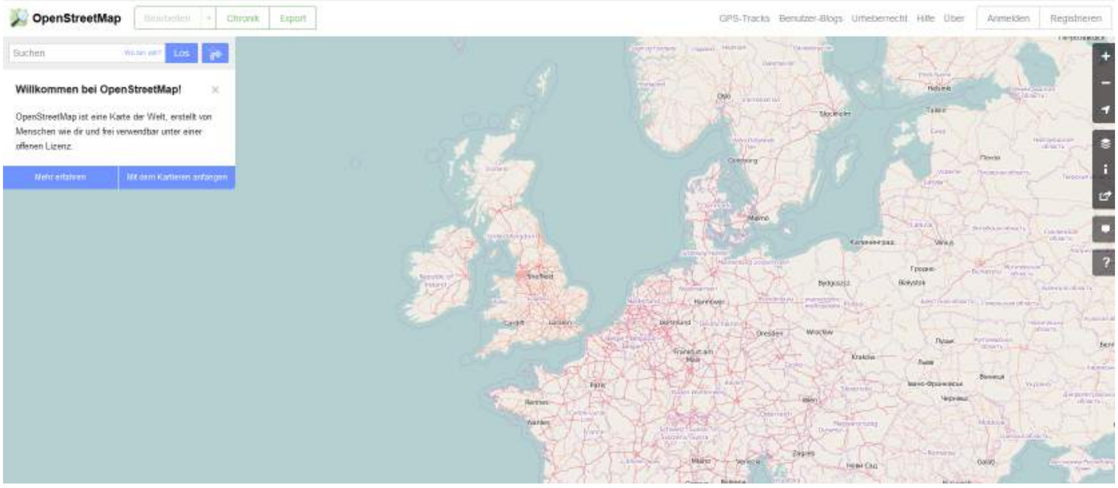
In content spaces, large amounts of information is stored. This information is uploaded and downloaded, or produced and consumed, by the users of these spaces. Therefore, Ritzer and Jurgenson call these users *prosumers* [80]. Photo- and video sharing platforms, like Flickr, Instagram or YouTube are the most important content spaces. While Instagram is used by over 400 million users to share more than 80 million photos a day, on YouTube approximately 400 hours of video material is uploaded every minute in 2015 [54][104]. Another form of content space is the online encyclopaedia Wikipedia, which is built up on the wiki collaboration model. This model allows anyone to add, edit or delete content from a Website. Wikipedia is the largest online encyclopaedia with about 36.4 million articles in 246 languages [113]. For more details about Wikipedia see Section 2.1.2.

Cosmographies contain information about the Earth, as their central element. Platforms like Google Maps store different types of information, like photographs, video and user comments that represent places of the real world. The OpenStreetMap database contains more than 4.9 billion GPS coordinates produced by approximately 2.4 million users, which are used to create a virtual map of the world (see Figure 2.2) [73]. Wikipedia can't be only seen as a content space, but also as a cosmography. There exist thousands of encyclopaedia entries that describe real world places in detail.

Our project can be classified into the categories cosmographies and content spaces, since we store the geographic locations of the generated point clouds together with useful information about them.

The following sections give an overview about two well known projects based on a collaborative community, namely OpenStreetMap and Wikipedia.

2.1.1 OpenStreetMap


Willkommen bei OpenStreetMap!

OpenStreetMap ist eine Karte der Welt, erstellt von Menschen wie dir und frei verwendbar unter einer offenen Lizenz.

Figure 2.1: The OpenStreetMap browser interface allows easy access to the underlying spatial data. The data can be viewed or edited after logging into the OSM system. [72].

The goal of the *OpenStreetMap (OSM)* project, founded by Steve Coast, is the creation and distribution of a freely available and editable map of the world. The access to spatial data has been made easier since the launch of this project in 2004. Until then the creation and distribution of cartographic material was reserved to governmental agencies or cartographic organizations. Surveyors, geographers and cartographers travelled all over the world to collect spatial information and transcribed it to paper. A lot of knowledge and expensive equipment was needed to map the Earth onto paper or enter the measured information into a computer system. Anyone who wanted access to the recorded geodata had to pay high license fees and the usage was restricted by several copyright rules. These barriers made it impossible for private persons or small companies to access high-quality geospatial information [5].

With the removal of selective availability of the *Global Positioning System (GPS)* signal in 2000, it became easier to collect spatial data. The GPS was originally developed for military usage by the U.S. Department of Defense. The project started in the 1970s and the system became fully operational in 1995, when it used 24 satellites for location determination. The constellation with 24 satellites ensure that from every point on earth 4 satellites are visible. This fact allows to derive a users 3D position by using the known positions of the four visible satellites. The GPS satellite constellation was expaneded recently, such that today 31 operational satellites are used for global positioning. The additional satellites are used for precision improvement and failure safety. Until the year 2000, the high-precision GPS signal was reserved for U.S. military use only, while the civilian GPS signal was deliberately degraded. With the removal of this degradation it is nowadays possible to determine one's position with better than 3.5 meters horizontal accuracy. With the help of Differential GPS (DGPS) an accuracy of distincly lower than

1 meter can be achieved. Here an additional ground station, with a well known location and a GPS receiver is used. As long as the measured location is near to a ground station, the inaccuracies in the measurements are similar for the GPS receiver in the ground station and the GPS receiver at the interested location. Therefore this inaccuracy can be easily subtracted out of the computed geolocation, which at last improves the accuracy of the final result [75][107][106].

The free availability of this high-precision GPS signal, affordable simple GPS receivers, the rise of the personal computer and the Internet made it possible to create a system like OSM. Since the beginnings of OSM in 2004, the numbers of registered users and contributions has increased steadily to approximately 2.4 million users and about 4.9 billion track points in November 2015, as can be seen in Figure 2.2 [73].

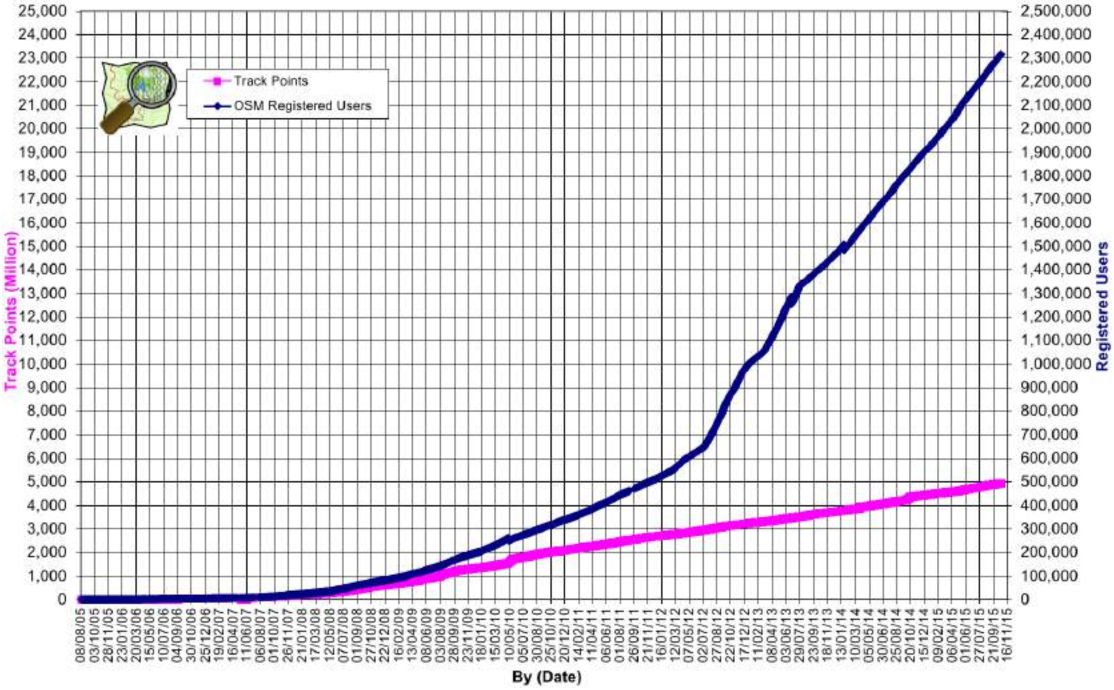


Figure 2.2: A graph showing the rapidly increasing user and contribution numbers of OSM [73].

Most of the registered users use the provided OSM technical infrastructure for data administration and collaboratively editing of the world map to follow the goals of OSM. Furthermore, there exists a core group of volunteers and software developers which dedicates its spare time to steadily improve the OSM infrastructure, maintain the servers, implement additional front-end features, and so on.

The OSM infrastructure consists of several distinct components as depicted in Figure 2.3. They can be classified into geodata, editing software, backend, rendering algorithms and visualization layer.

When visiting the OSM website `www.openstreetmap.org`, an online mapping interface becomes visible, which shows rendered OSM data. This interface is called *Slippy Map*, which uses the JavaScript APIs OpenLayers or Leaflet to display so called map tiles. Those tiles are prerendered squared raster images, which represent a map when displayed in a grid arrangement.

The tiled map data is rendered by an open source renderer called *Mapnik*. The rendering process runs on the tile server `tile.openstreetmap.org`, which also stores the resulting rasterized tiles. Mapnik uses the formatted data stored in a PostgreSQL database with PostGIS extension. This database runs on the tile server too and is filled by the *osm2pgsql* script which converts the OpenStreetMap data stored on the core OSM database server to a format, that can be handled by the renderer. This conversion is done on minutely diffs, so called *planet diffs*, which are created by a Java-based commandline application called *osmosis*. Besides the Mapnik renderer, these diffs are also used by other parts of the OSM infrastrucure. *Nominatim*, for example, is a search tool that uses these planet diffs to filter the whole OSM dataset by name and address.

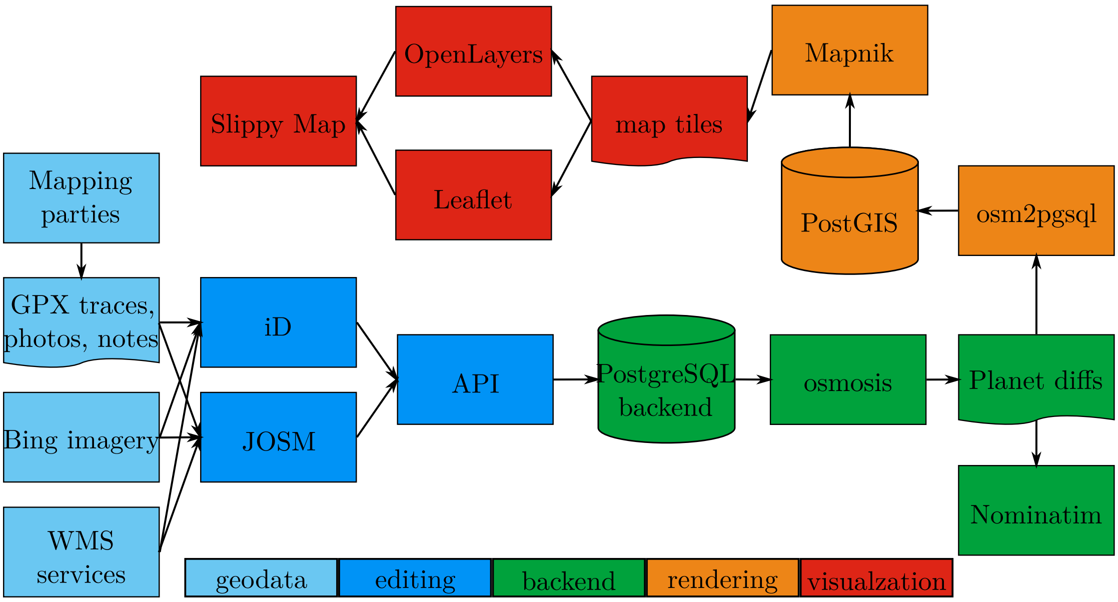


Figure 2.3: Sparse overview of the OSM infrastructure. Diagram simplified redrawn as seen in [73].

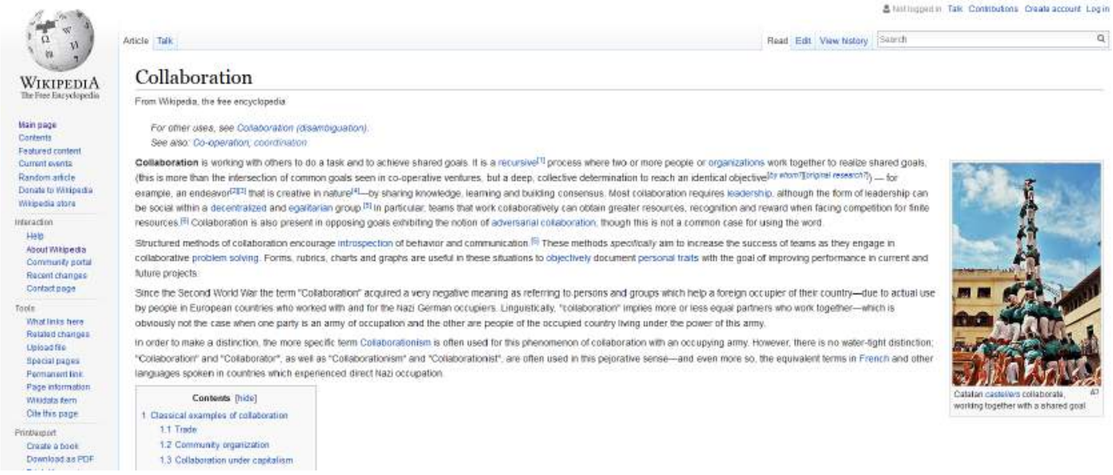
The OSM dataset is maintained with the help of editors. There exist several OSM editors, which can be used with an OSM account. The default application, which is launched when clicking the edit-button at the OSM webpage is called iD. It is a Javascript-based web application, which uses the OSM API to import spatial data and allows the editing of existing data, like nodes, ways, relations or metadata tags. Another popular OSM data editor is JOSM (Java OpenStreetMap Editor), which is a feature-rich cross-platform desktop application that works browser independent.

OSM supports a lof of data formats for import. The most common are GPX traces (GPS eXchange format or GPX) which are recorded by GPS devices that support the GPX interchange standard for geodata. Fortunately this standard is widely supported - even smartphones with builtin GPS receivers can nowadays save the recorded spatial data into GPX-files. Besides GPS coordinates, raster images can be imported into the system too. Here, different interfaces to other geoinformation systems (GIS systems) exist. For example, OSM supports the web map service (WMS) standard protocol, which allows the import of georeferenced map images, either in raster or vector graphics format. Additionally, other imagery services are supported, such as Microsoft's Bing maps. Mapping parties are another signature feature of OSM. These parties are local workshops organized by the OSM community, which have the goal to create content for specific local areas and establish local user groups to build up a community around the project. One of the first parties was on the Isle of Wight in May 2006, where more than 30 participants collected spatial data about all the roads and paths of the island by moving around with GPS receivers. The geodata was at last imported into the OSM system, which resulted in a nearly complete map of the island [45].

A lot of studies are published in literature that analyse the quality of *volunteered geographical information (VGI)*, as it is termed in [38]. Haklay et al. showed in their work that the quality of VGI can be very good. They compared the OSM datasets to the Ordnance Survey (Britain's mapping agency) datasets in the UK. When evaluating the data, they identified several different quality aspects, which they used to compare the different datasets. The analysis revealed that the OSM data has an accuracy of about 6m and the motorways and main roads referenced in the OSM road maps are nearly identical to the Ordnance Survey's road information. However, the analysis also showed the inconsistency of the OSM datasets in terms of quality. There are places where geographical information is collected and digitised very accurately, while other areas lack information or are described imprecisely. For more details about the comparison and analysis work, see [44]. Ciepluch et al. compared five areas of Ireland represented as OSM datasets to the corresponding datasets of Google Maps and Bing Maps [17]. Since Google and Microsoft don't provide the vector data behind their online maps, it is not possible to do a vector based comparison like Haklay et al. did in their analysis. The authors solved this problem by exporting roads and point-of-interests from OSM to a spatial format called KML. Furthermore, a web application was developed which allows the overlay of KML files over Google Maps and Bing Maps. This setup gives them the opportunity to do a visual comparison of the examined geodata. They finally concluded, that none of the mapping systems are accurate over all five studied locations and all the platforms have their advantages and drawbacks. OSM suffers from the fact, that there are regions, especially in the countryside, where nobody wants to collect GPS data [17].

Nonetheless, with the help of an ever-growing community and a modern technical infrastructure, the OSM project is well on the way to achieve its goal of a freely available map of the whole world.

2.1.2 Wikipedia



Collaboration

From Wikipedia, the free encyclopedia

For other uses, see Collaboration (disambiguation).
See also: Co-operation, coordination

Collaboration is working with others to do a task and to achieve shared goals. It is a recursive[1] process where two or more people or organizations work together to realize shared goals, (this is more than the intersection of common goals seen in co-operative ventures, but a deep, collective determination to reach an identical objective[by whom?][original research?]) — for example, an endeavor[2][3] that is creative in nature[4]—by sharing knowledge, learning and building consensus. Most collaboration requires leadership, although the form of leadership can be social within a decentralized and egalitarian group.[5] In particular, teams that work collaboratively can obtain greater resources, recognition and reward when facing competition for finite resources.[6] Collaboration is also present in opposing goals exhibiting the notion of adversarial collaboration, though this is not a common case for using the word.

Structured methods of collaboration encourage introspection of behavior and communication.[6] These methods specifically aim to increase the success of teams as they engage in collaborative problem solving. Forms, rubrics, charts and graphs are useful in these situations to objectively document personal traits with the goal of improving performance in current and future projects.

Since the Second World War the term "Collaboration" acquired a very negative meaning as referring to persons and groups which help a foreign occupier of their country—due to actual use by people in European countries who worked with and for the Nazi German occupiers. Linguistically, "collaboration" implies more or less equal partners who work together—which is obviously not the case when one party is an army of occupation and the other are people of the occupied country living under the power of this army.

In order to make a distinction, the more specific term Collaborationism is often used for this phenomenon of collaboration with an occupying army. However, there is no water-tight distinction; "Collaboration" and "Collaborator", as well as "Collaborationism" and "Collaborationist", are often used in this pejorative sense—and even more so, the equivalent terms in French and other languages spoken in countries which experienced direct Nazi occupation.

Catalan castellers collaborate, working together with a shared goal

Contents [hide]

1 Classical examples of collaboration
  1.1 Trade
  1.2 Community organization
  1.3 Collaboration under capitalism

Figure 2.4: Description of the term “Collaboration” in Wikipedia [116].

Wikipedia as another peer-production-based project has long since established itself as a multilingual, free-content online-encyclopaedia. As of September 2015 Wikipedia has about 370 million unique visitors each month, which makes it one of the most popular websites world-wide [112].

Wikipedia is based on wiki software, which was developed by Cunningham in 1995 [60]. His WikiWikiWeb tool, whose name is based on the abbreviation for world-wide-web “WWW” and the Hawaiian word “wiki” for “quick”, allows the direct editing of HTML documents by anyone, who has access to them. Each change is documented and a revision history is stored, which allows the comparison of different versions of a document [108]. Unlike OpenStreetMap, which allows only registered users to edit its map, Wikipedia follows an open-for-all way. It is not needed to login before editing an article. A simple click on the edit-button allows everyone, who is visiting the website to change its content, like article prose, images, references or others. However, it is also possible to register as a user on Wikipedia. When logged in, while editing articles, the edits are attributed to the registered user’s unique user name. Otherwise the IP address of the user will be made publicly visible, when document changes are made [114].

The principles and rules of Wikipedia are expressed in several essays. There exists no single definition of the values and principles of Wikipedia. One of the most popular summaries of what to consider when using Wikipedia are the so called *five pillars* of Wikipedia, which consist of the following statements [115]:

1. Wikipedia is an encyclopaedia
2. Wikipedia is written from a neutral point of view
3. Wikipedia is free content that anyone can use, edit, and distribute

4. Editors should treat each other with respect and civility
5. Wikipedia has no firm rules

The first pillar expresses that Wikipedia is an encyclopaedia that possesses many features of traditional, printed encyclopaedias or almanacs, like the *Encyclopeadia Britannica*. It is not a dictionary, news paper or an advertising platform.

The *neutral point of view (NPOV)* introduced in the second statement, is a key principle of Wikipedia. Articles should be written from a neutral point of view, which means that multiple points of view on topics should be represented in an accurate, factual and objective way. To achieve a neutral sight on a topic, no point of view should be seen as the best or the truth. Additionally, it is useful to describe the disputes between different views, rather than taking sides in them. The content of articles must be verifiable and reliable, authoritative sources have to be cited.

The third principle prohibits copyright infringement. It states that copyright laws have to be respected and plagiarism is not allowed. The content of Wikipedia is free, such that anyone can use, edit and distribute it according to the terms of the Creative Commons Attribution-ShareAlike 3.0 Unported License (CC BY-SA) and the GNU Free Documentation License (GFDL). This means that articles and images are not owned by anyone and are freely licensed to the public. It is recommended to use free media when composing Wikipedia pages, however, it is possible to use non-free or fair-use media, but this should be avoided.

The fourth pillar is to respect other contributors and follow the Wikipedia etiquette, which among others states that one should treat fellow Wikipedians as one would like others to treat oneself. Furthermore, good faith can be assumed and one has to be polite and calm, when discussing conflicts on talk pages, which exists beside every article. This rule is easier written than done, because it is not always followed by every contributor. There can be one, who attacks others, for example by means of edit wars, where changes of entries got immediately reverted out of spite. It such cases, it is hard to respect the other contributor. However, over time Wikipedia developed rules and mechanisms to temporarily or permanently ban such “difficult” users. A collective of administrators, bureaucrats and stewards, which is elected by the community, monitors and maintains the Wikipedia project. Users can be banned by the community by consensus, an elected Arbitration Committee, the board of the Wikimedia Foundation and by the Wikipedia founder Jimbo Wales.

“Ignore all rules (IAR)” is the fifth principle that should be followed, when a rule prevents the improvement of Wikipedia’s content. There exist several Wikipedia guidelines and policies regarding the content of articles, etiquette and behaviour, the deletion of pages, the enforcement of defined standards and processes behind the project. Furthermore legal policies concerning copyrights or terms of use are established. The IAR rule states that nobody needs to read every Wikipedia policy before editing content. Contributors should simply edit and write articles without always thinking about policies and guidelines.

However, IAR doesn't mean that anyone can fool around and act like an idiot. The basic principles, like civility, still apply and IAR should always be used with an explanation, why it has been used.

The above explained principles, the democratic structure and thousands of volunteers allowed the establishment of a huge online encyclopaedia. The mobilization of so many contributors has proved a key to the success of Wikipedia. According to the research and findings of [52] there are three propositions that explain how Wikipedia was able to attract a large number of volunteers. Firstly, Wikipedia has always defined itself clearly as an encyclopaedia and nothing more (see first pillar above). Second, it is easy to contribute to Wikipedia. While correcting small mistakes in written encyclopaedias is often inconvenient, time-consuming and expensive (e.g. sending letter to publisher), fixing typos or content in Wikipedia can be done with a simple click on the edit button. Not even an account is necessary to edit articles of the online encyclopaedia, which makes contributing even more attractive. At last, Wikipedia offers "low social ownership of content". The authorship of articles is not directly emphasized and there is no ranking of contributors or reward system for contributions. This releases volunteers from the obligation to contribute in order to achieve a specific reward or reputation. However, there are other incentives to contribute to Wikipedia as found in [30]. Authors try to write so called featured articles, which appear, when accepted, on the main page of Wikipedia for a day. Although a Wikipedia article doesn't explicitly belong to a single author, users often claim authorship of articles on their user page to indicate that they have done the most work on them. In the end, one can say that reputation or credits still create the greatest incentives for contribution to peer production systems like Wikipedia, as mentioned in most of the literature on peer production, even if those rewards are not visible at first appearance [59].

Of course there is not always consensus and there will always be vandalism and edit wars, but on the whole the Wikipedia project can be seen as a complete success.

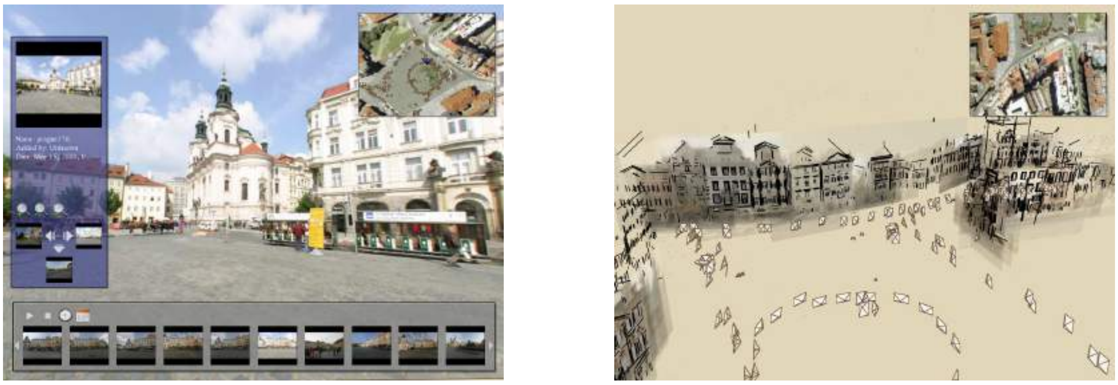
2.2 3D photo exploration applications

Our frontend-idea is based on a series of projects on community photo collections introduced at the University of Washington. As already stated in Section 2.1, online photo platforms like Flickr have a growing community and billions of images are stored on their servers. This fact is used by Snavely et al. for their work on 3D reconstruction, visualization and image-based rendering [99].

They developed computer graphics algorithms, which are able to handle this huge amount of photos taken from different viewpoints under several lighting conditions. Most of the applications introduced by Snavely et al. are based on 3D scene reconstruction, where, with the help of structure-from-motion (see Chapter 3), a geometric representation of the scene is computed out of given input images. These input images can be taken from online photosharing platforms like Flickr, but also from private offline photo albums. This scene representation describes the camera pose of each photo, consisting of the position and orientation of the camera in 3D space, a sparse scene reconstruction visualized by a 3D point cloud and the information about which 3D point corresponds to which underlying photograph. The gathered information, which they call midlevel representation, is then used in their software for tasks like path finding between camera views and creation of dense reconstructions, where an accurate point cloud representation of the 3D scene is calculated [94] [99].

The following subsections describe applications designed for 3D photo exploration and are strongly related to our work.

2.2.1 Photo Tourism



(a) Full-resolution photo, that appears, when a user selects a photograph.

(b) Non-photorealistic rendering of a top-down-view on the SfM-dataset representing Prague.

Figure 2.5: Screenshots of the Photo Tourism application's 3D interface. Image courtesy of Snavely et al. [97]

The application software called "Photo Tourism", developed at the GRAIL, the Graphics and Imaging Laboratory of the University of Washington, offers a novel 3D interface for browsing unordered, large photo collections [97]. Each image is placed at its corresponding view point in the 3D scene. The user can traverse through these images by visiting each viewpoint, which is computed using SfM algorithms. The viewpoints and directions of the cameras are visualized by camera frusta. When selecting a camera, its image gets rendered with the help of texture mapping at the backside of the frustum. The photo is displayed opaque and in full-resolution, such that the user can see every detail of the original image (cf. Figure 2.5a).

The movement between two cameras is calculated by linearly interpolating between the camera positions, orientations and field of views. During camera transition, camera view interpolation is used to display in-between images. To compute these in-between images, two techniques are implemented, namely triangulated morphs and planar morphs. The first mentioned technique uses 2D Delaunay triangulation to generate a 3D mesh for each of the two involved images and their corresponding reconstructed points. The images are then texture-mapped onto their computed 3D mesh. Finally, the in-between view is created by rendering each mesh from the current viewpoint. The two rendered images are blended together using the distance from the viewpoint to the appropriate endpoints as blending weights. Planar morphing between two cameras is done by projecting the two images onto a common plane of their underlying points. The in-between views are simply created by cross-fading between the projected images.

There exist several navigation functions within the application. Besides free-flight navigation the system offers object-based browsing and moving between related views. The related images of a currently displayed one are computed by projecting the points of the active image into other photos and evaluating the projected motion of these points. If the points have moved right, the other photograph is considered to be on the left side of the currently viewed image. Object-based navigation allows to search all images which depict an object selected by dragging a 2D box around an area of the currently shown photo or point cloud. The system uses the reconstructed corresponding 3D points for the search for suitable candidates.

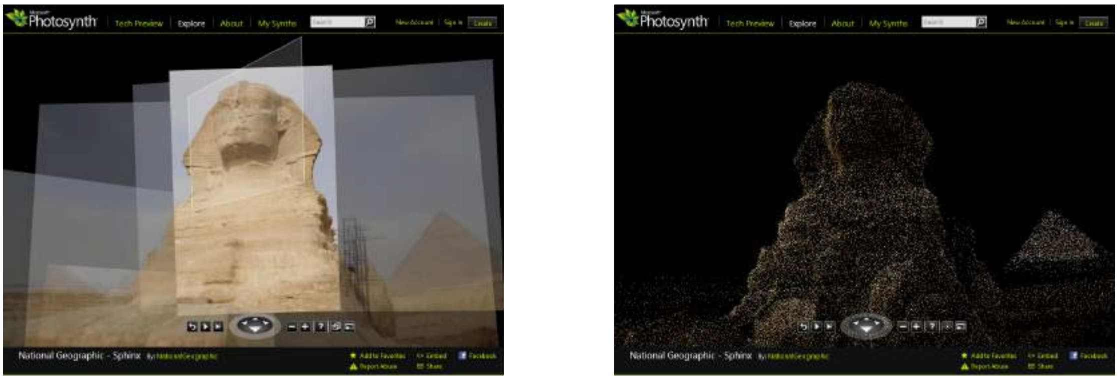
The underlying SfM algorithm used by the application for 3D point reconstruction is based on an early version of the *Bundler* SfM system [95]. The reconstructed sparse 3D point cloud is only used as a background for the rendered images and to model correspondences between the viewpoints of each photograph and the scene. In contrast, our system gives the computed point clouds a more important role, since we emphasize their location in the world (cf. Chapter 7). Additionally, the “Photo Tourism” software offers a non-photorealistic mode, which allows a more attractive overview of the scene, as can be seen in Figure 2.5b.

The system also offers the possibility to extend the point cloud or photo collection with new points or photos, by running the SfM algorithm just at a local level, involving just a part of the underlying images. Furthermore, it is possible to annotate image regions. These annotations are transferred to similar photos, which depict the same areas of the scene.

2.2.2 Photosynth

Besides the above described application there exists other work on 3D scene reconstruction and photo exploration. *Photosynth*, originally developed by Microsoft’s Live Labs, is such an example [78]. This project was inspired by the research on “Photo Tourism” and its algorithms are partly based on the work described in [94] and [96]. There currently exist three different generations of Photosynth technology.

The first generation of Photosynth offers the user the ability to create and navigate through so called “Synths”. The term stands for 3D scene reconstructions created with the help of SfM algorithms. These “Synths” are generated with the help of an offline desktop application and then uploaded to Microsoft’s Photosynth website or even embedded in own blogs, websites or posted on social media platforms like Facebook. The viewing software, based on Microsoft Silverlight (cf. [67]), offers nearly the same navigation possibilities and user controls as the “Photo Tourism” application. The user can view the different photos and close-ups by simply moving the mouse cursor around the screen. If he enters a region, where a picture is available, a semi-transparent outline appears (cf. Figure 2.6a). When the user clicks on this outline the high-resulting image is shown, surrounded by the other photos nearby. It is also possible to get a 2D grid view of all the photographs contained in the “Synth”. Furthermore, the user can switch to the point cloud view, where the sparse scene reconstruction is visualized, as depicted in Figure 2.6b.



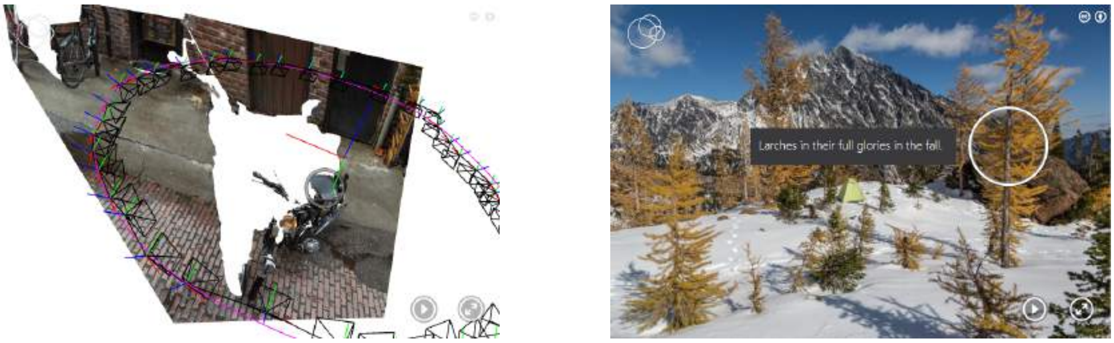
(a) “Synth” of a Sphinx showing the selected photo and the semi-transparent nearby images as context.

(b) The point cloud view of the Sphinx reconstruction.

Figure 2.6: Screenshots of the original Photosynth application based on the research on Photo Tourism. Image courtesy of Microsoft Research [78]

The second generation is not based on SfM, but on image-stitching technologies. It allows the user to create and explore panoramas, even of gigapixel size, by stitching multiple images together. As well as “Synths” these panoramas are created offline and can then be published on any website or social media page, where they can be viewed with the help of a Silverlight-based web application.

The third and latest generation of Photosynth is a completely revised version, which is based on the Spin project of the Interactive Visual Media group of Microsoft Research [79]. The newest version is now built around a cloud computing platform called Microsoft Azure [66]. This technology hosts a service, that allows the creation and exploration of immersive 3D representations of photo collections. In contrast to the first generation, this version allows more realistic transitions between the photographs. While the first generation and the approach realized with the “Photo Tourism” paper uses simple planar proxies for in-between view creation, the newest Photosynth version computes more complex proxy geometries out of piecewise planar depth maps. These depth maps are constructed for each input photo independently by solving a pixel labelling problem, where a salient scene plane, extracted from a sparse 3D reconstruction, is assigned to each pixel of an image. When interpolating between two views, each photo is projectively texture-mapped onto its corresponding proxy geometry in different off-screen render passes. The resulting images are blended together, where regions with contributions from both images are linearly cross-faded. More details about the algorithm introduced by Sinha et al. can be found in [91]. However, this approach is just suitable for opaque surfaces and cannot handle transitions in scenes where reflections or transmissions at specular surfaces are present. Therefore, Sinha et al. introduced an enhancement of the above described algorithm in their paper *Image-Based Rendering for Scenes with Reflections* [90].



(a) 3D proxy geometry texture-mapped with its corresponding photo.

(b) Annotations become visible when the user hovers the mouse over a displayed circle, which is placed nearby an important scene object.

Figure 2.7: Screenshots of the third generation of Photosynth application. Image courtesy of Microsoft Research [78]

The current application makes it possible to generate four different “Synths”: spin, panorama, walk, and wall. Spin is created by taking pictures of an object while moving around it. Panoramas are stitched from photos captured from a single location looking in every direction. Walks are, as implied by the word, generated by following a path while taking photographs. At last, walls are created by shooting pictures when sliding across a scene. When navigating through the scene, the images are consequently mapped onto the non-planar proxy geometries, as depicted in Figure 2.7a. Additionally, the application allows the integration of annotations, as can be seen in Figure 2.7b.

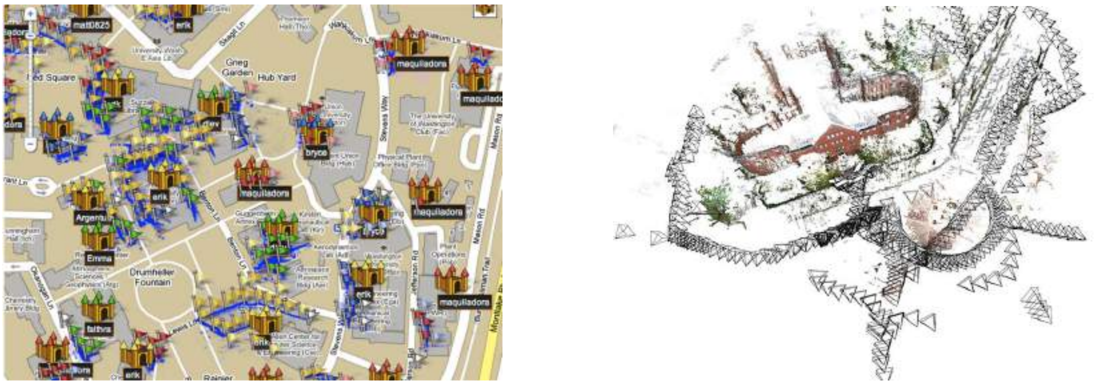
Photosynth uses a system called *Seadragon* to stream image data. This system is able to detect which part of an image at which resolution is currently viewed and sends just the required data over the network. This makes it possible to view and explore such large photo collections online in an interactive way in 3D.

2.2.3 PhotoCity

Another interesting project, related to our work, was realized by Tuite et al. at the University of Washington. Their corresponding paper *PhotoCity: Training Experts at Large-scale Image Acquisition Through a Competitive Game* was published in 2011 [105].

During the development of the *PhotoCity* platform they targeted the same problem as our attempt, namely that just landmarks and important sights are photographed often enough to be reconstructed as 3D point clouds. There exist not enough photos to recreate less popular regions, like sideways or buildings with no meaning to the broad mass of people.

PhotoCity is a game whose idea is to take as many photographs as possible to consequently enhance the depicted object's virtual 3D reconstruction. A player can extend existing 3D buildings or spawn new ones, called seeds. Seeds are dense scene reconstructions generated out of 20 to 200 input images. The players can earn or capture flags, castles or gems by following this game principle. Castles are captured by the player who has mostly contributed to the corresponding 3D reconstruction. Flags are placed automatically alongside walls of a building and get captured by players whose uploaded photos lead to the most points nearby them. While castles and flags can change ownership during game-play, gems are collectables that disappear after earned. They are similarly collected as flags, but are not automatically placed by the system, but by administrators to animate the players to reconstruct 3D structure at the gem's locations. Figure 2.8a shows a map of the campus of the University of Washington together with the flags and castles coloured with the colour of the current owner or annotated with his user name. In Figure 2.8b the positions are shown from where photos are taken to reconstruct the nearby building.



(a) PhotoCity map representing the campus of the University of Washington and the located castles and flags.

(b) A reconstructed building together with the camera view positions visualized as black triangles.

Figure 2.8: PhotoCity map and 3D reconstruction example. Image courtesy of Tuite et al. [105]

The game was playable during a field study for six weeks between March and May 2010. This study consists of a competition between forty-five players, each students either at the University of Washington or the Cornell University. During this competition, the players submitted over 109,000 photos, from which 68,000 photos were registered and used for scene reconstruction.

This results in the fact that 60% of the uploaded photos are used in the 3D buildings, while reconstructions done by using images from photo-sharing sites like Flickr are usually based on significantly less photos (cf. [1]). Tuite et al. summarized that unorganized photo collections can never reach such a coverage of target objects as it was achieved by the participants of their field study. Therefore, photos from online photo-sharing platforms are mostly not suitable for precise 3D reconstruction, which explains why models computed out of large community photo collections are based on a relatively small number of images compared to the large number of input photographs.

CHAPTER 3

The reconstruction pipeline

The aforementioned reconstruction and photo exploration applications are based on photogrammetry algorithms called *structure-from-motion (SfM)* and *multi-view stereo (MVS)*. SfM is, as its name implies, a method for estimating 3D structure and camera motion out of a series of 2D images. While SfM only reconstructs a sparse scene structure, MVS is a technique used to refine this sparse structure in order to receive a dense scene reconstruction. This chapter describes both techniques and how they can be used as parts of a classic reconstruction pipeline in more detail. Furthermore, an overview about some alternative approaches in contrast to the traditional reconstruction algorithm is given.

3.1 The classic reconstruction algorithm

As already mentioned in Chapter 1, triangulation is used in SfM algorithms to compute 3D points (cf. Figure 1.3). To do triangulation, the extrinsic and intrinsic camera parameters that make up its *projection matrix* are needed. These parameters can either be computed by calibrating the cameras before doing reconstruction or retrieved on the fly during the reconstruction process with SfM [46].

3.1.1 Camera calibration

Camera calibration is the process of determining the extrinsic and intrinsic parameters of an image sensor and its lens, in order to create its *camera matrix* or *projection matrix* P, which is used to compute the projected position x of a 3D point X on an image plane. While the extrinsic parameters describe the camera's position and orientation in the world, the intrinsic values stand for its internal characteristics like the focal length of the lens, its skew, optical centre and distortion. The camera matrix can be described by Equation 3.1, which is derived from a basic *pinhole camera model* (cf. Figure 3.1) [50].

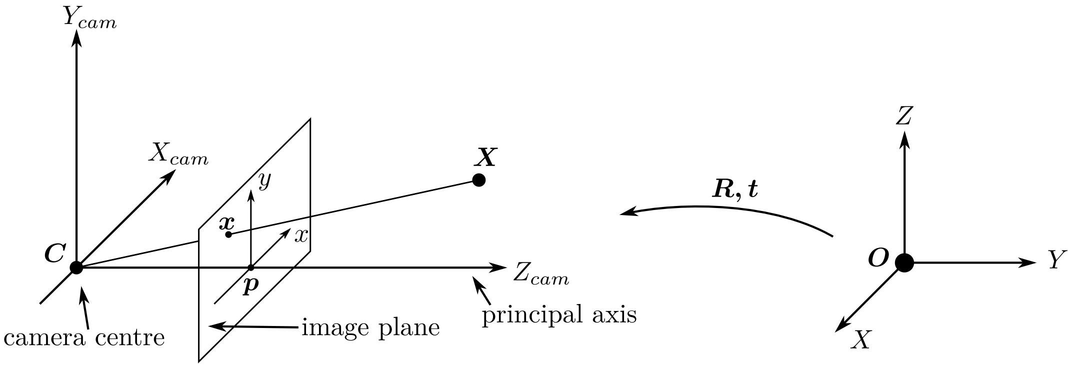
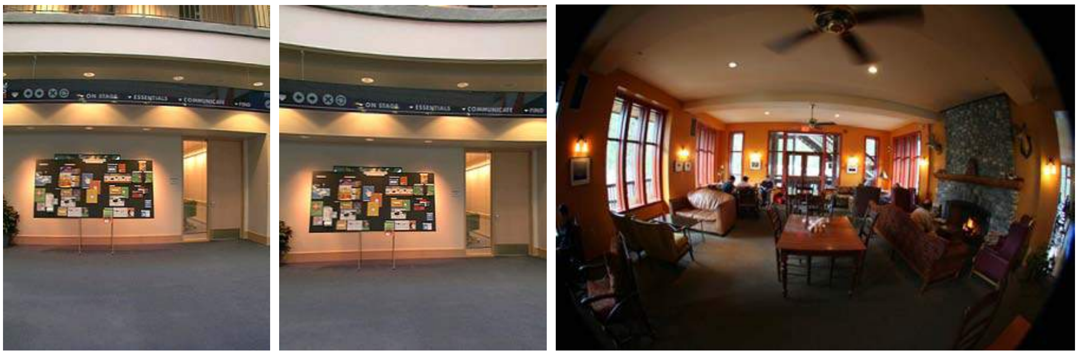


Figure 3.1: Pinhole camera model: The left part depicts the geometry of a pinhole camera, where C is the camera centre or optical centre. The line from C perpendicular to the image plane is called *principal axis* and the point where image plane and this axis intersect is known as *principal point* p. The right side shows the Euclidean transformation, consisting of a rotation R and translation t, between the world and camera coordinate system. This figure is redrawn as seen in [50].

$$x = PX = K[R|t]X = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3.1}$$

It consists of the intrinsic matrix K and the extrinsic matrix $[R|t]$, containing the rotation R and translation t. x_0 and y_0 are the coordinates of the principle point (cf. Figure 3.1). The parameters α_x and α_y describe pixel scale factors in x- and y-direction, which are not equal if the pixels on a camera sensor are non-square. These parameters are defined using the focal length f and the number of pixel per unit distance m_x and m_y in x- and y-direction. s defines the skew between the sensor axes. In addition to these intrinsics derived from the camera matrix, there exist other internal parameters that are not considered by the simple pinhole camera model and that describe optical distortion effects. These cannot be computed with the help of linear matrix calculations, since distortions are non-linear transformations. The most common type of distortion is radial distortion, which occurs when points are displaced away (barrel) or towards (pincushion) the image center. This results in straight lines having a curvature in their projection (cf. Figure 3.2). Another type of distortion is tangential distortion, which occurs when the image sensor isn't completely parallel to the lens. The different types of distortion can be eliminated by correction algorithms, like the ones introduced in [22][23][93].

After defining the transformation between 3D points and their projections, the question is how to find the unknown variables. There exists a lot of literature on how to answer this question. One of the most influential methods to compute these parameters was introduced by Zhang in 2000 [122]. The technique presented in this paper is based on



(a) Barrel (b) Pincushion (c) Fisheye

Figure 3.2: Different examples of radial lens distortion. Image courtesy of Szeliski [102]

a planar pattern, like a checkerboard, that is photographed from several positions and orientations. The locations of distinctive features on the pattern, like the corners of each chessboard square, must be known. Next the coordinates of these features are extracted from each taken photo by means of pattern recognition algorithms, like the Harris corner detector [47]. After that a homography H (cf. Equation 3.2) is computed, that describes the relationship between the known feature locations of the planar pattern and their coordinates extracted from the photographs (without loss of generality $Z = 0$ is assumed):

$$x = K \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = K \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = H \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \tag{3.2}$$

$$B = (K^{-1})^T K^{-1} \tag{3.3}$$

It can be seen, that the camera matrix P appears in this homography. Finally, H can be computed by using a maximum likelihood approach. The homography can then be used as a constraint to compute the matrix B (cf. Equation 3.3), from which the intrinsic parameters, and therefore K can be retrieved. The extrinsic values per view can be calculated by using the inverse of K and the corresponding homography. As of now the extrinsic parameters and the intrinsic camera matrix have been recovered. Distortion coefficients have not been computed yet, however, Zhang shows a way to compute the coefficients for radial distortion by using an algorithm based on least squares. More complex distortions are not covered in the paper. For more details about Zhang's algorithm see his publication [122].

3.1.2 Auto-calibration and projective reconstruction using SfM

For most of the reconstruction use-cases it is not possible to pre-compute a camera calibration and therefore a camera's projection matrix, because the used camera lenses for specific images are unknown or other parameters used for the calibration step are not recoverable. However, there exists an approach that does not need a pre-computed camera calibration, but detects the camera pose and intrinsics simultaneously with the 3D point reconstruction by only using image feature correspondences. This approach is called structure-from-motion (SfM). Since the sparse reconstruction recovered by this approach is unique up to a projective transformation, it is called projective reconstruction. The process of determining the camera parameters during the computation of a projective reconstruction is known as auto-calibration.

Figure 3.3 depicts the stages of a reconstruction pipeline, which all images have to pass in order to finally get a high quality, dense 3D reconstruction represented as a point cloud. The system developed during the work on this thesis is based on this reconstruction pipeline. The following subsections give a more detailed view of the individual parts of the pipeline concerning the SfM part. The final step that produces the dense reconstruction uses an algorithm called multi-view stereo (MVS). This step is explained in more detail in Section 3.1.3.

Feature detection

The first step of the pipeline is the feature detection step, where distinctive image interest points are recovered, that can be subsequently used in the next step to find relations between different images. There exist various feature detectors in literature.

One of the most prominent detectors is the *SIFT* algorithm proposed by Brown and Lowe [14][62]. SIFT is an abbreviation for scale-invariant feature transform, which implies, that the introduced algorithm delivers feature descriptors that are not affected by image transformations like rotation or scaling. This means that the features of two images can be compared with the help of their descriptors even if one image doesn't have the same rotation or scaling as the other. This is advantageous for typical reconstruction use-cases, since often images are compared that are not even shot with the same camera. SIFT finds features placed at the minima or maxima of a Difference-of-Gaussians (DoG) function that are present at multiple scales, which explains the scale-independence of the descriptor. For each feature a description vector containing 128 elements is computed. The elements of this descriptor represent a 4x4 array of orientation histograms, each of them having 8 bins ($4 \times 4 \times 8 = 128$), that contain orientation and magnitude values of samples in a 16×16 region around the feature point (also known as key point). The descriptor is rotation independent because the orientations are given relative to the feature point's dominant gradient direction. A GPU-based SIFT implementation called SiftGPU is used by the SfM software VisualSFM, which we incorporate into our virtual tourism system (cf. Chapter 6) [119][120].

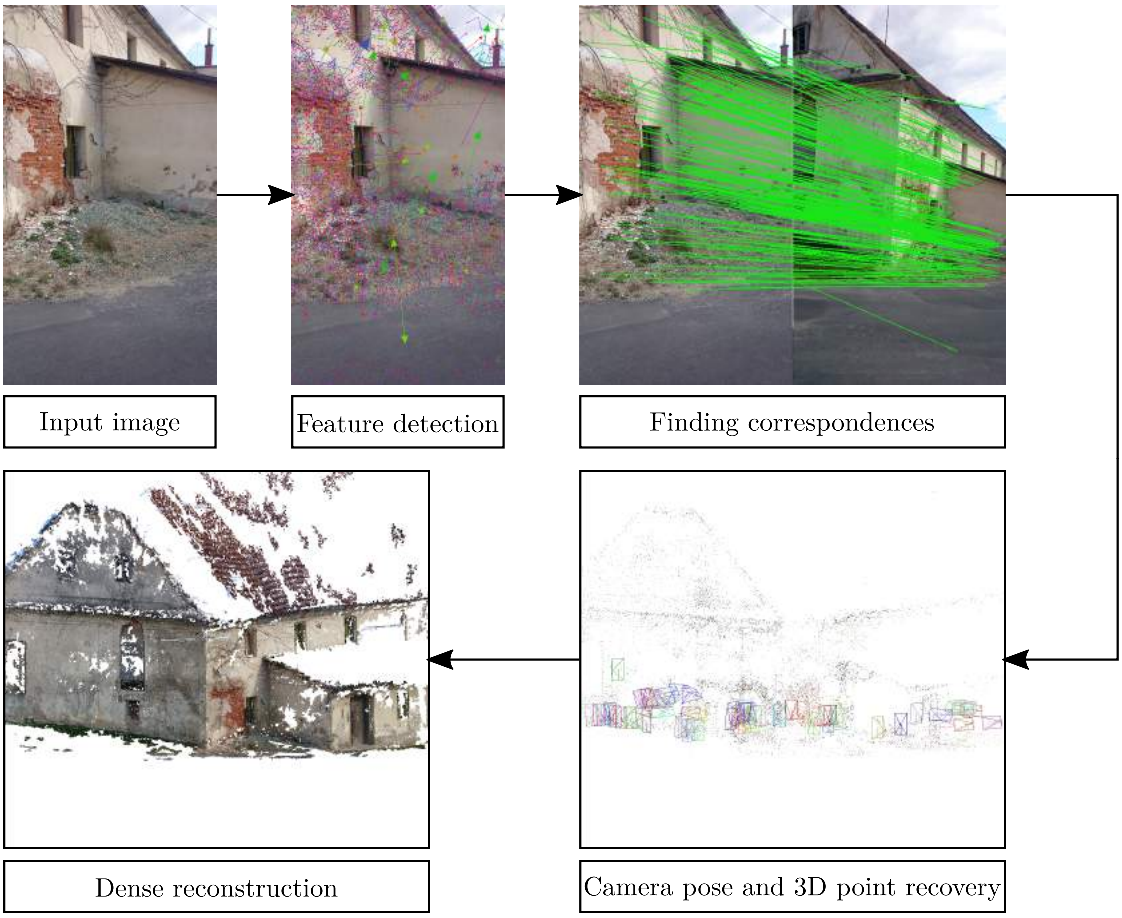


Figure 3.3: The classic reconstruction pipeline: Each image passes through the stages feature detection, feature matching, camera pose detection (motion), sparse reconstruction (structure) and at last dense reconstruction using multi-view stereo (MVS).

Another often used and widely accepted feature detector is *SURF (Speeded Up Robust Features)* introduced by Bay et al. in 2006 [8]. In contrast to SIFT, which uses the Difference-of-Gaussians (DoG) function for feature extraction, the algorithm behind SURF detects interest points with the help of a Hessian matrix together with integral images. The descriptor is based on a distribution of Haar wavelet responses of values in the neighbourhood of the key point. Compared to SIFT, this descriptor contains only 64 values, which should make comparison operations faster. According to the authors, SURF is faster and in certain situations more precise than SIFT. However, different comparisons show that on the whole SIFT delivers the more accurate results and mostly more features, while SURF comes with less computation time. On the other hand, there exist implementations of SIFT that improve the runtime by using a smaller descriptor. For time-sensitive applications, SURF or SIFT with a smaller feature descriptor should be preferred. If time is not that important, the classic SIFT method delivers the most accurate features [7][56].

A more recently proposed feature detection algorithm is called ORB (Oriented FAST and Rotated BRIEF) by Rublee et al. [83]. It combines the feature detector FAST (cf. [82]) with the feature descriptor BRIEF (cf. [16]). This combination results in a rotation invariant interest point detector, that is according to the authors one magnitude faster than SURF and even two magnitudes faster than SIFT. The drawback of ORB is, that scale invariance is not adequately observed and therefore cannot be guaranteed, which makes SIFT or SURF a better alternative when differently scaled images are going to be processed.

Finding correspondences, motion and structure

After identifying the features, they can be used to find features correspondences between the images, which are then used to recover the relative camera positions and orientations (*motion*) and the locations of the 3D points (*structure*). The following paragraphs explain how to obtain these correspondences and compute the camera parameters and 3D reconstruction.

Relations between the features in image pairs can be found by using a k-nearest neighbour classification in feature space. Brown and Lowe suggest a k-d tree based implementation to optimize the run-time [14]. Snavely et al. use the approximate nearest neighbour algorithm by Arya et al. for feature matching [6][97]. The pair-wise matches are optimized using a *RANSAC* (Random Sample Consensus) loop, where in each iteration a so called *fundamental matrix* is calculated describing the relation between the two views. The original matches are then verified by using the recovered matrix. Outliers which are according to the matrix not geometrically consistent matches are removed. Besides a brute-force method, that pair-wise matches all input images and therefore has $O(n^2)$ complexity (cf. O-notation by Knuth [57]), in literature more enhanced implementations, that are faster and have a better scaling behaviour regarding the number of input images are proposed. Agarwal et al. use an approach commonly found in document-comparison algorithms to find pairs of images that are similar to each other [1]. They assign “visual words” to each SIFT feature, such that every image is described by a set of words. This information can be used by document-retrieval algorithms to find similar images for further feature matching computations. When using photos from internet photo collections, it is often the case that most of the photographs represent nearly the same scene. Snavely et al. proposed an approach that filters the input images by removing as many images as possible without exceeding a bounded lose of accuracy and completeness in the final reconstruction. The filtered image collection is called a skeletal set and is based on the maximum leaf spanning tree of a match graph, where the input images represent nodes that are connected by edges if the corresponding images have features in common. The reconstruction revealed with the help of this skeletal set is an approximation to the solution that would have been found with the full set of input images. However, the runtime of the SfM algorithm can be reduced by this approach, since the camera poses and 3D points are just calculated from a subset of input images [98].

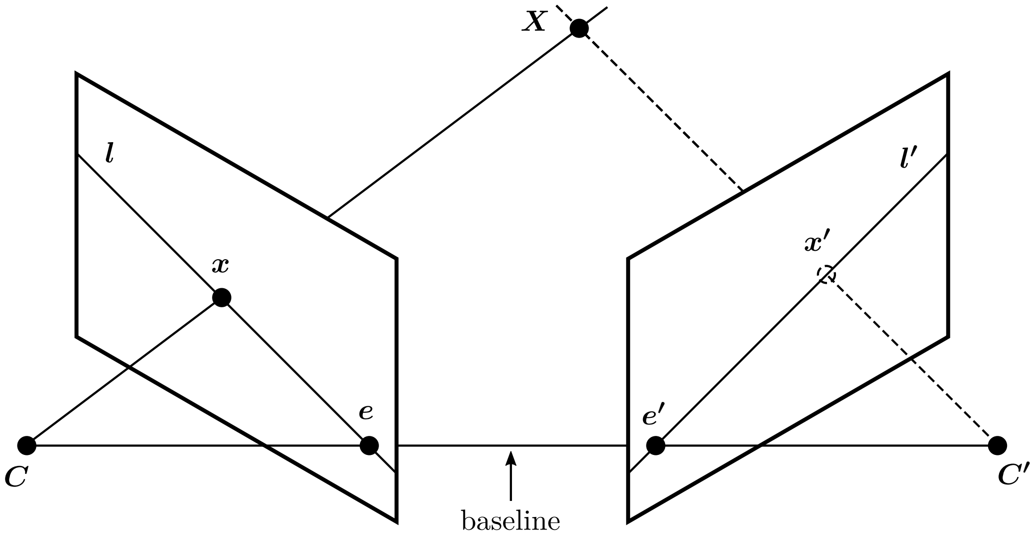


Figure 3.4: Epipolar geometry: The epipoles e and e' are defined by the intersections between the camera baseline and the image planes. A projection x of a 3D point X has its corresponding projection x' restricted to the epipolar line l' going through its epipole e'. l and l' span up an epipolar plane containing the 3D point X. This figure is inspired by an illustration seen in [81].

The above mentioned fundamental matrix F was nearly simultaneously introduced by Faugeras et al. [27][28] and Hartley et al. [49][51] in 1992. It is a 3×3 matrix describing the epipolar geometry of two camera views, as can be seen in Figure 3.4. A projection x of a 3D point X in one image has its corresponding projection x' in the other image on the so called epipolar line l'. The epipolar geometry and therefore the fundamental matrix is independent of scene structure and depends just on the camera's intrinsics and their relative position and orientation. If a projection x has its corresponding projection x' in the other view, then the following condition, known as epipolar constraint, is satisfied [50]:

$$x'^T F x = 0 \tag{3.4}$$

This condition allows the computation of F only out of point correspondences between the two images using the so-called *eight-point algorithm*. It was originally developed by Longuet-Higgins in 1981, to compute the *essential matrix* out of at least eight image point correspondences [61]. The essential matrix is a specialized version of the fundamental matrix describing relations between calibrated cameras, while the fundamental matrix also covers uncalibrated cases. However, Hartley showed that an optimized version of this algorithm, that uses normalized image point coordinates, is able to recover the fundamental matrix too [48]. The algorithm mainly consists of two parts. The first part is the formulation and solving of a set of homogeneous equations. Such a system of equations can lead to a unique result or a least-squares solution. The second step enforces the internal constraint of a fundamental matrix, namely that it is singular, which means that the condition $\det F = 0$ is fulfilled. Hartley and Zisserman proofed that it is even possible to compute the fundamental matrix out of only seven correspondences by making use of this singularity constraint [50].

Having determined F, it is subsequently used to recover the camera's extrinsic and intrinsic parameters, and therefore their projection matrices P and P' by means of triangulation. Figure 3.5 depicts the idea behind triangulation in the context of SfM. It can be seen that the image points x and x' are back-projected along their viewing rays. The reconstructed point X fulfils the epipolar constraint $x'^T F x = 0$, when it is placed on the epipolar plane, spanned by the epipolar lines of the two corresponding views, at the intersection point of the two rays [50].

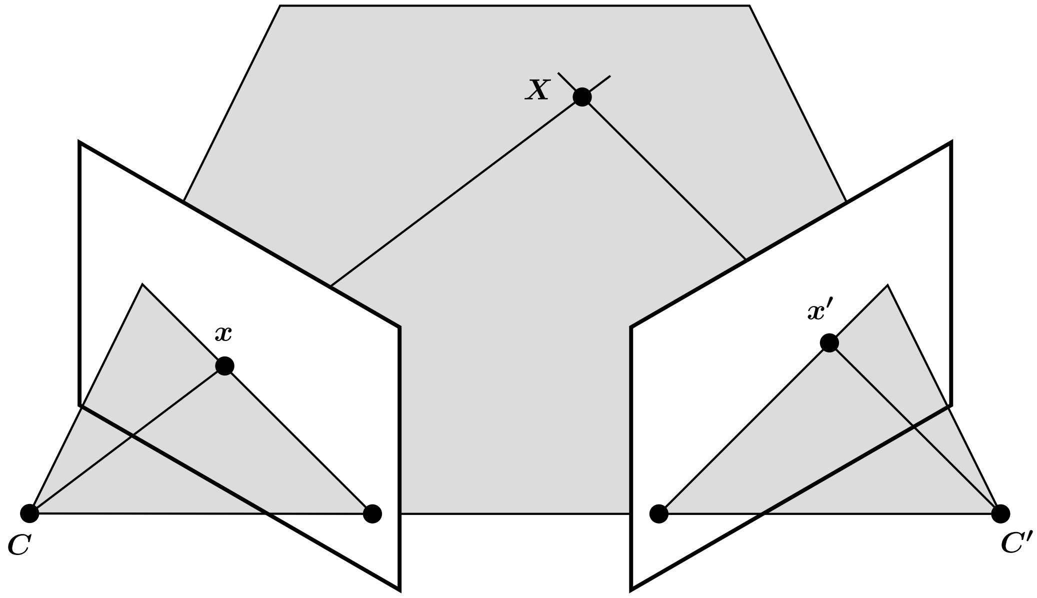


Figure 3.5: Triangulation: The reconstructed point X lies on the epipolar plane, spanned by the epipolar lines of the two corresponding views, at the intersection of the two viewing rays, originating at their camera's center point and going through the image points x and x'. This figure is redrawn as seen in [50].

The points of a scene can be reconstructed up to a projective ambiguity when using the fundamental matrix. There exist two possible solutions to remove this ambiguity and upgrade the reconstruction to a metric reconstruction that preserves length ratios and angles between lines and is therefore similar to the real world, however the scale is still indeterminable. One solution is to use the essential matrix instead of the fundamental matrix to reconstruct the 3D points. The reconstruction is then unique up to a scale factor and a four-fold ambiguity, which means that there are eventually four possible solutions. However, there exists just one plausible solution where a reconstructed point X lies in front of both cameras, which eventually makes the resulting projection matrices unique up to an arbitrary scale factor. As already mentioned above, the essential matrix can only be retrieved if the camera's intrinsic parameters are known. For uncalibrated cases still a projective reconstruction must be computed with the help of the fundamental matrix. This reconstruction has to be refined to a metric by using five or more ground control points with known Euclidean coordinates X_{Ei}. With the help of these ground control points a homography H should be computed, such that $X_{Ei} = HX_i$. H can then be used to compute the metric reconstruction $\{P_M, P'_M, X_{Mi}\}$, consisting of the metric projection matrices P_M and P'_M and the reconstructed 3D points X_{Mi}, according to [50] with

$$P_M = PH^{-1} \quad , \quad P'_M = P'H^{-1} \quad , \quad X_{Mi} = HX_i \tag{3.5}$$

Another way to convert the given reconstruction to a metric one is called stratified reconstruction. Here, firstly an affine reconstruction is computed and at last it is converted into a metric reconstruction. This method makes usage of different constraints extracted from, for example, parallel or orthogonal lines in the scene. The process of computing a metric reconstruction either directly or via an affine reconstruction out of a projective reconstruction is called *rectification* [50].

Usually the process of computing structure and motion out of input images follows an incremental approach. At first the cameras and 3D points are calculated using an initial pair of input images. Subsequently one image after another is added to the existing reconstruction, until all photos are processed. After each incrementation an operation called *bundle adjustment* is performed. This time-critical algorithm minimizes the reprojection error across all input images. Snavely et al. use the Levenberg-Marquardt (LM) algorithm to solve this non-linear least squares problem [97][118]. Due to the problem's complexity, a lot of papers aim to improve that part of the reconstruction pipeline in order to speed up the reconstruction process. Byröd and Åström suggest to use iterative approaches like conjugate gradients, instead of the LM algorithm with its cubic complexity, to solve the problem [15]. Wu et al. introduced a multicore bundle adjuster implementation that runs on CPUs and GPUs up to 10 to 30 times faster than conventional single core implementations, like the one by Agarwal et al. [2].

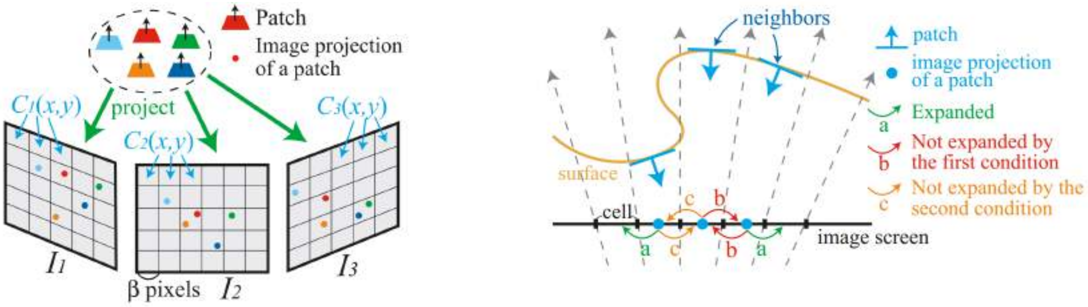
3.1.3 Dense reconstruction using MVS

Before this last stage of the reconstruction pipeline introduced in Section 3.1.2 an estimation of the camera's intrinsic and extrinsic parameters and a sparse scene reconstruction are computed. For some applications, like image-based modelling and rendering, a more accurate reconstruction is desirable. *Multi-view stereo (MVS)* approaches are able to produce a dense scene reconstruction out of stereo correspondences and camera parameters. Since these properties are a result of the previous pipeline stages that make up the SfM algorithm, all prerequisites for the final dense reconstruction step are given.

According to Furukawa and Ponce the different MVS approaches can be classified into four categories, according to the generated scene's representation [34]:

- Voxels
- Polygonal meshes
- Multiple depth maps
- Patches

Voxel-based approaches generate a scene's geometry on a regularly sampled 3D grid known as volume. The accuracy of such methods is given by the sampling density of the volume. Techniques based on polygonal meshes represent the surface of a scene with the help of connected facets (flat faces). Other methods define the scene by using multiple depth maps, one for each input image. These maps are finally fused into a single 3D representation. At last patch-based techniques combine multiple patches or surfels into a 3D scene. These patches can be represented by points and the resulting scene is then defined by a point cloud, which can be rendered with the help of point-based rendering techniques. The following paragraphs give an overview about patch-based algorithms that are even suitable for large outdoor scenes, because our work mainly focuses on reconstructions of tourism sights or outdoor architecture scenes that are rendered as point clouds at the client application (cf. Chapter 7) [34].


Patch
Image projection of a patch
$C_1(x,y)$
project
$C_2(x,y)$
$C_3(x,y)$
I_1
β pixels
I_2
I_3
neighbors
patch
image projection of a patch
a
Expanded
b
Not expanded by the first condition
c
Not expanded by the second condition
surface
cell
c
b
a
c
b
a
image screen

(a) Patch projection into image cells $C_i(x, y)$: The goal of the algorithm is that each image cell contains at least one patch projection.

(b) Patch expansion: Patches are expanded if neighbouring image cells are empty and not expanded if there already exists a projection of a neighbouring patch in the cell or if they are not visible from the camera's viewpoint.

Figure 3.6: Concepts of the patch-based dense reconstruction algorithm by Furukawa and Ponce. Image courtesy of Furukawa and Ponce [34].

Furukawa and Ponce introduced a *patch-based multi-view stereo (PMVS)* algorithm that is able to reconstruct large scenes in an accurate and complete manner [34]. They propose a three-step algorithm, that firstly generates a patch-based scene reconstruction which is subsequently converted into a polygonal mesh and at last refined by a mesh-based MVS algorithm. Since we don't handle polygonal meshes in this thesis, we focus only on the first part of the published algorithm, that is also used by the VisualSFM software, which is integrated into our web application (cf. Chapter 6). The patch-based algorithm consists of the steps feature matching, patch expansion and patch filtering. The initial feature matching step is required to generate a sparse set of patches. The second part aims to expand the sparse set of patches such that each image cell $C_i(x, y)$ contains at least one projected patch (cf. Figure 3.6a). Figure 3.6b depicts cases where patches can be expanded and cases where it is not necessary or possible to expand them. The last step removes faulty patches, so-called outliers, that are not consistent with the overall visibility information. For example, if two patches p and p' are not neighbours but projected to the same cell $C_i(x, y)$ of an image C_i, then either p or p' is filtered out as an outlier, according to a given inequality. The expansion and filtering steps are repeated n times. The authors suggest to iterate at least 3 times to get a dense result with a minimum number of outliers or inaccuracies [34]. The algorithm by Furukawa and Ponce was reused in their large scale approach *Clustering Views for Multi-view Stereo (CMVS)* [33]. This method, as the name implies, clusters the input images into sets of overlapping photographs and runs the reconstruction process for every cluster in parallel. At last the single reconstructions are merged with the help of a parallelized out-of-core merging algorithm. The authors showed that this technique is able to reconstruct large point clouds with nearly thirty million points out of more than ten thousand input images in reasonable time [33].

Vu et al. proposed a different approach in 2012 [109]. In contrast to the previously presented work that uses patch expansion, they work with sparse depth maps that are generated between pairs of input images. These depth maps are merged and clusters of points are extracted according to their positions in the camera's view frusta. After that the clusters are split until their projected bounding boxes are small enough. With the help of a 3D k-D tree the k-nearest neighbouring clusters are identified. A plane is then created at each point's neighbourhood using a least squares fitting algorithm. If the plane fits the neighbourhood well (according to a thresholded matching score), the point is retained and iteratively refined. In a second step the computed quasi dense point cloud is triangulated using Delaunay triangulation. The extracted Delaunay tetrahedra are labelled as inside or outside the object and at last a surface is generated that consists of triangles that connect the inside and outside tetrahedra.

Goesele et al. produce multiple depth maps with their multi-view stereo algorithm [37]. As the algorithm by Furukawa and Ponce, their algorithm aims to reconstruct scenes out of large community photo collections. They achieve that in multiple stages. First, the distortions of the input images are removed and the camera's extrinsic and intrinsic parameters are computed using a robust metric SfM algorithm. Then, for each input image a depth map is derived. Each input image is used once as a reference view for stereo matching. To find appropriate neighbourhood images as candidates for stereo matching, the authors proposed a two-level view selection scheme. First, a global view selection at image level identifies candidate images. These images are further filtered using local view selection at pixel level. This final filtering step identifies a subset of images for each pixel that produce a stable stereo match. Having identified appropriate image subsets, a multi-view stereo algorithm based on a surface growing approach is used to create the depth maps for each reference image. The surface growing algorithm uses the points of the previously computed sparse reconstruction as input. Finally, the generated depth maps can be merged and converted to a surface mesh with the help of several different techniques. Goesele et al. use Poisson surface reconstruction in their work [55].

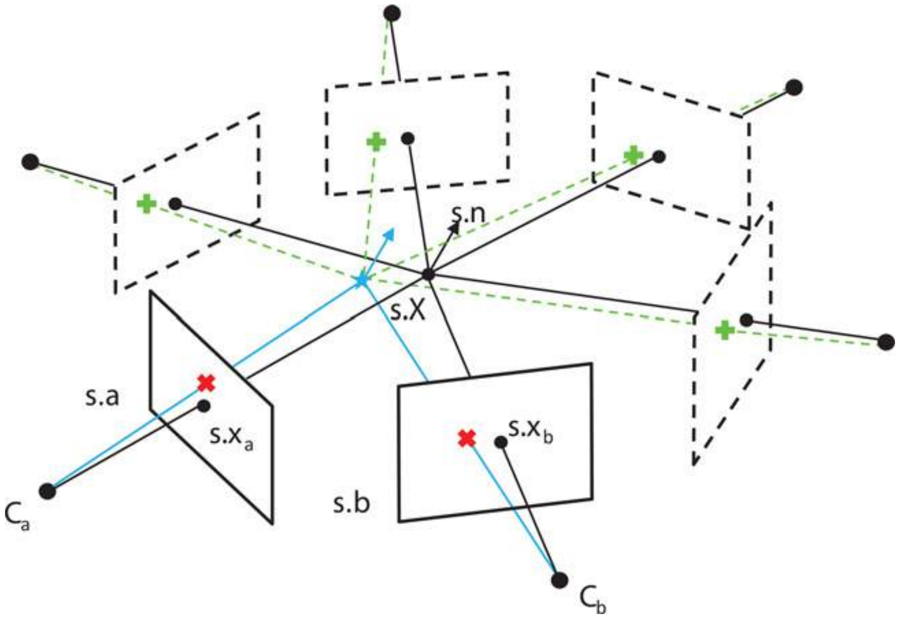


Figure 3.7: The expansion process of the patch-based reconstruction algorithm with prioritised patch expansion. Red crosses symbolize new correspondence pairs, the blue star indicates the newly reconstructed 3D point and the green plus signs are the projections of this point. See the text for more details. Image courtesy of Ylimäki et al. [121].

A more recent approach by Ylimäki et al. is based on patch expansion and is similar to the approach by Furukawa and Ponce described above [121]. Their contribution is that they use a so-called prioritised patch expansion algorithm that outperforms non-prioritised approaches, like the PMVS method, in accuracy, completeness and time. The previously matched features and corresponding sparsely reconstructed 3D points called “seeds” are ordered into a priority queue according to their similarity. This queue is processed sequentially, such that the best placed seed always comes first. Each of the seeds in the queue is expanded. This expansion process is depicted in Figure 3.7, where s describes the currently processed seed. To expand it, the method searches for new correspondence pairs in the neighbourhood of the current seed in the corresponding reference views. A new 3D point is reconstructed by means of triangulation. At last the newly created point is projected into all other images and the matches are evaluated. If a specified quality threshold is exceeded, the new point is added to the reconstruction and the priority queue, so that it can be processed by a later iteration. This expansion process continues until the queue is empty. After that a new process can be started with initial seeds placed nearby holes or at outer boundaries of the reconstruction to increase its accuracy and completeness. Compared to the PMVS algorithm, this approach lacks an additional filtering step, which is, according to the authors, one reason, that their algorithm exposes an improved computational efficiency. Furthermore, the best-seeds-first tactics guarantees the good quality and accuracy of the reconstruction.

3.2 Alternative approaches

Besides the classic incremental approach described in the previous section, there exist other attempts to solve the 3D reconstruction and camera pose estimation problem. One of those are batch or global approaches where, unlike the incremental algorithm, all camera estimates are calculated at once. The advantage of global methods compared to sequential or incremental methods is, that expensive intermediate bundle adjustment and outlier removal algorithms are not needed, because all parameters are computed simultaneously and therefore no error propagation can occur due to a sequentially growing reconstruction. This fact allows a more efficient computation, as shown by Sinha et al. in their proposed version of a global SfM algorithm that performs up to two orders of magnitude faster than incremental SfM [92]. They firstly extract vanishing points (VPs) and feature point matches that are used to recover all camera rotations. After that the camera positions and 3D points are all estimated at once using a linear reconstruction approach. Finally bundle adjustment is applied to the completely reconstructed model and all cameras.

Hierarchical SfM techniques are mostly out-of-core algorithms that produce several partial scene reconstructions and finally merge them together into one overall solution. The drawback of such algorithms is that they still need an intermediate bundle adjustment step to reduce error propagation, like the classic incremental approach. However, they are largely parallelizable and usually perform better than sequential SfM, as shown by Gherardi et al. [35]. Their technique organizes the input images in a tree hierarchy, called dendrogram. At the reconstruction stage, the tree hierarchy is followed starting at the leaves, where a two-view reconstruction similar to the one performed by an incremental approach, is computed. Moving towards the root at each node either an additional image is added to the reconstruction or two reconstructions are merged. To improve the computational efficiency compared to incremental SfM, the used tree must be balanced, otherwise the benefit of the hierarchical method vanishes. To enforce the tree balancing, the views are ordered by the number of common keypoints and the distribution of those points over the image. Then agglomerative clustering is used to generate the dendrogram. Based on a balanced dendrogram, the complexity of the SfM algorithm can be improved from $O(n^5)$ to $O(n^4)$ (cf. O-notation by Knuth [57]), where n denotes the number of input images [35].

Another different approach to the traditional reconstruction pipeline are so called SLAM (Simultaneous Localization and Mapping) algorithms that are used to solve a similar problem as SfM, but in real-time. That is the reason why SLAM is often called the real-time SfM. While SfM algorithms usually recover camera parameters and a 3D reconstruction from a large unordered set of input images, SLAM is used mainly in robotics to track and reconstruct the movement and view of a single camera (monocular) or a pre-calibrated stereo camera rig. Since SLAM approaches focus on robot or camera localization and real-time map creation, they don't use expensive error minimization approaches like bundle adjustment in order to reconstruct accurate 3D scenes. They only produce sparse or at most semi-dense scenes or maps. However, with ongoing hardware improvements and increasing computational possibilities the results of SLAM and SfM approaches converge. Engel et al. introduced an interesting approach in 2014 known as Large-Scale Direct Monocular SLAM or *LSD-SLAM* [24]. While SfM and SLAM algorithms are usually based on feature detection and matching, their featureless approach directly works with image intensities. The camera is tracked with the help of image-to-image alignment, where the camera's pose is estimated with respect to a current keyframe by using the previous frame as initialization. The map is reconstructed in form of depth maps by filtering over a lot of pixel comparisons and matches [24].

3.3 Applications

In addition to the usage in tourism applications as seen in Chapter 2, nowadays SfM algorithms can be used successfully to reconstruct whole cities from very large photo collections, as shown by Agarwal et al. in their publication "Building Rome in a day" [1]. They presented a system which is able to reconstruct the city of Rome out of 150K photographs within a day using a cluster of 496 compute cores and a novel parallel SfM algorithm.

Another interesting application field of SfM is interactive modelling of urban scenes. Reitinger et al. proposed an augmented reality application based on SfM, where a "scout", equipped with a mobile PC, a USB camera and a GPS receiver, explores the city and sends a series of georeferenced photographs to a remote server. The server computes a 3D reconstruction, stores it in a database and sends it back to the scout's mobile device, where he can explore the 3D registered reconstruction [77].

Due to the increasing accuracy and completeness of their reconstructions, SfM approaches have become a low-cost and affordable option beside high-cost laser scanning installations in the field of geoscience and cultural heritage. There exist several papers, comparing the results of laser scanning and SfM in the mentioned application fields [4][111].

As already stated in Section 3.2 so called SLAM methods are used in robotics for robot localization and environment reconstruction. Nuchter et al., for example, use their SLAM technique for creating a 3D reconstruction of an abandoned mine by an autonomous robot [71]. Mars rover research also experiments with SLAM based algorithms, like FastSLAM [68], for their autonomous robots.

CHAPTER 4

Working with large point clouds

Structure-from-motion algorithms are used to retrieve camera poses and 3D reconstructions out of a number of input images, as already mentioned in Chapter 3. The resulting reconstructions can be stored and visualized using different data structures, like meshes, depth maps and point clouds. Since our work is based on point clouds and our front- and back-end implementation, including the database, must handle even large outdoor models, this chapter gives an overview about methods of working with such large point-based scene representations.

4.1 Basic data structures

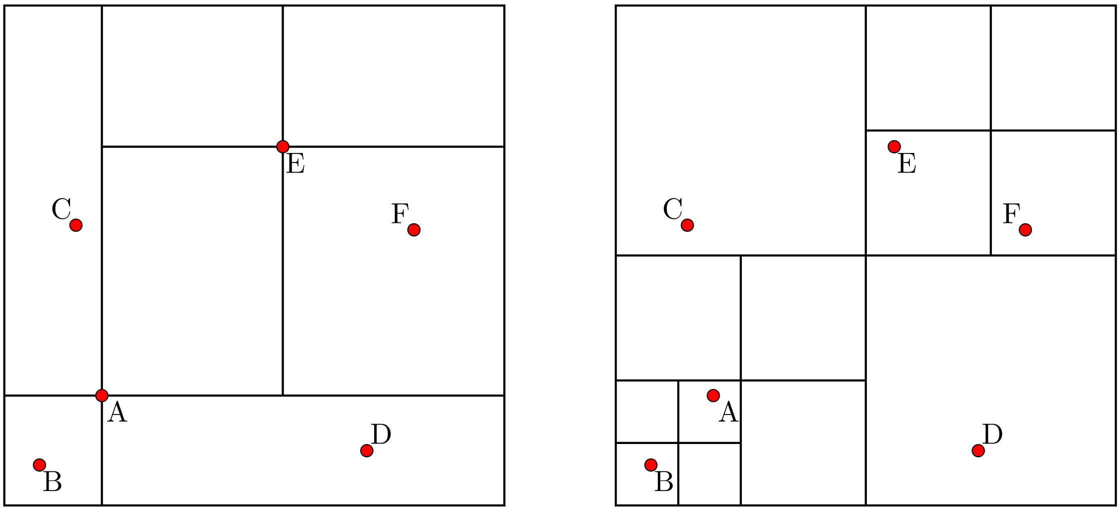
Nowadays structure-from-motion and multi-view stereo algorithms can output point clouds consisting of millions of points, as can be seen in [1]. Laser scanners are even able to capture billions of points. In order to deal with such large amounts of point data several data structures exist (cf. overview by Samet [85]). This section describes some fundamental representations on which the more enhanced data structures and rendering methods, like the one discussed in Section 4.2, are based on.

4.1.1 Quadtree / Octree

A *quadtree*, originally proposed by Finkel and Bentley in 1974, is a two-dimensional hierarchical data structure that is well suited for fast point searching [29]. A quadtree consists of nodes, which are generally axis-aligned quads. The space occupied by a node is subdivided into four non-empty subspaces, that contain further nodes (that can be again subdivided), or empty subspaces. The subdivision splits each node's space along axis-parallel lines. Samet's overview contains different types of quadtrees used for point storage [85]. They generally differ in how the space division is done and where the points are stored.

A point quadtree subdivides the space at lines that intersect in a data point. It can be seen as a simple two-dimensional binary search tree, where the firstly inserted point serves as the root node. Subsequently inserted points are placed at the quadrant derived from the point's metrics (for example its location in a two-dimensional space). Data points are stored in inner nodes as well as in leaf nodes. Searching and insertion time complexity is $O(log(n)$ (cf. O-notation by Knuth [57]), where n denotes the number of currently stored points. An example of a point quadtree is depicted in Figure 4.1a [85].

Quadtrees where spaces are not subdivided at data points are called trie-based quadtrees. Such quadtrees split a node's space along lines that go through the center of a node, or at arbitrary positions within the node's space. In the context of point data with indiscrete properties (like floating-point coordinates) a trie-based quadtree is called point region quadtree (*PR quadtree*). While a point quadtree stores data points at each node, in a PR quadtree points are only saved in leaf nodes, as can be seen in Figure 4.1b. The time complexity of search and insertion operations depends on the depth of the PR quadtree. In contrast to the two described variations of quadtrees, which contain only at most one point per node, bucketed versions of this data structure are able to store more than one point per node [85].



(a) Point quadtree: The node spaces are subdivided at data points. Each node can contain data points.

(b) PR quadtree: The node spaces are subdivided at their centres. Only leaf nodes contain data points.

Figure 4.1: Quadtree variants for point storage. The figures are inspired by illustrations seen in [85].

The quadtree data structure can be easily extended into three-dimensional space. The extended 3D data structure is called *octree* and was introduced by Meagher in 1980 [65]. In contrast to a quadtree, each node can hold at most eight instead of four children. The above described variations in space division and point storage can be similarly applied to octrees [85].

4.1.2 B-Tree

The *B-Tree* invented by Bayer and McCreight in 1972 is a self-balancing tree that allows any number of child nodes at any node [9]. Therefore it can be seen as a generalization of the classic binary search tree, which allows only two child nodes per parent node. Since it is self-balancing, each leaf node is at the same tree level. Like a point region tree, the B-Tree stores data points only at leaf nodes, while the internal nodes are just used for the derivation of a search path, when looking for specific data points.

The B-Tree is primarily used as data structure for secondary, external data storage, because it is optimized for storing and reading large data blocks. Due to its self-balancing property, the height of a B-Tree is kept low, which further leads to a reduction of slow disk accesses when searching for a specific dataset. Therefore, it is commonly used in databases and file systems, like Microsoft's NTFS and Apple's HFS+ [11]. Other variants like B^*-Trees and B^+-Trees are developed to further improve the performance of a B-Tree as a data structure for external memory. The B^*-Tree optimizes his fill-rate by sharing keys between neighbouring interior nodes. This leads to a more dense and lower tree, where non-root nodes are at least $2/3$ full instead of just $1/2$, which further increases the storage utilization and makes search operations faster, when compared to the conventional B-Tree. The B^+-Tree speeds up search operations by directly linking leaf nodes for faster sequential access. Search, insertion and deletion operations have only logarithmic time complexity $O(log(n))$, where n denotes the number of stored records [18]. Figure 4.2 shows a B-Tree of order 5, where the order describes the maximum number of children of a non-leaf node [85].

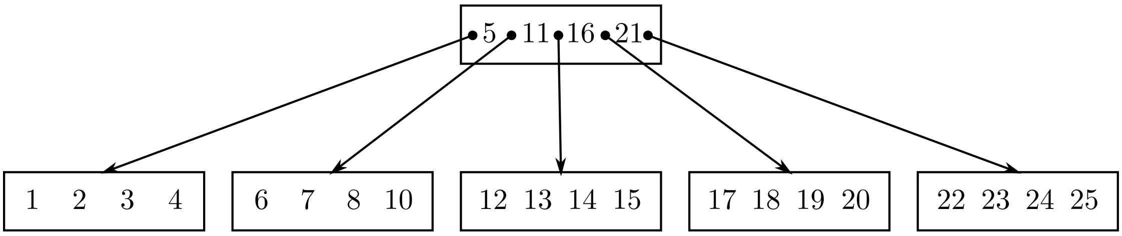
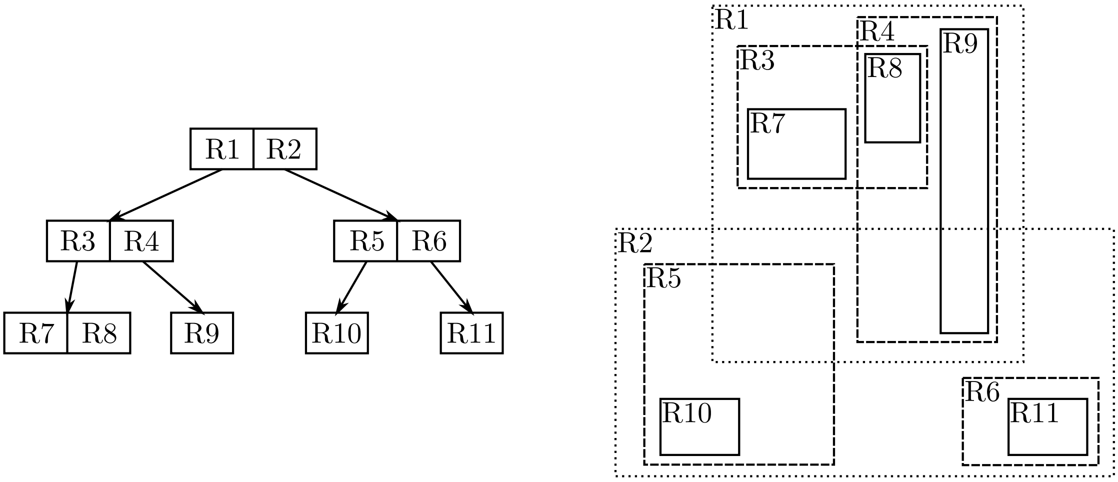


Figure 4.2: A B-Tree of order 5, where a non-leaf node has at most 5 children. The keys stored in the nodes are used for searching specific datasets, which are linked in the leaf nodes. This figure is redrawn as seen in [9].

4.1.3 R-Tree

The *R-Tree* extends the idea of the B-Tree to multidimensional spaces [43]. An R-Tree divides the point data into n-dimensional regions and builds up a region hierarchy. The regions are defined by the minimum bounding rectangles (MBRs) of the data points they contain. Each parent knows just the bounding boxes of its children. The data points itself are stored in the leaf nodes. Like the B-Tree, the R-Tree is a balanced tree, which means that each leaf node is at the same tree level. Searching for a specific point is done by using the MBRs. This approach makes it possible to ignore whole subtrees, if the searched point doesn't lie within their bounding boxes. That leads to the fact, that just a fraction of nodes are visited and therefore needed during a search operation. This nodes can be paged to memory on demand, which makes a *R*-Tree suitable for large amounts of data that cannot be loaded into memory at once, like the point data stored in spatial databases. Figure 4.2 depicts an example of a *R*-Tree.



(a) *R*-Tree

(b) The bounding boxes of the nodes stored in the *R*-Tree depicted in (a)

Figure 4.3: *R*-Tree together with the bounding boxes of its nodes. These figures are simplified redrawn as seen in [43].

It can be seen that there are many overlapping regions, which minimize the search performance of an *R*-Tree in cases where a searched point lies in the overlapping part of two regions. In such cases both subtrees need to be expanded and queried. An R^*-Tree reduces the overlap of regions by using split and reinsertion algorithms. The R^+-Tree goes one step further and prohibits the overlap of two search regions. The drawback of this constraint is, that objects that belong to more than one region must be stored in each region separately (e.g. a polygon that extends to several search regions). However, in the case of points this drawback is negligible and R^+-Trees boost the search speed because all search regions are just covered by at most one search node, which makes unnecessary subtree expansions impossible [85].

4.2 Large point cloud rendering

Finding a suitable data structure for point datasets is an inevitable task when it comes to rendering the point clouds. Especially when dealing with large datasets that exceed the graphics memory, optimized data structures are needed to ensure an acceptable real-time performance. The previous section already introduced some basic representations which provide the basis for the rendering techniques described in the following paragraphs.

4.2.1 QSplat

Rusinkiewicz and Levoy proposed a data structure called *QSplat* for the storage and rendering of large point clouds [84]. They organize point clouds in a bounding-sphere hierarchy which is preprocessed and stored on the disk. The preprocessing algorithm creates this tree-like hierarchy by splitting the input point cloud along its longest bounding box axis. This operation retrieves the two bounding spheres of the tree root's children. This splitting operation is repeated until a node's bounding sphere contains just a single point. The points are stored in the leaf nodes of the bounding-sphere hierarchy. Internal nodes store the averaged values of their children, which makes it possible to render the point cloud in different LODs. The hierarchy is stored by traversing it in breadth-first order starting at the root node, which means that the levels of the tree and therefore the different LODs are stored sequentially in the file on the disk. Each LOD or tree level contains the whole point cloud in a more or less dense representation. This fact allows the rendering of a low-resolution point cloud by only reading the first part of the file, that stores the QSplat. A heuristic based on the projected point size on the screen decides which LOD is used when rendering the point cloud. Since large point datasets usually exceed the internal memory, the levels are loaded on demand from the disk. To further improve rendering performance, a visibility check allows to skip the loading of whole subhierarchies of the QSplat data structure, if their bounding spheres are outside the camera's view frustum.

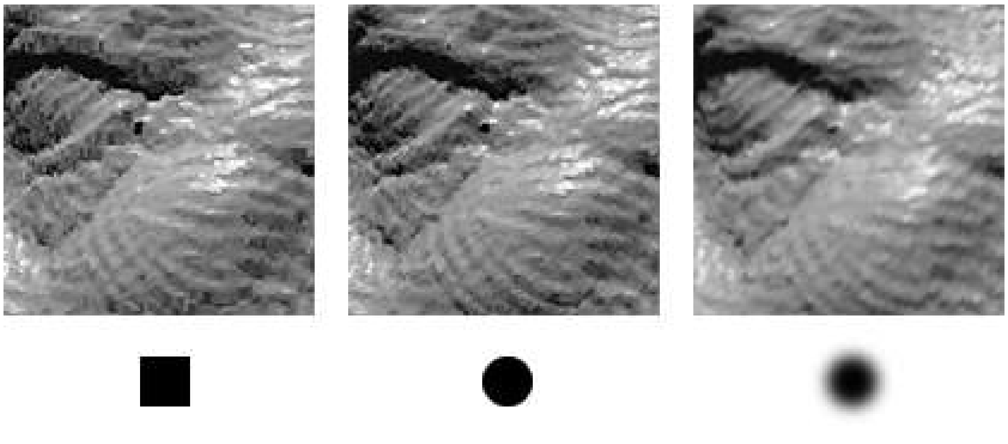


Figure 4.4: A scene represented as a QSplat and rendered by using a square, circle or Gaussian splat kernels. Image courtesy of Rusinkiewicz and Levoy [84].

Figure 4.4 depicts rendering results of the QSplat algorithm using different splat kernels. It can be seen that the Gaussian kernel delivers the best result with less aliasing compared to the other renderings. However, according to the authors it takes four times longer to render a scene with a Gaussian kernel than with a square kernel. Another problem of the QSplat approach is, that each leaf node stores just one single point and rendering is therefore done point-by-point, which is not very efficient when using modern GPU-based rendering techniques, since they perform better when transferring larger blocks of data to the GPU.

4.2.2 Layered Point Clouds

Another rendering and storage method for large point clouds is the approach by Gobbetti and Marton called *layered point clouds (LPC)* [36]. In contrast to the QSplat algorithm, LPC stores up to M points in each node of the tree-like hierarchy, making the data structure better optimized for GPU-based rendering. As QSplat, LPC uses a binary tree hierarchy that splits each node at its longest bounding box axis. Each node stores uniformly subsampled points of the original input cloud. After storing points in the data structure, these are removed from the input cloud. The remaining point cloud is further split along its longest bounding box axis and the two resulting chunks are subsampled to generate the child nodes of the previously created parent. This process is repeated until a node contains $\leq M$ points. Figure 4.5 shows the LPC data structure created from the input point cloud. It can be seen that each child node locally increases the density of the point cloud stored in its parent node.

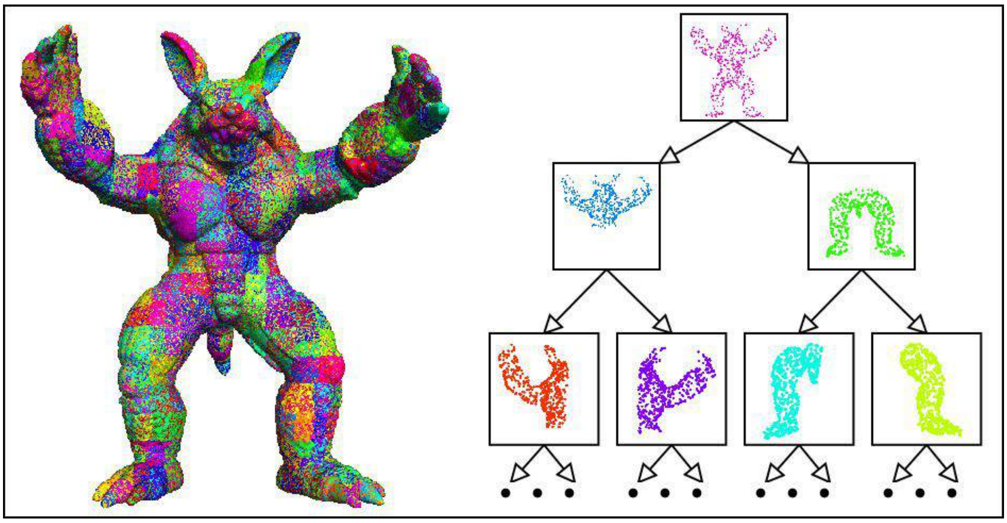


Figure 4.5: LPC data structure: The input point cloud is split into equally sized chunks. These chunks are stored out-of-core in a repository on a local or remote filesystem and are loaded on demand during the rendering process. They are further used to build up a level-of-detail hierarchy, where each child node locally refines the point cloud stored in its parent node. Image courtesy of Gobbetti and Marton [36].

The point cloud chunks are stored in compressed form in a point cloud repository either on a local or remote filesystem, which is accessed via NFS mounts or HTTP connections. The repository is structured in a way, such that the storage order is the same as the traversal order, when rendering the point cloud. This order is breadth-first, which means that the point cloud rendering process renders a coarse level of detail, which is refined by loading further child nodes. The needed nodes are requested from the repository if they are not currently available in the graphics card memory. Additionally, an index tree, representing the parent-child relationships of the data structure, stores unique links that identify the several point clouds saved in the repository. When rendering the cloud, the index hierarchy is traversed down until a specific level of detail is reached. The required level of detail is computed for a current view point by using the maximum distance between the projected visible points.

4.2.3 Instant Points

Instant Points proposed by Wimmer and Scheiblauer in 2006 is another out-of-core algorithm used to render large point clouds [117]. Like the algorithms described in the previous sections, the instant points method also builds up a hierarchy in order to allow out-of-core rendering in several levels-of-detail. The data structure created by this algorithm is an octree-based representation named *nested octree*.

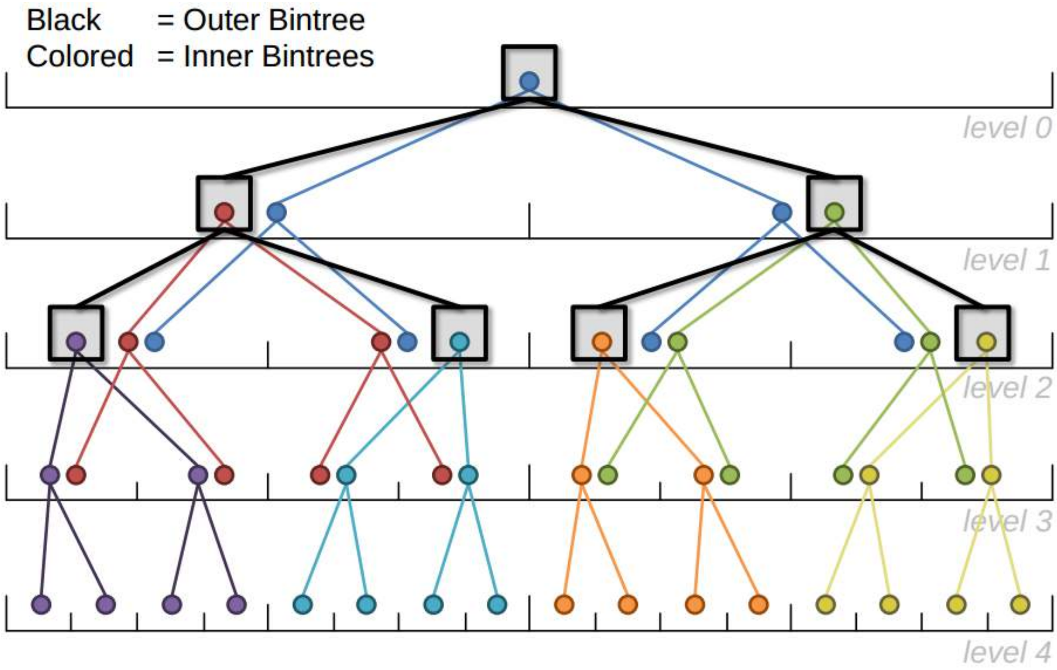


Figure 4.6: The nested octree data structure, as basis of the instant points rendering algorithm, simplified illustrated in 1D as a nested bintree. The bintree is of order 5, while the outer and all its inner bintrees have order 3. The points are stored in internal and leaf nodes. Image courtesy of Scheiblauer [86].

Wimmer and Scheiblauer distinguish between inner and outer octrees in their data structure. The outer octree specifies the traversal order for rendering and the inner octrees, which are bound to a maximum depth, actually store the points. Each node of the outer octree has a corresponding inner octree and each node of an inner octree can store at most one point. The bounding boxes of the inner octree's root nodes are the same as the bounding boxes of the outer octree nodes they belong to. Figure 4.6 depicts a one-dimensional nested octree, which is in fact a nested bintree. The inner bintrees are represented by different colours and the outer bintree is visualized with bold black edges and square nodes. It can be seen, that the bintree is full, because each of its levels contain points [86].

The nested octree is precomputed by iterating through the points of the original input point cloud. The octree is build up from top to bottom, first inserting into the inner octree of the outer octree's root node. If all nodes of the inner octree are occupied, the subsequently inserted points are saved in inner octree's inscribed at the appropriate child node of the outer octree's root. Generally, the recursive insertion algorithm iterates through all nodes of the outer octree until it finds an inner octree that is not fully occupied and offers a suitable free node for a currently processed point. A main novelty of the instant points rendering algorithm is the LOD creation by using the inner octrees. When the camera is far away from the scene, such that it is sufficient to render just the root node of the outer octree, the rendering algorithm can use the levels-of-detail of the inner octree to further reduce the number of rendered points. However, when the camera comes closer to the scene more outer octree nodes are needed for a satisfying result. In such cases more points than needed are rendered, as can be seen in Figure 4.6, where each of the bins in level 2-4 contain points from different inner octree nodes. Because ultimately all points of an inner octree, corresponding to a requested outer octree node, are rendered, it can happen that more than one point per bin is drawn, although it would be sufficient to render just one point per bin.

When rendering the point cloud the outer octree is loaded completely into memory and traversed in order to create a priority queue using the size of the node's projected bounding box as priority value. The queue is then iterated and nodes are filtered out according to several conditions. Outer octree nodes are only considered for rendering, if the projected bounding boxes of the lowest nodes stored in the corresponding inner octrees are larger than a predefined threshold, which is usually 1 pixel. Candidate nodes are view frustum culled before rendering, removing nodes that are not visible from a current view point. Furthermore, nodes that are not in graphics memory are requested from an external storage and skipped in the rendering process of the current frame. Instant points uses a budget-based rendering approach, which means that a user-defined maximum number of points must be considered when rendering the point cloud. If the points stored in the inner octree of a candidate node would exceed this maximum number, the node is skipped and the iteration of the priority queue is stopped. Inner octrees collected during this filtering process are finally stored in a render queue, which is actually traversed during rendering. When a node of an outer octree is rendered, all points of the inscribed inner octree are drawn [86][117].

To optimize the rendering process, points that are not rendered any more are inserted into a least-recently-used cache (LRU) in the main memory and deleted from the graphics card's memory. If a point is needed for rendering subsequent frames, they are loaded from the external memory, only if they are not available in the LRU-cache [86][117].

4.2.4 Potree

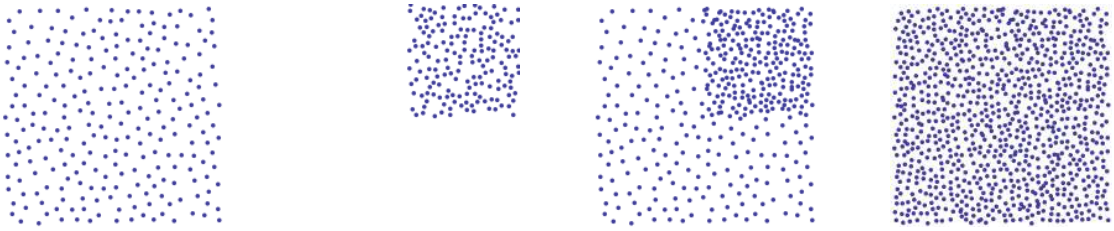
The web-based point cloud renderer *Potree*, which we use in our front-end implementation (see Section 6.2), is based on a simplified version of the nested octree data structure used by the instant points rendering algorithm [87]. In detail, instead of differentiating between outer and inner octrees, Potree uses just one octree, which stores points in all nodes: root, internal and leaf nodes [88].

In contrast to the nested octree data structure described in Section 4.2.3, that stores all points of the original input point cloud, the points stored in the original Potree data structure are uniformly selected with the help of a user-defined point spacing. This spacing value defines the minimum distance between two points in the root node. For each level below the root level this distance is halved, which leads to a more detailed representation of the point cloud in those levels, because they contain more points [88]. However, since Potree version 1.3, all points of the input point cloud are stored. The minimum distance check is only applied for non-leaf nodes. Points in leaf nodes are not affected by this constraint and contain all points that are rejected by the internal nodes [87].

Again levels-of-detail are introduced by the data structure behind the Potree renderer. The octree's root stores a sparse version of the original input point cloud. Combining the points of the root with points from subsequent child nodes, makes the resulting point cloud more dense until it reaches it's highest LOD when all octree levels are rendered. Figure 4.7 illustrates the described principle [87].

During the rendering process the octree nodes are loaded on demand. Each of the nodes is stored in a separate file in the file system. Furthermore, the octree hierarchy is split into hierarchy chunks, each of them described in a hierarchy file. The description of a hierarchy contains a node and a predefined number of its descendants. For each of the nodes, the number of stored points is saved. All hierarchy files together describe the octree hierarchy of the whole point cloud.

Before loading the point data stored in the node files, the hierarchy file of the first chunk is loaded and processed in order to decide which nodes are actually rendered. For this purpose Potree uses a priority queue to sort the nodes according to their screen projected size. This is done for the child nodes of each parent node separately to decide the processing order of its children in subsequent iterations. Nodes (and their children) that are not inside the current view frustum are rejected. Furthermore, nodes with a screen projected size below a predefined threshold are discarded. Potree also uses a



(a) Points stored in the root node.

(b) Points stored in the first child of the root node.

(c) Points of the root and its first child combined.

(d) Points of the root and all of its children combined.

Figure 4.7: An illustration of the LOD representation as implemented in the octree-based data structure used by the Potree renderer. While the root node contains only a sparse version of the stored point cloud, combining the points from all octree nodes increases the level-of-detail and leads to a more dense point cloud. Image courtesy of Schütz [87].

point budget, to bound the number of rendered points to a maximum value in order to guarantee real-time frame rates. This means that the tree-traversal is stopped when a node exceeds the budget or when there are no further nodes to process. Nodes that are considered visible are requested from the server, unless they are already loaded into memory. When a node has a hierarchy file attached (same file name, but other file extension), the hierarchy chunk contained in this file is also downloaded from the server and processed as described above. Since it is not possible to store an infinite number of points in memory, not needed points are removed by using a least-recently-used cache (LRU). When a specified number of points are in memory, points from the LRU-cache are removed before new ones are requested from the server.

Since we adapted the octree generation and loading algorithm for our application, more details about it are covered in Section 6.3.

CHAPTER 5

Geographic coordinate systems

The visual tourism client realized during the work on this thesis is based on point clouds that are georeferenced and placed on a globe. Therefore, our implementation uses geographic coordinate systems to describe point positions and transformations. To be able to follow the details presented in the implementation chapter (see Chapter 6), this chapter gives a short overview of the used coordinate systems and the map algebra behind them.

5.1 World Geodetic System 1984 (WGS84)

The World Geodetic System (WGS) is a standard used in geodesy, cartography and other earth sciences for defining positions on the Earth. Its latest version named WGS84 originated in 1984 and was last updated in 2004. The standard defines a coordinate system for the Earth, a reference ellipsoid approximating the Earth's surface and a geoid derived from a gravitational model, that defines the nominal or mean sea level. The global positioning system (GPS) is one of the important applications that are based on the WGS84 standard [53].

5.1.1 Reference coordinate system

The coordinate system defined by the WGS84 standard is a Cartesian coordinate system having its origin at the Earth's center of mass. Figure 5.1 depicts the coordinate system defined by its origin and its three orthogonal axes. The z-axis is derived from the reference pole defined by the International Earth Rotation Service (IERS) and coincides with the Earth's rotation axis [64]. The x-axis lies on the equatorial plane, that passes through the origin, is orthogonal to the z-axis and intersects the IERS reference meridian at 0° longitude. This meridian, also known as prime meridian, passes approximately 102 metres east of the Royal Observatory in Greenwich [63]. At last the y-axis, computed as

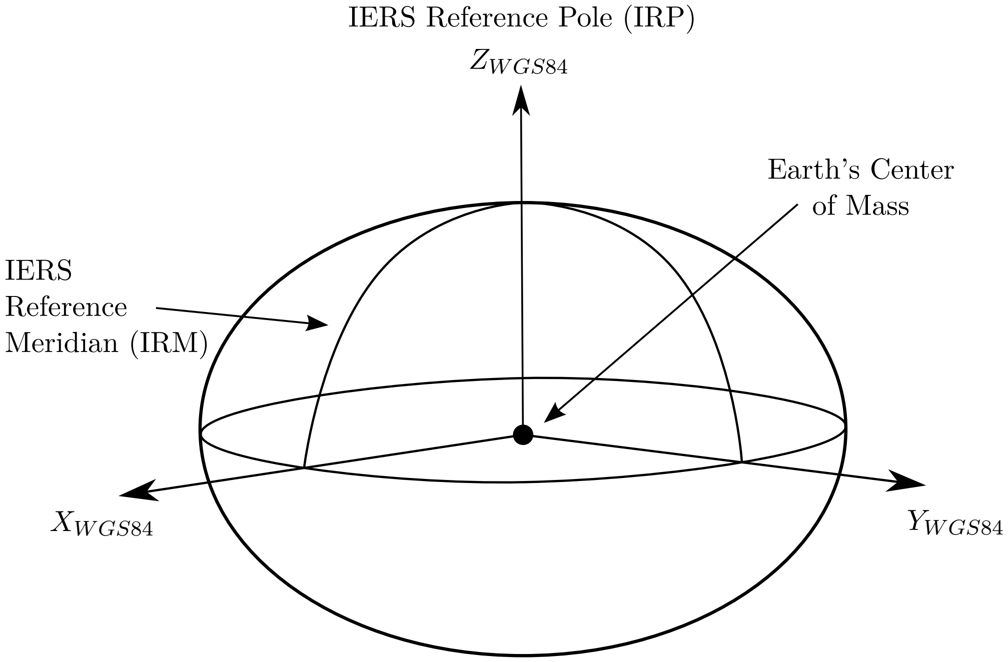


Figure 5.1: Definition of the WGS84 coordinate system. This figure is redrawn as seen in [53].

the cross product of the other two axes, completes the right-handed and earth-centered orthogonal frame. Additionally, the WGS84 frame is an earth-fixed coordinate system, which means that its axes are fixed with respect to the Earth's surface. The Earth doesn't rotate about the z-axis of the coordinate system, but the coordinate system rotates with the Earth. Because of these two properties the WGS84 coordinate system is a so called *Earth-centered, Earth-fixed (ECEF)* coordinate system or a *Conventional Terrestrial System* [58].

5.1.2 Reference ellipsoid

The WGS84 *reference ellipsoid* or WGS84 *datum* is a geocentric ellipsoid of revolution that is defined by four parameters [53]:

- Semi-major axis (equatorial radius) $a = 6378137.0m$
- Reciprocal of flattening $1/f = 298.257223563$
- Angular velocity of the Earth $\omega = 7292115.0 \times 10^{-11} rad/s$
- Earth's gravitational constant $GM = 3986004.418 \times 10^{8} m^3/s^2$

The semi-major axis and the flattening are used to describe the ellipsoid geometrically, whereas the other two are physical parameters that are used by some geodetic applications, for example, to approximate the Earth's gravitational field.

When it comes to coordinate conversion tasks, as described in Section 5.3, another important geometric parameter of the ellipsoid is needed. The ellipsoid's semi-minor axis or polar radius b can be derived from a and $1/f$ as following [53]:

$$\frac{b}{a} = 1 - f \qquad \rightarrow \qquad b = a(1 - f) \approx 6356752.314245m \tag{5.1}$$

There exist several other geometric and physical parameters that can be computed with the help of the four defining parameters mentioned above. However, for our purpose these parameters are not needed and therefore not discussed in more detail. For further information about the WGS84 reference ellipsoid see the technical report containing its detailed definition [53].

Geodetic coordinates

Besides the usage as an Earth's surface approximation, the reference ellipsoid is also used to define a so-called geodetic coordinate system. Such a coordinate system describes curvilinear coordinates known as geodetic *latitude* and geodetic *longitude*. The geodetic latitude ϕ is the angle between the equatorial plane and the normal to the reference ellipsoid at a given point. The geodetic longitude λ is the angle between the plane of the prime meridian and the plane of a given meridian. Longitudes are usually measured positively from the prime meridian eastwards and negatively westwards. Latitudes are usually measured positive from the equator northwards and negative southwards [76]. The altitude or height as the third components of a geodetic coordinate is usually given as the distance above the reference ellipsoid. GPS is an example, that uses height values above the WGS84 reference ellipsoid [32]. However, there exist some applications that define the height above mean sea level (MSL), which can differ significantly from the height above the ellipsoid as depicted in Figure 5.2. Figure 5.3 shows the relationship between the different coordinate systems used by our system.

5.1.3 Geoid

The third part completing the WGS84 standard is the definition of a *geoid* based on an earth gravitational model. The geoid represents the shape of the Earth's ocean surfaces considering the influences of the Earth's rotation and gravitation and ignoring disturbances like winds, tides, waves, etc. The mean sea level can be identified by the geoid. Compared to the reference ellipsoid, which is a mathematically idealized representation of the Earth's surface, the geoid is a more irregular surface that is, however, still smoother than the Earth's real topographic surface. Figure 5.2 depicts the relationship between the three different surfaces [76]. It illustrates that the difference N between the geoid and the reference ellipsoid can be significant. Therefore it is important to distinguish between the height above the ellipsoid and the height above the geoid, which is known as the height above mean sea level (MSL). Our application is based on height values relative to the reference ellipsoid.

To compute the mean sea level and therefore the geoid several earth gravitational models exist. The latest released version that describes a gravitational model relative to the WGS84 ellipsoid is the Earth Gravitational Model 2008 (EGM2008). The EGM consists of a number of so called spherical harmonic coefficients that are used to mathematically approximate the geoid.

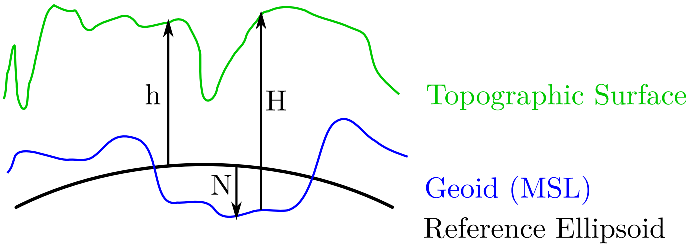


Figure 5.2: The relationship between the reference ellipsoid, the geoid defined by the gravitational model and the real topographic terrain. Here h denotes the height above the reference ellipsoid, H the height above mean sea level and N the difference between the ellipsoid and the geoid. This figure is inspired by an illustration seen in [32].

5.2 Local ENU reference frame

For transformations, like user modifications to correct georeferencing errors in our application, a local ground reference frame is more intuitive and convenient than the global ECEF or geodetic coordinate system. Therefore we introduce a local Cartesian coordinate system called *East-North-Up (ENU)*, that is defined by a tangent plane to the reference ellipsoid at a specific location. The axis pointing eastwards is labelled with x, the northward axis y and the normal to the tangent plane defines the z-axis of this local coordinate system, that is shown coloured in green in Figure 5.3 [110].

5.3 Coordinate conversions

The previous sections defined different coordinate systems that we deal with in our application. To optimize the rendering process and simplify matrix calculations, it is often necessary to switch between different frames, depending on the computation task. Furthermore, several services like the Bing Location Service we use in our front-end implementation, deliver results in geodetic coordinates. However, to use such coordinates during the rendering process with WebGL or OpenGL, they have to be converted to Cartesian coordinates. This section describes algorithms we used to convert coordinates between geodetic and Cartesian frames. Geodetic longitude is labelled as λ, geodetic latitude as ϕ, and height above the WGS84 ellipsoid as h. Cartesian coordinates are

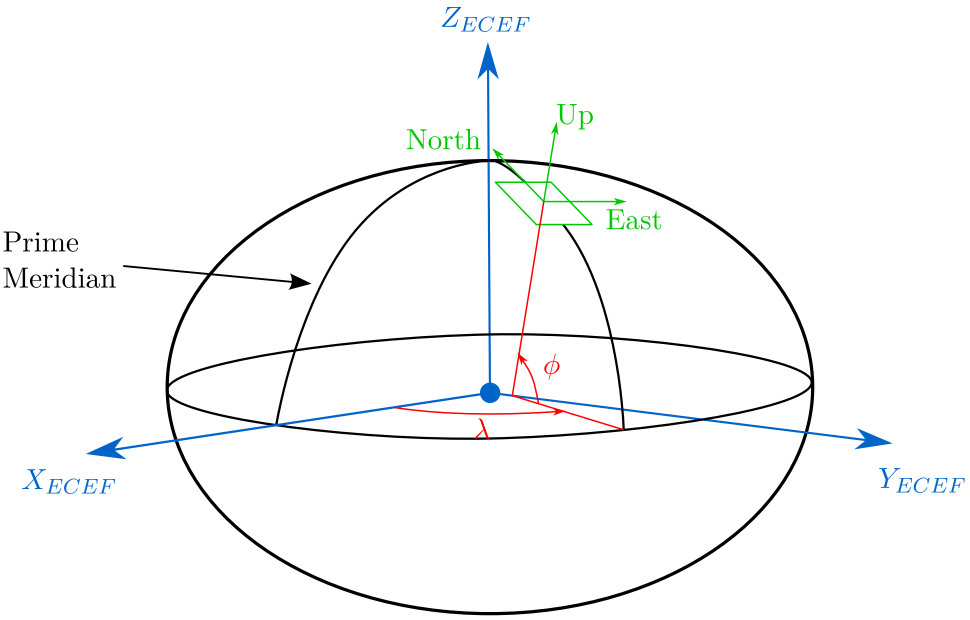


Figure 5.3: Different coordinate systems used by our application: The global ECEF coordinate system (blue), the geodetic coordinate system (red) and the local ENU frame at a given location on the ellipsoid surface. This figure is inspired by a drawing seen in [110].

denoted by (x, y, z) and if the point lies on the ellipsoid surface it is denoted by (x_s, y_s, z_s). The WGS84 ellipsoid is centred at the global origin and defined by its axis length's (a, b, c).

5.3.1 Geodetic to ECEF

Converting geodetic coordinates (λ, ϕ, h) to Cartesian coordinates (x, y, z) can be done by using the geodetic surface normal $\boldsymbol{\hat{n}_s}$ and the unnormalized geodetic surface normal $\boldsymbol{n_s}$ at the surface point $\boldsymbol{r_s} = (x_s, y_s, z_s)$ [21].

$$\boldsymbol{\hat{n}_s} = cos\phi \; cos\lambda \; \boldsymbol{\hat{\imath}} + cos\phi \; sin\lambda \; \boldsymbol{\hat{\jmath}} + sin\phi \; \boldsymbol{\hat{k}} \tag{5.2}$$

$$\boldsymbol{n_s} = \frac{x_s}{a^2}\boldsymbol{\hat{\imath}} + \frac{y_s}{b^2}\boldsymbol{\hat{\jmath}} + \frac{z_s}{c^2}\boldsymbol{\hat{k}} \tag{5.3}$$

The symbols $\boldsymbol{\hat{\imath}}$, $\boldsymbol{\hat{\jmath}}$ and $\boldsymbol{\hat{k}}$ in Equation 5.2 and Equation 5.3 stand for the unit vectors representing the axes of the ECEF coordinate system. Relating $\hat{n}_s$ to n_s, which point both in the same direction but have different magnitudes leads to

$$\hat{\boldsymbol{n}}_{\boldsymbol{s}} = \gamma \boldsymbol{n}_{\boldsymbol{s}}. \tag{5.4}$$

Substituting Equation 5.3 and rearrangement forms the following system of scalar equations

$$x_s = \frac{a^2 \hat{n}_x}{\gamma} \qquad y_s = \frac{b^2 \hat{n}_y}{\gamma} \qquad z_s = \frac{c^2 \hat{n}_z}{\gamma} \tag{5.5}$$

This system can be solved for $\boldsymbol{r}_{\boldsymbol{s}} = (x_s, y_s, z_s)$ by using the implicit ellipsoid representation denoted in Equation 5.6 as side condition to eliminate the unknown γ at the right hand side.

$$\frac{x_s^2}{a^2} + \frac{y_s^2}{b^2} + \frac{z_s^2}{c^2} = 1 \tag{5.6}$$

However, this side condition forces the computed points to lie on the ellipsoid surface. For points below or above the surface the input height h is used to compute a height vector $\boldsymbol{h} = h\hat{\boldsymbol{n}}_{\boldsymbol{s}}$ along the surface normal $\hat{\boldsymbol{n}}_{\boldsymbol{s}}$. This vector is at last added to the previously calculated surface point $\boldsymbol{r}_{\boldsymbol{s}}$ to retrieve the final point $\boldsymbol{r} = \boldsymbol{r}_{\boldsymbol{s}} + \boldsymbol{h}$.

5.3.2 ECEF to Geodetic

The way from Cartesian coordinates of an arbitrary point to geodetic coordinates is not as straightforward as the opposite direction. The approach that we use in our application consists mainly of three steps. First, the arbitrary point is scaled to the geodetic surface along its geodetic surface normal $\hat{\boldsymbol{n}}_{\boldsymbol{s}}$, which is the normalized $\boldsymbol{h}$-vector seen in Figure 5.4. This step is an iterative process as can be seen in the next paragraphs. After that a simple conversion of the point on the ellipsoid surface to geodetic coordinates can be made. At last the height is computed to move the point to its final position [21].

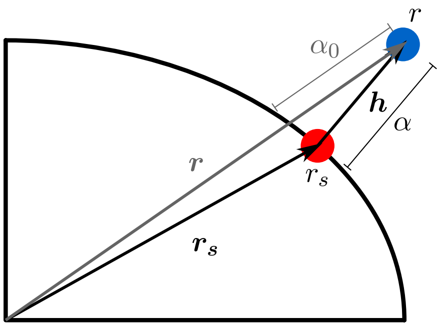


Figure 5.4: Scaling a known point r along $\boldsymbol{h}$ to the surface in order to compute r_s. r_s can then be used to calculate the geodetic coordinates in closed-form. For more details see the text. This figure is redrawn as seen in [21].

The first step, as already mentioned above, is the scaling of an arbitrary point, denoted by its position vector $\boldsymbol{r}$, to its surface point denoted by its position vector $\boldsymbol{r_s}$. This procedure is illustrated in Figure 5.4. Equation 5.3 shows that an unnormalized surface normal $\boldsymbol{n_s}$ can be computed by using $\boldsymbol{r_s}$ and the ellipsoid parameters (a, b, c). The height vector $\boldsymbol{h}$ depicted in Figure 5.4 points in the same direction and therefore differs from $\boldsymbol{n_s}$ only by its magnitude, which we denote by α. Summarizing this paragraph, we can define the vector $\boldsymbol{r}$ as following:

$$\boldsymbol{r} = \boldsymbol{r_s} + \alpha \boldsymbol{n_s} \tag{5.7}$$

Substituting Equation 5.3 in Equation 5.7, rearranging such that the searched vector $\boldsymbol{r_s} = (x_s, y_s, z_s)$ is placed on the left-hand side leads to the following system of equations:

$$x_s = \frac{x}{1 + \frac{\alpha}{a^2}} \qquad y_s = \frac{y}{1 + \frac{\alpha}{b^2}} \qquad z_s = \frac{z}{1 + \frac{\alpha}{c^2}} \tag{5.8}$$

α is now the only unknown left at the right-hand side, because we know the input point $\boldsymbol{r} = (x, y, z)$ and the ellipsoid parameters (a, b, c). To find α the implicit ellipsoid representation Equation 5.6 is used. Combining this equation with the equations seen in Equation 5.8 result in the following equation:

$$S = \frac{x^2}{a^2(1 + \frac{\alpha}{a^2})^2} + \frac{y^2}{b^2(1 + \frac{\alpha}{b^2})^2} + \frac{z^2}{c^2(1 + \frac{\alpha}{c^2})^2} - 1 = 0 \tag{5.9}$$

This non-linear equation can be solved for the only unknown α by using an approximation procedure known as Newton-Raphson method. For a value α that fulfils the equation $S = 0$, the point r_s on the ellipsoid surface is found. Initially, α is guessed as $\alpha = \alpha_0$, that would lead to a geocentric surface point, which lies at the intersection of the ellipsoid and the position vector of the input point r, as can be seen in Figure 5.4. The Newton-Raphson method now iterates by subsequently computing a new α according to Equation 5.10 until a value is found that makes S sufficiently close to 0.

$$\alpha_{i+1} = \alpha_i - \frac{S(\alpha_i)}{\frac{\partial S(\alpha_i)}{\partial \alpha}} \tag{5.10}$$

Having found α, r_s can be computed as shown in Equation 5.8, which leads to the second step. Converting the Cartesian coordinates of points on the ellipsoid's surface to geodetic coordinates can be done in closed-form using simple trigonometry. As already seen above in Equation 5.3, an unnormalized surface normal can be computed by using the surface point r_s. This normal can be normalized using its length to obtain the normalized geodetic surface normal $\boldsymbol{\hat{n}_s}$ at the surface point r_s. $\boldsymbol{\hat{n}_s}$ is then used to calculate the geodetic longitude λ and the geodetic latitude ϕ:

$$\lambda = arctan\frac{\hat{n}_y}{\hat{n}_x} \qquad \phi = arcsin\frac{\hat{n}_z}{||\boldsymbol{n_s}||} \tag{5.11}$$

The height value h, as the last missing component of the (λ, ϕ, h)-tupel, is computed from the height vector $\boldsymbol{h} = \boldsymbol{r} - \boldsymbol{r_s}$:

$$h = sign(\boldsymbol{h} \cdot \boldsymbol{r}) \; ||\boldsymbol{h}|| \tag{5.12}$$

5.3.3 ECEF to/from ENU

The last conversion algorithm covered in this chapter is the conversion between two Cartesian frames, namely the local east-north-up ENU frame and the global earth-centered, earth-fixed ECEF frame [100].

To convert ENU coordinates to global ECEF coordinates, the frames must be aligned such that their axes coincide. Figure 5.3 shows the relationship between the local, green coloured ENU coordinate system and the global, blue coloured ECEF coordinate system. It can be seen, that the frames can be aligned by two rotations. First a clockwise rotation about the ENU's east-axis by $90° - \phi$ aligns the ENU's up-axis with the ECEF frame's z-axis. Then a $90° + \lambda$ clockwise rotation about the Z_{ECEF}-axis aligns the east-axis of the ENU frame with the x-axis of the ECEF-frame. The angles λ and ϕ are the geodetic coordinates of the ENU frame's origin on the ellipsoid surface. Combining these two rotations leads to the following transformation:

$$\begin{bmatrix} X_{ECEF} \\ Y_{ECEF} \\ Z_{ECEF} \end{bmatrix} = \begin{pmatrix} -\sin\lambda & -\cos\lambda\sin\phi & \cos\lambda\cos\phi \\ \cos\lambda & -\sin\lambda\sin\phi & \sin\lambda\cos\phi \\ 0 & \cos\phi & \sin\phi \end{pmatrix} \begin{bmatrix} E \\ N \\ U \end{bmatrix} \tag{5.13}$$

The column vectors of the rotation matrix are the unit vectors pointing in the axis direction of the ENU frame given in ECEF coordinates. It can be seen that the third column vector consists of the coefficients used in Equation 5.3, which define the normalized geodetic normal $\boldsymbol{\hat{n}_s}$ at the origin of the ENU frame.

The conversion from ECEF to ENU coordinates is done by using the inverse transformation of Equation 5.13. Since the transformation matrix is a rotation matrix with the property $R^{-1}(\alpha) = R^T(\alpha)$, the inverse transformation matrix is simply the transposed original transformation matrix:

$$\begin{bmatrix} E \\ N \\ U \end{bmatrix} = \begin{pmatrix} -\sin\lambda & \cos\lambda & 0 \\ -\cos\lambda\sin\phi & -\sin\lambda\sin\phi & \cos\phi \\ \cos\lambda\cos\phi & \sin\lambda\cos\phi & \sin\phi \end{pmatrix} \begin{bmatrix} X_{ECEF} \\ Y_{ECEF} \\ Z_{ECEF} \end{bmatrix} \tag{5.14}$$

The above described transformations rotate vectors from the ECEF frame to the ENU frame and vice-versa. However, they lack the translation from the global origin to the origin of the local coordinate system. This translation is defined by the Cartesian position vector of the ENU frame's origin at the ellipsoid surface, because the origin of the ECEF frame is defined as the center of the ellipsoid at (0,0,0).

CHAPTER 6

Implementation

This chapter aims to provide technical details about the realized collaborative structure-from-motion system. Special code snippets are reviewed and the overall code structure of the implemented framework is explained in more detail.

6.1 System overview

The individual parts of the client-server architecture of our OpenSfM system are depicted in Figure 6.1.

What a user sees when visiting our webpage `opensfm.cg.tuwien.ac.at` is the front-end, which acts as an interface to the underlying structure-from-motion system. With the help of the front-end the user can browse or edit existing SfM-datasets or add new datasets to the system. More details about the front-end implementation and features are covered in Section 6.2.

Since our system is a client-server application, the front-end generates requests, which are processed at the server-side by our back-end implementation. The server application answers client requests and processes the data that is uploaded onto the system by using a number of independent worker threads, which finally notify the user about a successful or unsuccessful processing of their data. Furthermore, it provides an interface to the OpenSfM database, which stores the data that is presented by the front-end. Section 6.3 gives more detailed information about the back-end realization.

The underlying OpenSfM database contains the data populated into the structure-from-motion system. Client requests are usually fulfilled by the server application using the information provided by the database. The table layout and database-related details are introduced in Section 6.4.

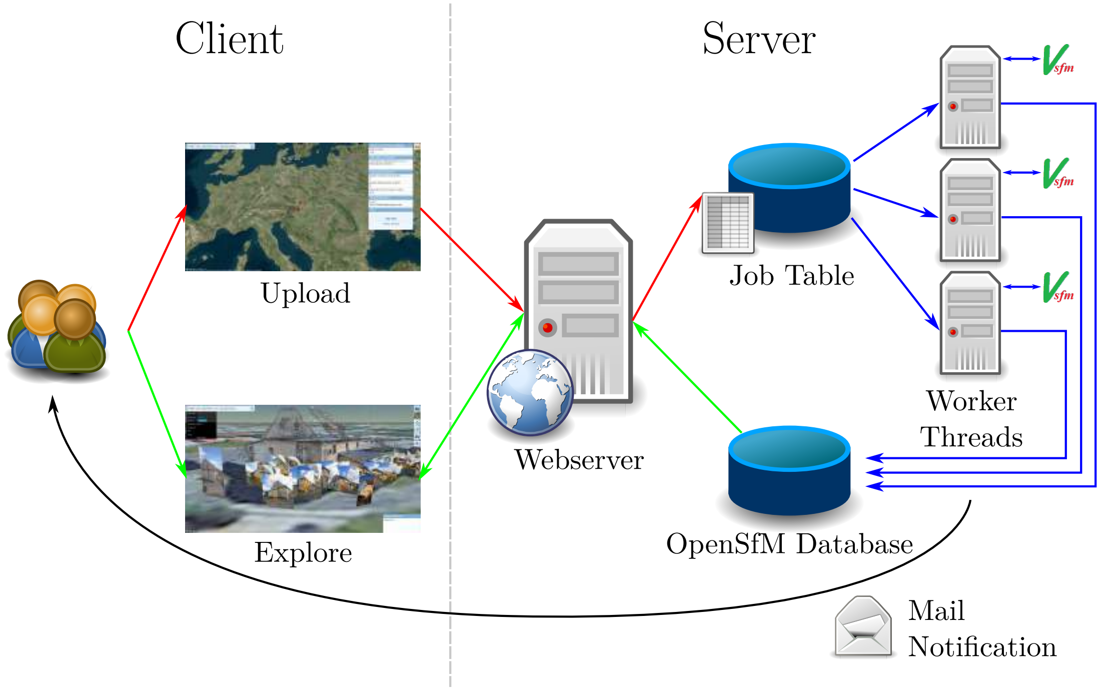


Figure 6.1: A user accesses our system via a webpage that shows the OpenSfM front-end. This virtual tourism system offers different modes that enable either upload, edit or exploration features. The uploaded information is processed with the help of a job pattern. The webserver inserts jobs consisting of input parameters for the SfM engine and the point cloud preparation, the users email address and an ID that identifies the uploaded images on the servers file system into a database table (red arrows). A second web application polls this database table and starts worker threads that process their assigned job with the help of VisualSFM, a state-of-the-art SfM engine, and inserts the finally generated and preprocessed SfM-dataset into the OpenSfM database (blue arrows). The user is notified per mail about the final processing state (black arrow). If his data was processed successfully, he can visit the OpenSfM front-end to explore the generated point cloud together with its corresponding camera information (green arrows).

6.2 The front-end: A virtual tourism client

The front-end is realized as a single-page web application using HTML, CSS and JavaScript with WebGL. HTTP requests are sent to the webserver with the help of the XmlHttpRequests API. This API allows sending and receiving data in form of HTML, XML, plain text, JSON objects and even byte streams. This makes it possible to request any form of data from the server, which is especially useful for handling the renderer requests (for more details see Section 6.2.3).

6.2.1 Code structure

To ensure a modular and easily maintainable front-end code structure, the JavaScript code is organized by using “widgets” that break the code and the user interface into small independent and reusable parts. The widgets are implemented following the so-called asynchronous module definition (AMD) specification that allows an asynchronous loading of the widgets and their dependencies. In the context of AMD, a widget represents a single module that is loaded together with its dependencies on demand using the dynamic script loader *RequireJS*. To further reduce the loading time of our front-end, the JavaScript and CSS files are minified by using the RequireJS library. This minification step removes all unnecessary characters from the source code without changing its functionality in order to reduce its file size and therefore the loading time of the application.

When implementing the widgets, we followed the model-view-viewmodel (MVVM) pattern that separates the presentation layer from the business logic. To realize this pattern in JavaScript, we use a library called *Knockout*. This library allows the definition of data-bindings for the elements of the view layer. This means that when the data referred to in the bindings changes, the presentation layer automatically refreshes itself in order to render the correct underlying information. Each of our widgets consists of a view with data-bindings dynamically created using JavaScript and a viewmodel, also realized using JavaScript, that contains the business logic behind its corresponding view. The following listing describes the widgets we use in our front-end implementation:

CesiumWidget

The CesiumWidget is a module from the *CesiumJS* library that renders the main interface of our webapplication: the globe [3]. The locations of the SfM-datasets are marked on the globe using viewport-aligned icons.

PotreeViewer

The PotreeViewer integrates the Potree renderer [87] into our virtual tourism system. This widget gets active when a user chooses a dataset by double-clicking an icon rendered on top of the globe.

GeoCoder

The GeoCoder widget is based on the geocoder widget contained in the CesiumJS library and allows the user to search for and visit specific addresses on the globe. This widget is realized by using the Bing Maps location webservice, that returns geographic coordinates for a specific input address. We improved the geocoder widget of the CesiumJS library by adding an autocompletion feature.

BaseLayerPicker

The BaseLayerPicker widget is like the CesiumWidget a module from the CesiumJS library. However, we replaced the view of the widget with our own view implementation that fits the graphical user interface (GUI) of our front-end. This widget allows to switch between different globe textures/tiles and terrain meshes.

VerticalTabBar
The VerticalTabBar widget represents the main menu of our webapplication. It is placed on the right border of the screen and contains several menu items which are used to interact with the system.

SfmUploadForm
The SfmUploadForm widget represents the form one has to fill in when he wants to add new datasets to the system or extend existing datasets.

SfmPopupMenu
The SfmPopupMenu widget appears when a user selects a location on the globe, where more than one dataset is placed. This popup menu allows an additional selection to identify the desired dataset.

Several CallbackButtons
The OpenSfM front-end contains several buttons that execute a specific functionality when clicked. Such buttons are defined with the help of a callback function, that is executed when the user clicks the button.

6.2.2 System modes and GUI

In the OpenSfM front-end the user can switch between four different modes that we call *system modes*. The graphical user interface (GUI) is adapted to the currently active system mode and is implemented in JavaScript using the user interface library jQueryUI. The following subsections explain the different system modes and their GUIs in more detail.

Overview mode

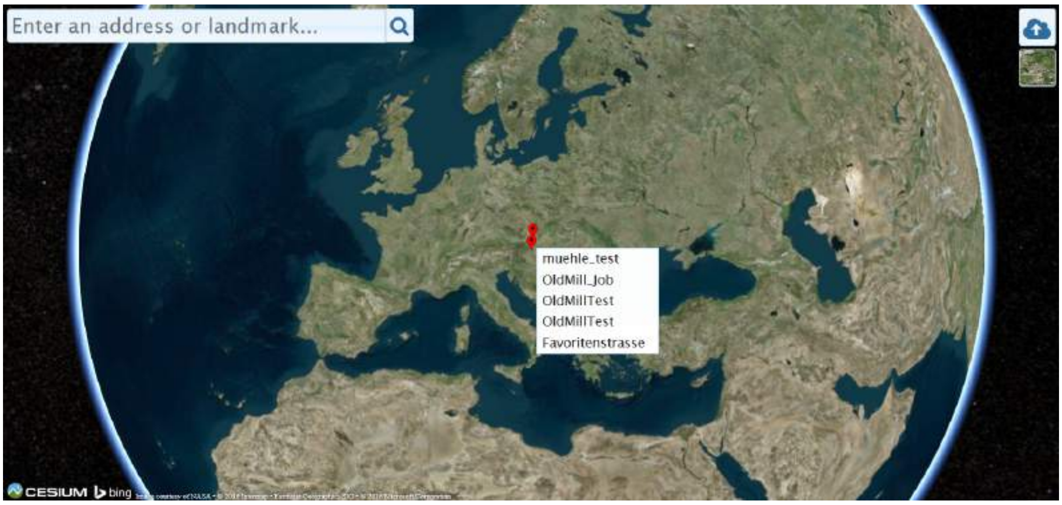
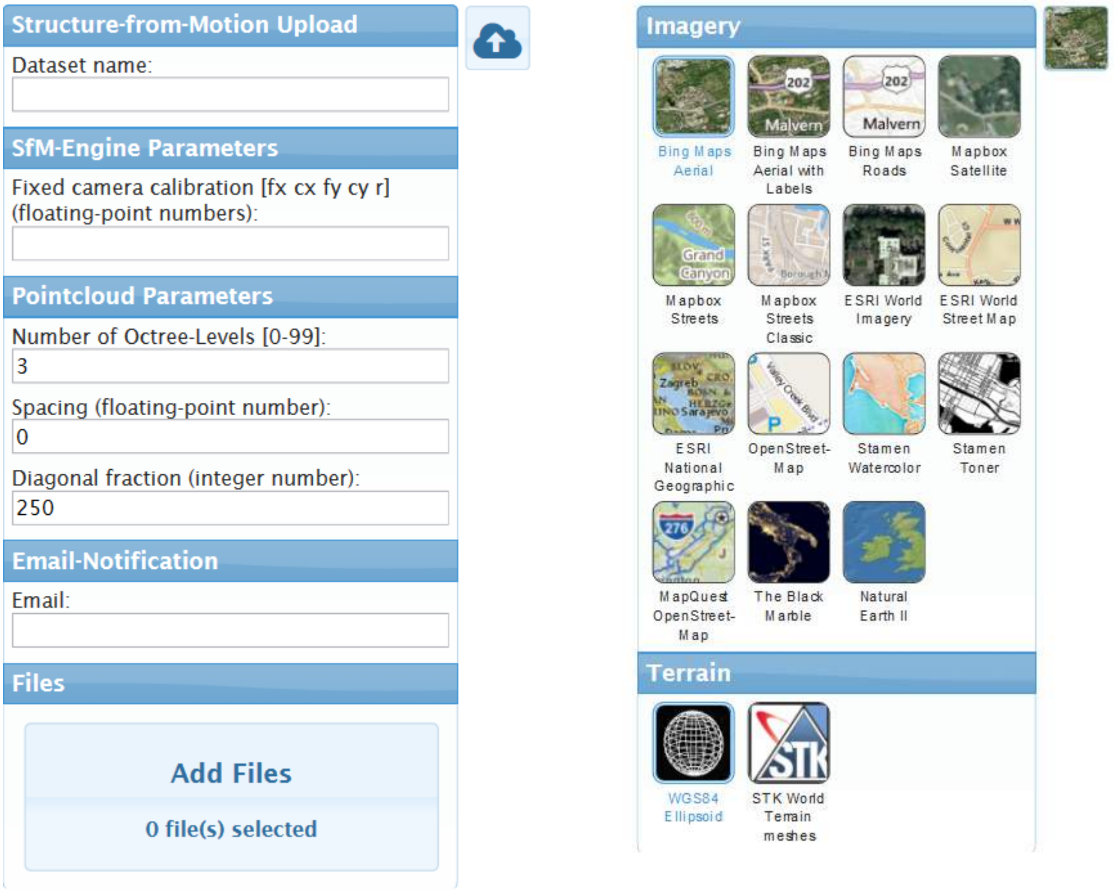


Figure 6.2: Screenshot of the OpenSfM webapplication in overview mode.

The overview mode depicted in Figure 6.2 is active when the user enters our webpage. He can explore the globe and search for SfM-dataset locations, marked by red location signs, either by manually navigating on the globe or by using the GeoCoder on the top left corner. When he double clicks a location sign, the underlying SfM-dataset is loaded and the application switches to the view mode described below. If the selected location offers more than one dataset, an additional selection must be done via a popup menu to switch to the desired SfM-dataset.


Structure-from-Motion Upload

Dataset name:

SfM-Engine Parameters

Fixed camera calibration [fx cx fy cy r] (floating-point numbers):

Pointcloud Parameters

Number of Octree-Levels [0-99]: 3

Spacing (floating-point number): 0

Diagonal fraction (integer number): 250

Email-Notification

Email:

Files

Add Files

0 file(s) selected

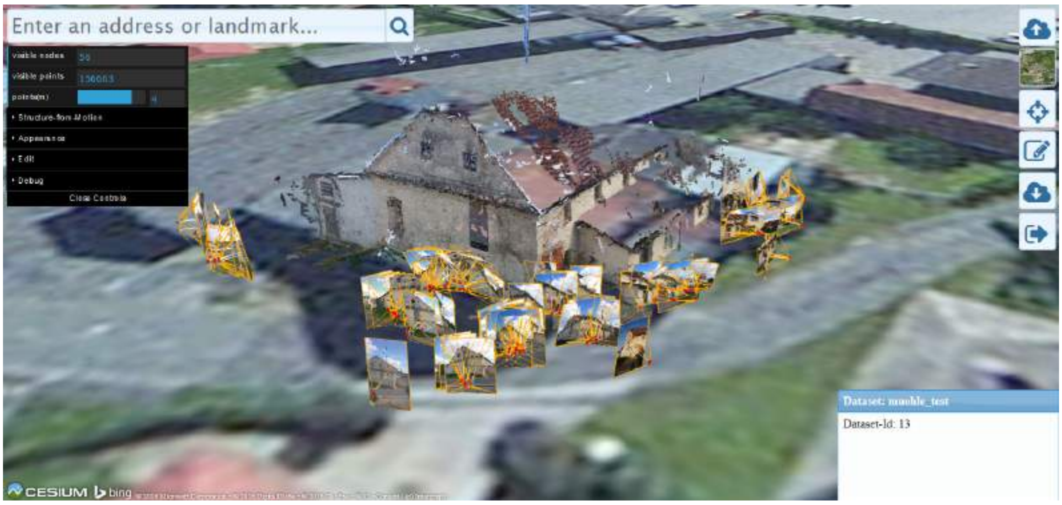
(a) OpenSfM upload form and its menu button (b) Layer picker and its menu button

Figure 6.3: The upload and layer picker menus that are accessible via the menu items depicted beneath each form.

Furthermore, the overview mode allows to add new datasets to the system by using the SfM upload form that can be accessed by clicking the upload button (cf. Figure 6.3a). The form is partitioned into five input areas. The first part at the top lets the user insert the name of the dataset. The second input field allows the definition of a fixed camera calibration that is used by the SfM engine at the backend for the 3D reconstruction. The third area consists of three input fields. Here the user can enter alternative values for controlling the point cloud preprocessing step at the backend that generates the octree data structure needed by the Potree pointcloud renderer. The email address that must be entered into the next field is used to inform the user about the final processing result of his uploaded image data. Finally, the last input area is the most important. Here the user defines the images that are uploaded to the system and that are used for 3D reconstruction. Furthermore, the user has the possibility to upload a so-called gcp-file (ground control points) that is a simple text file containing GPS coordinates of at least 3 input images. This file is needed if the uploaded images do not contain localization information in their EXIF tags. Otherwise the back-end cannot georeference the generated point cloud and the final processing step fails (cf. Section 6.3).

The layer picker shown in Figure 6.3b allows to switch between different globe tiles and terrain meshes offered by the BaseLayerPicker widget included in the CesiumJS API. By default the Bing maps tiles are rendered on the WGS84 ellipsoid. However, as can be seen in the figure, several other imageries and terrains like the OpenStreetMap tiles and the STK World Terrain meshes can be chosen.

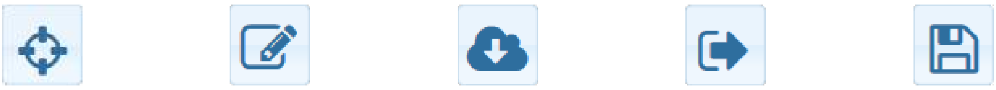
View mode



Dataset-Id: 13

Figure 6.4: Screenshot of the OpenSfM webapplication in view mode.

The user enters the view mode by double-clicking an icon, placed on the globe, that represents an SfM-dataset. The camera moves to the dataset and the renderer starts drawing the corresponding point cloud and cameras. As can be seen in Figure 6.4, the vertical menu on the right side of the application now contains four additional menu items. The menu item depicted in Figure 6.5a allows zooming to the viewed dataset if the user looses the dataset while navigating on the globe. A click on the button represented by Figure 6.5b enters the edit mode that is described below. The download button shown in Figure 6.5c lets the user download the viewed dataset as a compressed zip-file containing the dataset images, the VisualSFM nvm-project file that describes the camera parameters and the underlying sparse reconstruction, an alignment aln-file that stores the transformation extracted by georeferencing and a user transformation specified with the help of the edit mode. The fourth menu item depicted in Figure 6.5d exits the view or edit mode and returns to the overview mode. Besides these four icons, both menus already available in overview mode are still active. However, the SfM upload menu is now used to extend the currently viewed dataset by uploading additional images.



(a) Zoom to dataset (view/edit mode)

(b) Enter edit mode (view mode)

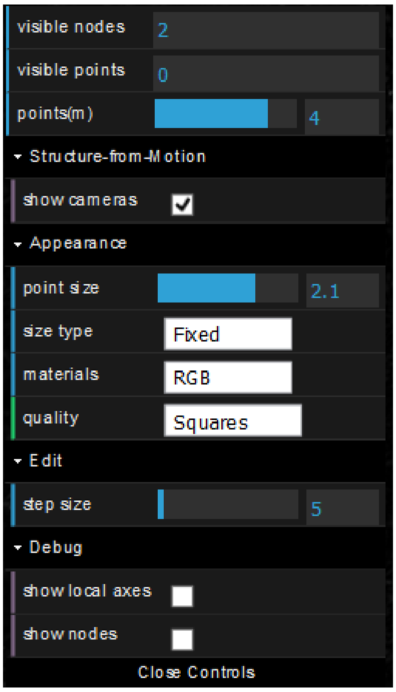
(c) Download dataset (view/edit mode)

(d) Exit view/edit mode (view/edit mode)

(e) Save user transformations (edit mode)

Figure 6.5: Different menu items available in view and edit mode.

An info box in the bottom-right corner of the screen contains additional information about the dataset. The current version of OpenSfM stores just the dataset name. Therefore the internal dataset ID is the only additional information to be displayed here.



visible nodes 2

visible points 0

points(m) 4

Structure-from-Motion

show cameras ☑

Appearance

point size 2.1

size type Fixed

materials RGB

quality Squares

Edit

step size 5

Debug

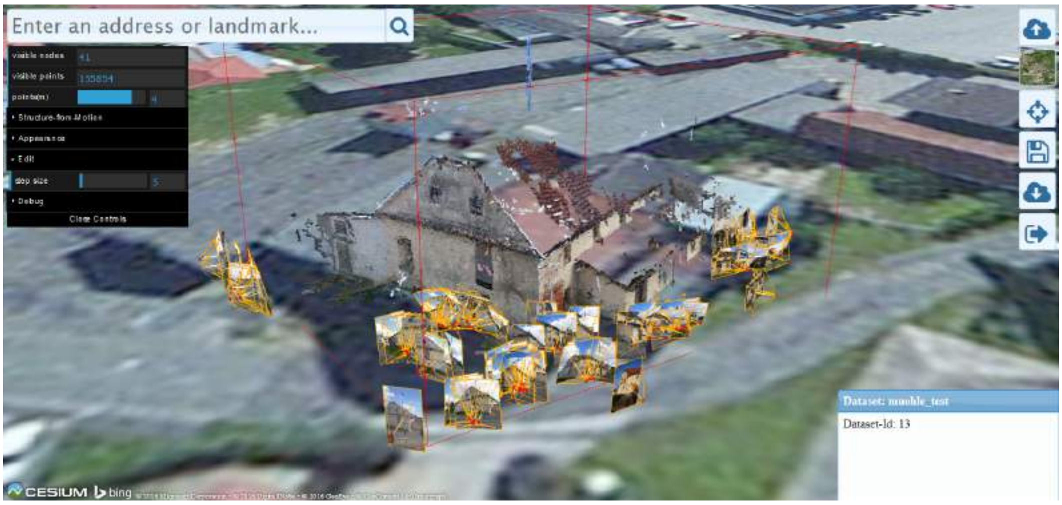
show local axes ☐

show nodes ☐

Figure 6.6: Menu used to control the rendering of the point cloud and cameras.

Furthermore the menu shown in Figure 6.6 and placed below the GeoCoder in our application can be used to control the point cloud and camera rendering. The menu is realized using the lightweight user interface library called dat.gui. It shows how many octree nodes and points are currently rendered. The point budget of the point cloud renderer that defines the maximum number of renderable points, can be adjusted using this menu. A checkbox allows to show or hide the cameras of the SfM-dataset. Furthermore the appearance of the rendered point cloud can be controlled. Here the point size, the used point size calculation algorithm, the material and the rendering quality can be adjusted. The different parameters are covered in more detail in Section 6.2.3. Another menu entry is used to adapt the step size used in the edit mode, when translating the point cloud. The last two entries are used to control debug flags that let the user show or hide the axes of the point cloud's local coordinate ENU coordinate system and the nodes of the point cloud's octree data structure.

Edit mode



visible nodes 41

visible points 155854

points(m) 4

step size 5

Dataset: mouble_test

Dataset-Id: 13

Figure 6.7: Screenshot of the OpenSfM webapplication in edit mode.

The edit mode is activated by clicking the "Enter edit mode" button placed in the vertical menu on the right side of the screen. It allows the selection and transformation of point clouds. Within the edit mode, the button used to enter the edit mode is replaced by a save button shown in Figure 6.5e. The save button is needed to store the user-defined transformations in the database. When the dataset is visited again, the stored transformations is applied to the point cloud.

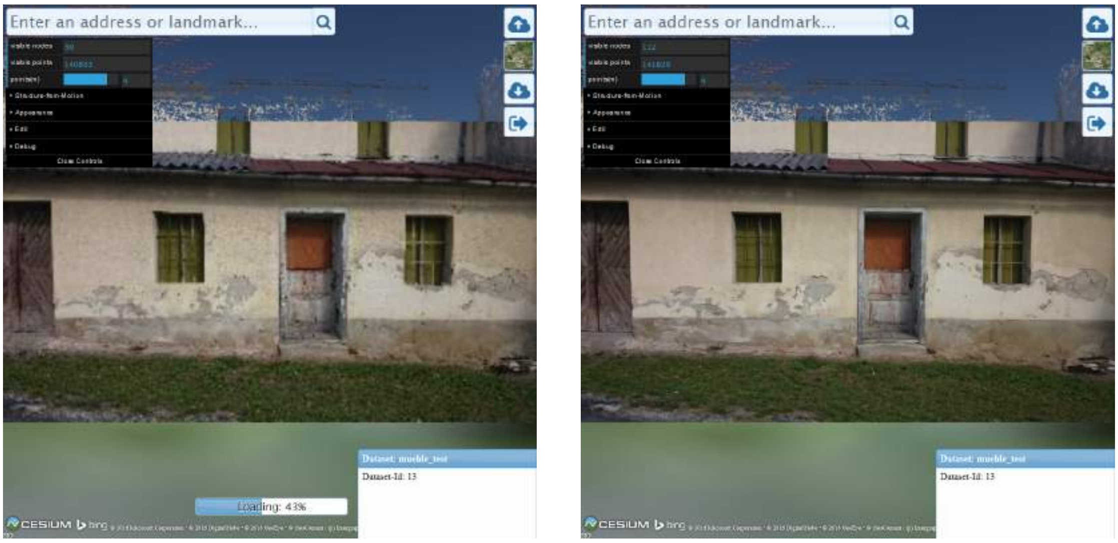
As can be seen in Figure 6.7, a selected point cloud is marked by rendering the outline of its bounding box in red colour. The point selection algorithm implemented in the Potree library is used to select the point clouds. This algorithm simply renders 3 byte point indices (RGB component) and 1 byte octree node indices (alpha component) into an offscreen render target that fits the application's window size. When the user clicks on the screen, the buffer entries in a fixed-size area around the mouse position are checked. If there are indices rendered in this area the point with the index that is rendered nearest to the mouse position is selected. At last, the point cloud that contains this point is selected.

Having selected a point cloud, different keyboard short-cuts exist to transform it. Table 6.1 lists all available operations together with their short-cuts. The precision of the transformations can be adjusted in the rendering menu by setting an appropriate step size in the “Edit” submenu.

Transform	Short-cut	Transform	Short-cut
Translate north	Up	Rotate CW east-axis	Alt + Right
Translate south	Down	Rotate CCW east-axis	Alt + Left
Translate east	Right	Rotate CW north-axis	Alt + Up
Translate west	Left	Rotate CCW north-axis	Alt + Down
Translate up	Plus	Rotate CW up-axis	Alt + Plus
Translate down	Minus	Rotate CCW up-axis	Alt + Minus
Increase uniform scale	Shift + Plus		
Decrease uniform scale	Shift + Minus		

Table 6.1: Keyboard short-cuts in edit mode.

Photo mode


Enter an address or landmark...
Loading: 43%
Close Controls
CESIUM
Enter an address or landmark...
Close Controls
CESIUM

Dataset: mueble_test
Dataset-Id: 13

Dataset: mueble_test
Dataset-Id: 13

(a) Low-quality image

(b) High-quality image

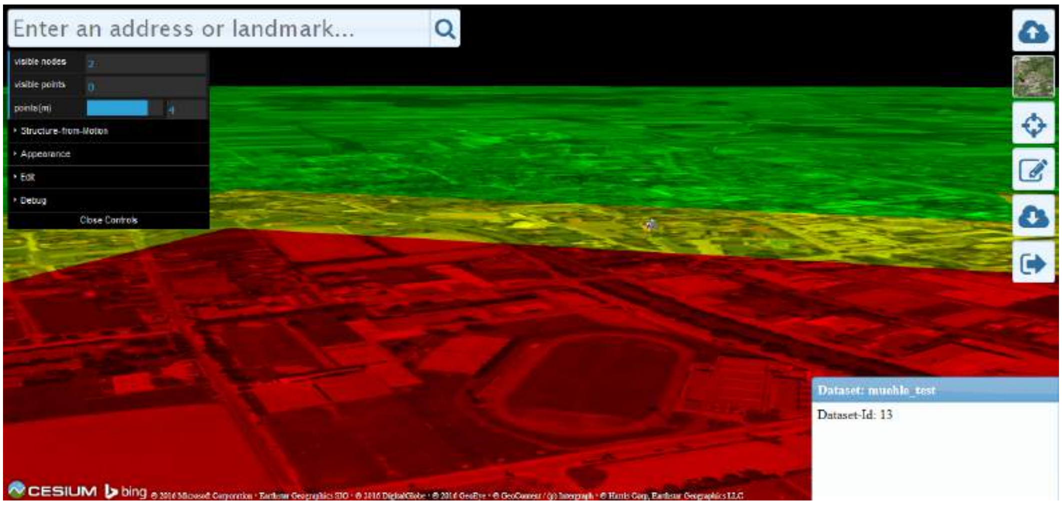
Figure 6.8: Screenshot of the OpenSfM webapplication in photo mode.

The photo mode is entered by double-clicking an image in the view mode. The viewing camera moves to the image of the selected camera such that it fits the screen space and the globe navigation is deactivated (for more details see Section 6.2.5). Furthermore, in contrast to the view mode, which simply renders small thumbnail images, the photo mode loads a high-quality version of the image. Figure 6.8 depicts a screenshot of the photo mode. The left picture shows the low-quality image that is replaced in photo mode by its corresponding high-quality image as can be seen in the right picture. The reason for not loading the high-quality images already in view mode is that in view mode several hundred cameras could be rendered with their images placed as a texture on their view planes. Assigning each view plane a high-quality texture would unnecessarily fill up or even exceeded the graphics memory, because in overview mode the user usually focuses on the rendered point cloud instead of the camera images. Therefore we decided to load the high-quality texture just on demand, when the user enters the photo mode for a specific camera. The texture is replaced again with its low-quality thumbnail version when the user leaves the photo mode and returns to the view mode by double-clicking the screen.

6.2.3 Rendering

Since our front-end is a WebGL-based application, 3D rendering algorithms played an important role during the development process. In our application we distinguish between two renderers that create the output presented to the user. The renderer implemented in the CesiumJS library is the main renderer of our application. It is responsible for the background rendering including the globe with its tiles and meshes. The second renderer used in our system is an implementation based on the ThreeJS framework, which is a high-level JavaScript interface to the low-level WebGL API [69]. It is used to render the actual SfM-dataset including the point cloud data structure with the help of the Potree library.

Rendering virtual globes is not as straightforward as it may seem. The CesiumJS library approximates the shape of the virtual globe by using the tessellation pipeline for generating subdivision surfaces. However, when we deal with virtual globes, we deal with large world coordinates too. As already mentioned in Chapter 5, the ECEF coordinate system defined on the WGS84 ellipsoid covers the whole world, containing places with large coordinates, like the ones on the equator with an x-coordinate of 6378137.0 in metres, and even larger distances between two locations. To handle WGS84 coordinates and distances in the ECEF coordinate system, double-precision values are needed. However, WebGL is based on the OpenGL ES 2.0 specification for mobile 3D rendering and does not support double-precision floating-point values in the shader code [40]. Therefore, precision artifacts like z-fighting and jittering can occur when dealing with such large coordinate values. The rendering algorithm realized in the CesiumJS library overcomes these precision problems by partitioning the rendered scene into multiple view frustums and rendering it relative to the eye. The scene partitioning approach increases the depth precision and allows rendering large view distances without any z-fighting artifacts. Rendering the scene relative to the eye or camera and splitting up the double-precision vertex positions into two single-precision floating-point numbers by separating the high and low bits removes the jittering artifacts. For more detailed information about efficient



visible nodes 2
visible points 0
points(m) 4

Dataset: muehle_test
Dataset-Id: 13

Figure 6.9: Debug view showing the multiple view frustums into which the globe is partitioned on rendering. This is needed to remove z-fighting artifacts that may appear due to large vertex positions that suffer from floating-point round-off errors when using 32-bit vertex attributes in WebGL.

virtual globe rendering see the book by Cozzi and Ring, that describes the theoretical background behind the CesiumJS library and the SIGGRAPH '13 "Rendering massive virtual worlds" course notes [21][89].

One of our contributions when it comes to rendering the OpenSfM front-end was the integration of the ThreeJS-based OpenSfM renderer into the CesiumJS rendering pipeline. We choose ThreeJS for rendering the SfM-datasets because the Potree library is also built upon this library, which made the integration of the Potree's point cloud loader and renderer easier. The illustration depicted in Figure 6.10 shows how we integrated our OpenSfM renderer into CesiumJS' rendering pipeline. The left part of this figure contains the main packages of the CesiumJS library. The core package drawn at the bottom offers several mathematical algorithms like coordinate system transformations described in Chapter 5. The non-public renderer above is the interface to the underlying low-level WebGL API and actually draws the frames. The most important element for our work is the scene package that dispatches different high-level commands to the renderer. The datasource layer encapsulates different CesiumJS scene types and offers loaders for different assets like map tiles or 3D models. The widget package consists of several useful modules like the CesiumWidget that renders the globe or the above mentioned BaseLayerPicker. When integrating our renderer into the rendering pipeline of the CesiumJS library we modified the *render* method in the scene package. This render method gets a scene object that contains a list of frustums for multi-frustum rendering and a list of commands that encapsulate the final draw calls as parameters. We added an additional parameter that represents a ThreeJS renderer object. This *customRenderer* implements a render method that takes a *near* and *far* value for multi-

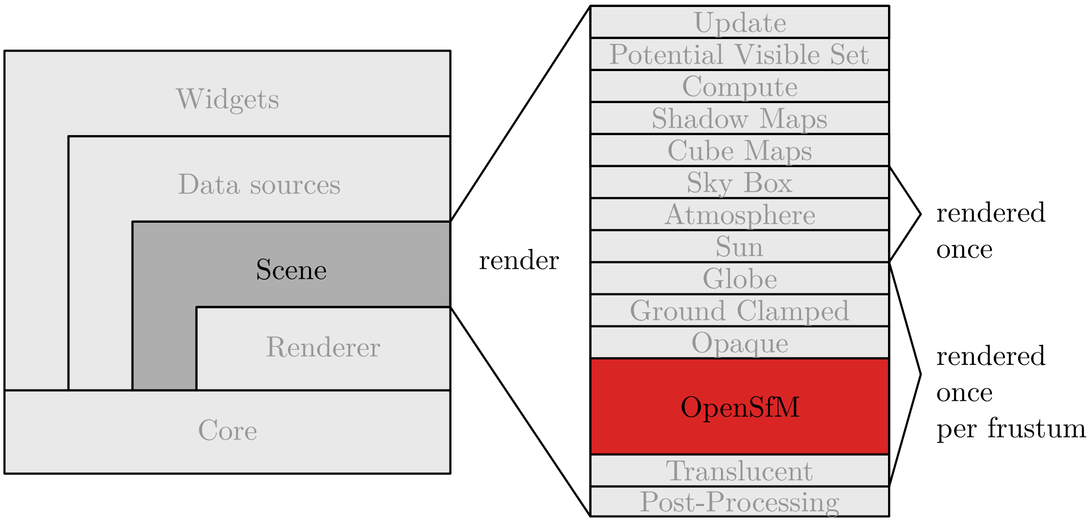


Figure 6.10: (left) The main elements of the CesiumJS framework. To integrate our OpenSfM renderer into the CesiumJS rendering pipeline we have to modify the render-method of the *Scene* package. (right) The pipeline that is traversed during rendering. We render our scene objects after the CesiumJS render command that draws opaque objects and before the one that draws translucent objects. For more details see the text. This figure is inspired by illustrations seen in [19][20].

frustum rendering. These values are needed by our OpenSfM renderer to draw the scene in the same partitioned way as the CesiumJS renderer in order to achieve correct depth testing across objects drawn by different renderers. Furthermore, our renderer draws the SfM scene after the opaque objects and before the translucent objects of the CesiumJS scene to prevent errors when rendering transparent CesiumJS scene objects. Another thing that has to be considered when harmonizing both renderers is to reset the WebGL state after having drawn the SfM scene in order to avoid conflicts with the following draw calls done by the CesiumJS renderer.

Coordinate systems and precision problems

The OpenSfM renderer itself uses the Potree library to render the octree-based point cloud data structure. Besides the point cloud it also renders the cameras together with their photographs that build the basis of the whole SfM-dataset.

When rendering georeferenced scene objects in a large coordinate system like ECEF, precision problems can occur, as already mentioned above. However, our system is not affected by such precision issues, because the scenes that are rendered by our OpenSfM renderer do not cover large distances in the ECEF coordinate system. Furthermore, the points are originally stored in the database in a local coordinate system used by the

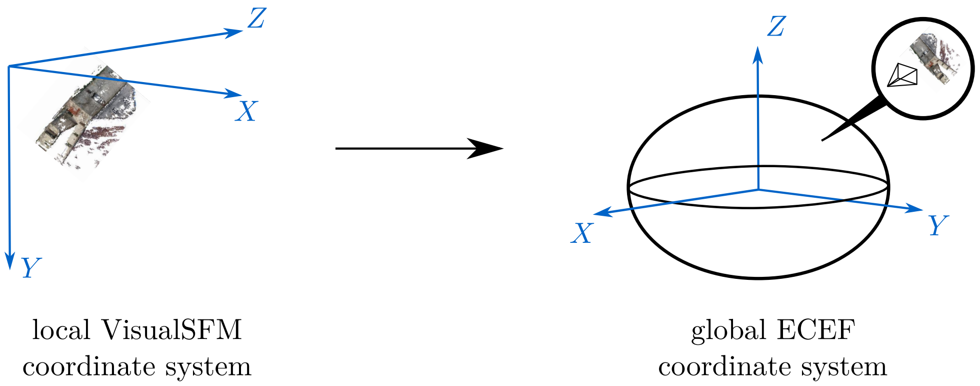

Z
X
Y
Z
X
Y
local VisualSFM coordinate system
global ECEF coordinate system

Figure 6.11: (left) The point datasets are originally stored in the database in a local coordinate system defined by VisualSFM. (right) To correctly place the dataset onto the surface of the globe, the local coordinates have to be converted into global ECEF coordinates.

VisualSFM software, that is used for reconstructing the scene and generate the point clouds. This right-handed coordinate system is often used in computer vision applications and has its y-axis pointing downwards and z-axis pointing in viewing direction into the scene, as can be seen in the left part of Figure 6.11. These point coordinates are directly used as single-precision vertex positions p_{vsfm} in the vertex shader.

The double-precision georeferencing transformation matrix M_{geo} is used as a part of the scene object's model matrices to finally place the whole scene into the global geographic ECEF coordinate system, as can be seen in the right part of Figure 6.11. This georeferencing places the dataset at its correct location on the virtual globe. Furthermore, the synchronization of the coordinate systems used by the independent renderers allows an easy synchronization of their viewing cameras as can be seen in Section 6.2.4. However, since this transformation is part of the scene object's model matrices, these must be defined in double-precision. The same applies to the camera's view matrix M_{view}, which also contains large values, since the camera is placed in the global ECEF coordinate system. As already mentioned above, WebGL does not support double precision values and uniforms in the vertex shader. Therefore, it is not possible to use the model matrix and the view matrix directly in the shader. However, combining both matrices to a so called model-view matrix M_{mv} and uploading this matrix as a uniform to the GPU solves our precision problem, because the large values in the two matrices are eliminated on matrix multiplication. This means, that if the camera is close enough to the rendered scene, the model-view matrix M_{mv} will always contain values representable by single-precision floating-point numbers.

The following equation denotes the position computation in the vertex shader. The matrix M_{proj} is the camera's projection matrix, which is defined in single-precision. The projection matrix is adapted before each render-call to the near and far values defined by the CesiumJS renderer for multi-frustum rendering. It can be seen, that this computation only contains single-precision matrices and vectors, which finally prevents precision artifacts.

$$gl\_Position = M_{proj} * M_{mv} * vec4(p_{vsfm}, 1.0); \tag{6.1}$$

The scenegraph

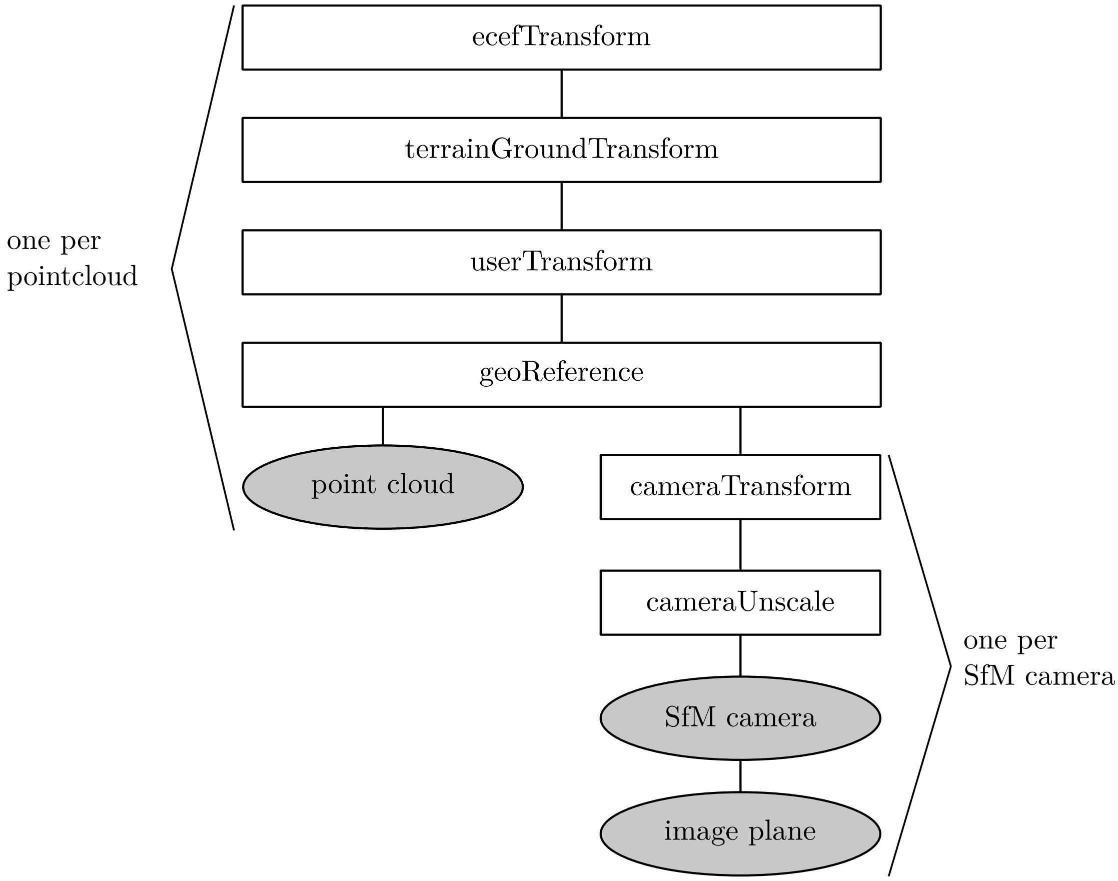


Figure 6.12: The scenegraph that is rendered by the OpenSfM renderer.

Figure 6.12 shows the scenegraph that is traversed by our renderer during the drawing process. It consists of three object nodes, depicted as ellipses with gray background, that store the actual scene objects and six transformation nodes that ensure that the scene objects are rendered at their correct positions in the coordinate system.

The *point cloud* node contains the point cloud object defined in the Potree library. This object represents the octree-based data structure that is converted into primitives on rendering. A loader algorithm that dynamically requests and reloads points from the server application on demand depending on the view and point cloud position adds to or removes points from this point cloud object.

For each SfM camera object in the scene, one *cameraTransform*, *cameraUnscale*, *SfM camera* and *image plane* node exists. The *cameraTransform* node moves the SfM camera to its correct position in the local VisualSFM coordinate system. Since this coordinate system has its y-axis pointing downwards another 180° rotation about the x-axis has to be done to fit the coordinate system used by WebGL in order to correctly rotate the SfM camera frustum visualizations. The *cameraUnscale* transformation is necessary to undo scaling operations contained in the *userTransform* or *geoReference* nodes placed higher in the scenegraph. Otherwise, the SfM camera frustums visualized with simple line geometry would be scaled according to the point cloud scale, which is not desired. The *SfM camera* node contains the frustum visualization and the *image plane* node a plane geometry textured with the photo behind the SfM camera.

The *geoReference* node converts the points and camera locations stored in local VisualSFM coordinates in the database to local ENU coordinates (cf. Chapter 5). This transformation is necessary to easily apply user transformations stored in the *userTransform* node, since they are defined in this local geographic coordinate system. Furthermore, terrain tiles that are loaded by the layer picker widget result in a translation of the whole dataset along the up-axis of the coordinate system. This translation is represented by the *terrainGroundTransform* node and ensures that the OpenSfM scene is still visible at the surface of the globe. To obtain the correct translation value, the loaded terrain dataset is sampled at the center of the OpenSfM scene's bounding box. The *ecefTransform* node transforms the whole scene from ENU to ECEF coordinates, to finally get an OpenSfM scene that is represented in the same coordinate system as the CesiumJS scene into which it is placed. Since an OpenSfM scene can contain more than one point cloud, namely one point cloud per reconstructed scene, all nodes from the *point cloud* node upwards appear once per point cloud in the OpenSfM scene.

6.2.4 Camera synchronization

As already mentioned in Section 6.2.3, we use two renderers to draw the front-end output. The CesiumJS renderer draws the globe in the background and our OpenSfM renderer, which we integrate into the CesiumJS rendering pipeline, draws the point cloud with the help of the Potree library and the cameras. However, each of these two renderer instances uses its own camera. These two cameras have to be synchronized in order to create a correct rendering output. This synchronization is done before rendering each frame by equalizing the camera matrices. Here it has to be considered that the CesiumJS camera matrix is defined column-wise, while the ThreeJS camera is defined row-wise. Listing 6.1 shows the code snippet that synchronizes the ThreeJS camera with the CesiumJS camera.

```
var civm = cesium.viewer.camera.inverseViewMatrix;

camera.matrixWorld.set(
civm[0], civm[4], civm[8], civm[12],
civm[1], civm[5], civm[9], civm[13],
civm[2], civm[6], civm[10], civm[14],
civm[3], civm[7], civm[11], civm[15]);
```

```
 8 camera.matrixWorldInverse = new THREE.Matrix4();
 9 camera.matrixWorldInverse.getInverse(camera.matrixWorld);
10 
11 var c = camera.clone();
12 c.near = near;
13 c.far = far;
14 c.updateProjectionMatrix();
```

Listing 6.1: Synchronization of the CesiumJS camera with the ThreeJS camera.

It is known that the inverse view matrix represents a cameras transformation matrix, that places the camera in a specific orientation in world space. Therefore in line 3 of Listing 6.1 the ThreeJS camera's world matrix is set to the inverse view matrix of the CesiumJS camera. Since we deactivated an automatic update of the ThreeJS cameras matrices, in line 9 we have to set the inverse world matrix manually. The remaining lines of the code snippet handle the fact that CesiumJS does multipe view frustum rendering, as already mentioned in Section 6.2.3. Therefore we have to adjust the ThreeJS camera's projection matrix such that its near and far plane equal the near and far plane of the currently rendered CesiumJS view frustum. To avoid reference problems, we clone the camera for this step.

6.2.5 Photo exploration

A user can enter our front-end's photo mode by double-clicking a photograph in view mode. This double-click triggers an animation that moves the main camera in front of the photo, such that the photo fills the screen. This animation is achieved by a linear transformation between a start position and orientation defined by the main camera's current state and an end position and orientation. The end orientation is extracted from the SfM camera's view matrix that belongs to the selected photo. The end position in world space is the position in perpendicular distance d in front of the image plane. Figure 6.13 illustrates the mathematical background behind the calculation of this distance d by using the formula denoted in Equation 6.2.

$$d = \frac{s}{2 \tan \frac{\alpha}{2}} \tag{6.2}$$

In cases where the camera's aspect ratio is less than the SfM camera's aspect ratio, α denotes the vertical field of view angle in radians and s the image plane width in world space. This finally leads to a camera placement that fits the image plane to the screen height. If the camera's aspect ratio is greater than or equal to the SfM camera's aspect ratio, α represents the horizontal field of view of the viewing camera in radians and s the image plane height in world space. In this szenario the viewing camera is placed in world space, such that the image plane fits the screen width. If both aspect ratios are equal, the image plane is evenly displayed in full screen.

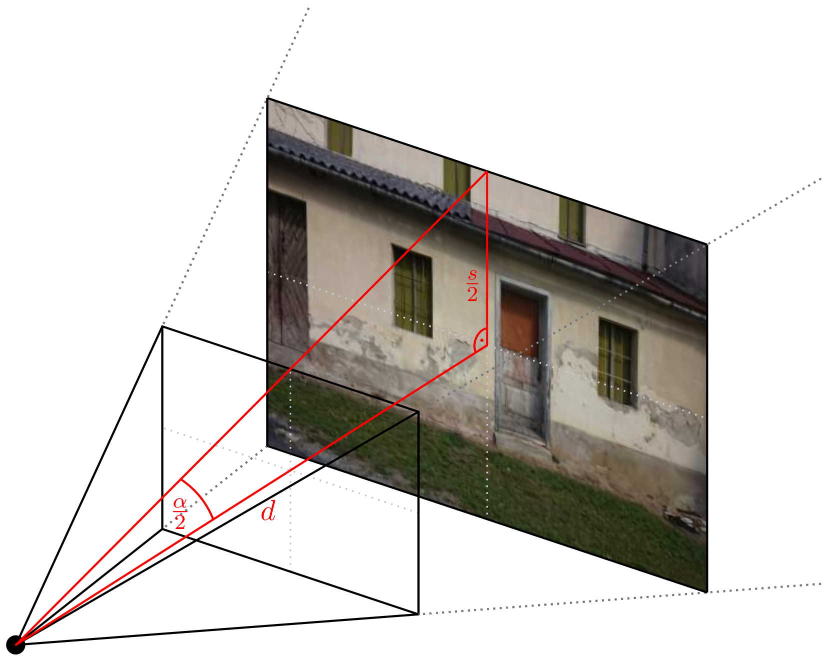


Figure 6.13: Illustration of the math behind our photo mode. The viewing camera is moved towards the selected photo, such that its frustum covers the whole image. This is achieved by placing the camera in a perpendicular distance d in front of the image plane. d is calculated using the trigonometric relationship denoted in Equation 6.2 between the half field-of-view angle in radians $\frac{\alpha}{2}$ and the half world space height of the image plane $\frac{s}{2}$.

6.3 The back-end: SfM, point cloud preparation and database interface

The back-end is a Java-based application hosted with the help of an Apache Tomcat web server. It is implemented using the Spring Web MVC framework, which provides a model-view-controller architecture that allows to build a clearly structured and loosely coupled application [103]. Its central element is the so-called *DispatcherServlet* that handles all HTTP requests by calling appropriate controller methods that process the received input data and update the underlying model. These methods return HTTP responses that are used by the client application to update its view rendered by the browser.

6.3.1 Code structure

The back-end code is organized in several modules to realize a multi-layered software architecture. Each layer implements a different logical part of the back-end. The following listing shortly describes the individual modules:

OpenSfM-Web

This module contains the implementation of the above mentioned controller. The methods of our *SfmDataController* class are invoked by the *DispatcherServlet* according to the request URL sent by the client. The controller functions call the underlying service-layer methods that implement the business logic. Furthermore, this module contains the presentation layer consisting of the front-end code in the *webapp* folder. This is required to easily build the final web application using the Maven build tool [31] for the back-end build and the Gulp build system for the front-end build [42].

OpenSfM-Service

Here the service-layer of our multi-layered architecture is realized. This layer encodes the business rules that define how data can be added, edited or deleted. Here the different algorithms are implemented that generate the octree-based Potree data structure. Furthermore, different file importer and exporter for VisualSFM project files (nvm-files) and point cloud files (ply-files) are realized. The implementation of the interface to the VisualSFM engine is also part of this module. The methods of this layer usually invoke persistence-layer methods if processed data changes must be stored in the database.

OpenSfM-Persistence

The persistence layer acts as an interface to the underlying database. It consists of so-called data-access-objects (DAOs) that offer methods to read information from the database or write data into the database.

OpenSfM-Job

The job module contains a second independent web application that polls a database table to detect and execute tasks. These tasks are enqueued by the main application that has its presentation layer in the OpenSfM-Web module.

OpenSfM-Log

This module is a helper module that implements a logging functionality for the two above mentioned web applications. It adds an additional page to the applications that offers access to the last 500 log entries. This module is realized with the help of the *logback* logging library.

6.3.2 Processing Potree renderer requests

The renderer requests created by the Potree point cloud renderer and all other HTTP requests are handled at the server-side by the OpenSfM back-end by evaluating the request URL using so-called *URI templates*. Listing 6.2 contains the method declaration of the controller function that handles the renderer requests on the server-side. The *value* parameter of the *@RequestMapping*-annotation describes the request URL as an URI template that contains the two variables *sfmid* and *pclid*. These variables are evaluated in the method header by using the *@PathVariable*-annotations. The wildcard at the end of the URL is evaluated by the *DispatcherServlet* and leads to a call of this function for every request targeting a URL pattern that starts with $/cloud/sfmid/pclid$.

```
@RequestMapping(value="/cloud/{sfmid}/{pclid}/**",method=RequestMethod.GET)
public @ResponseBody ResponseEntity<byte[]> loadHierarchyAndNodes(
    @PathVariable String sfmid, @PathVariable String pclid,
    HttpServletRequest request);
```

Listing 6.2: Controller method declaration of the function that handles the Potree renderer requests.

The *sfmid* and *pclid* are used to find the requested point cloud. The remaining last part contains a hierarchy path that uniquely identifies a required octree node. If this last part ends with the extension *.bin*, the point data of a node is requested, otherwise, if it ends with *.hrc* the underlying octree hierarchy information must be returned to the Potree renderer at the client-side.

Having identified the hierarchy path's extension, an appropriate service-layer method is called that requests the required information from the database via a data-access-object of the persistence-layer. The data delivered from the database is preprocessed such that the response sent to the client can be interpreted correctly by the Potree renderer. For more details about the database behind the OpenSfM backend see Section 6.4.

A readable hierarchy byte stream contains a list of 5-byte packets. The first byte contains a mask describing a node's children and the remainig four bytes represent the number of points stored in the node. This information is selected from the database using a recursive hierarchical query that traverses the octree node hierarchy. The bytes in the returned byte array must be in little endian order which means that they are ordered from least significant to most significant. This means for a 1-byte children mask of the form 00000011 that a node r has two descending children r_0 and r_1.

A valid point data byte stream describes each point of the requested octree node by using 3×4 byte unsigned integer position data and 4×1 byte unsigned char colour data. The points are already stored in this packed format in the database (cf. Section 6.4. However, the used database extension adds an additional 10-byte header to each point, describing the endianness and a reference to a point format definition, that must be truncated when retrieving the data. Like the hierarchy data explained above, the returned binary point data must be in little endian order.

6.3.3 Task processing

The back-end realizes two different task processing approaches. The first one is an asynchronous approach where the main web application inserts task descriptions into a database table. A task description consist of task parameters, a function to be called on task execution and a callback mail address. The database table is polled by a second application that detects new tasks and executes them using the information stored in the table. A notification is finally sent to the deposited mail address. This approach is used for the time-consuming SfM tasks that reconstruct the point clouds and generate the octree-based representation that is stored in the database. The second approach is synchronous, which means that a requested task is immediately executed by the main application and the result is returned to the client. This task processing method is used for simple requests like the above-described renderer requests, texture loading requests for the SfM camera's image planes and also requested download tasks that allow the user to export a dataset from the system and store it on his own file system for further usage.

The next two subsections explain the asynchronous SfM tasks and the synchronous download task in more detail.

SfM tasks

If a user uploads new images to the system, these images are either used to generate a new SfM-dataset or to extend an existing SfM-dataset. Which of the two SfM tasks gets triggered is controlled by the currently active system mode in the front-end, as already mentioned in Section 6.2.2. If the overview mode is active, a new SfM-dataset is created using the uploaded image information. Otherwise the currently active/rendered SfM-dataset will be extended to improve its 3D reconstruction.

In both cases a new task entry is created in the task table of the OpenSfM database. This entry stores different task-related properties like the parameters for the 3D reconstruction and octree-generation in a JSON-String attribute, the mail address entered in the front-end and the method that is invoked at the task execution. The method name marks a task as a “generation” task that creates new SfM-datasets or a “resume” task that extends existing datasets. The generated task-id, which is a database sequence value, is finally used to retrieve a folder on the server's file system that contains the uploaded images until the processing task deletes the folder when it is finished. The status attribute represents the current state of task processing:

- UPLOADING - data upload is in progress
- PENDING - task waits for execution
- EXECUTING - task is getting executed
- COMPLETED - task execution finished successfully
- ERROR - an error occurred during task execution

Additional attributes like tasktimestamp, status and error are used for logging purposes.

A second web application implements a so-called *CronJob* that polls a task table according to a configured cron expression stored in the application's property file on the server. We configured our job application such that the table gets polled every minute to detect unprocessed tasks waiting for execution. If an unprocessed task is found, the job application tries to start a new worker thread that finally executes it. However, the application can spawn only up to five worker threads for task processing due to performance reasons and hardware limitations, because the subsequently called SfM engine requires a lot of system memory especially when it has to process a lot of input images. If all threads are running, pending tasks must wait for processing until a worker thread finishes its task.

The VisualSFM engine that reconstructs the 3D scene offers a socket interface that can be used to control the communication between our application and the SfM engine (cf. Section 6.3.4 for more details). To run several engines in parallel, each of them has to listen on a different socket port. We achieve this by maintaining a thread-safe port-list that contains all currently used ports. If a new worker thread gets started, the next free port below the starting port 9999 is assigned to it. The VisualSFM engine is then started by the thread in server mode listening to commands on the assigned socket port. After having started the engine, our worker thread sends commands according to the assigned SfM task to it. The engine processes the received commands to generate a georeferenced dense scene reconstruction out of the input images stored in the corresponding task folder on the server and, in case of a “resume” task, existing information from the OpenSfM database. The worker thread uses the SfM engine’s output stored in the same task folder to create an octree-based point cloud representation that is finally persisted in the underlying OpenSfM database.

Download task

Besides the asynchronous SfM tasks that are executed by a separate web application, there exist synchronous tasks that are executed directly by the main web application. The download task is one of those synchronous tasks.

Its only input parameter is the *sfmid* of the currently viewed dataset. The download task uses this id to fetch the information about the dataset from the database. It saves the VisualSFM project file that stores the SfM camera parameters and a sparse reconstruction, the input images and alignment-files containing the transformation used for georeferencing and the user-defined transformation to a temporary folder on the server. The content of this folder gets subsequently compressed into a zip-file that is finally sent to the client. The user can save the generated file on his local file system and use the downloaded content for further offline processing.

6.3.4 VisualSFM integration

The above described SfM tasks use a state-of-the-art SfM-engine called VisualSFM to generate dense 3D scene reconstructions that are georeferenced using the GPS information stored in the input images. The output of the SfM-engine is stored in the task folder on the server for further reprocessing by our web application. The 3D reconstruction process is divided into several subtaks, as already mentioned in Chapter 3. First of all, the uploaded input images must be loaded by the SfM engine. After that, features and correspondences between features are detected. These are subsequently used to compute a sparse reconstruction and to recover the SfM camera's extrinsic and intrinsic parameters. At last a dense reconstruction is generated, which approximates the real-world scene depicted by the input images. Since our application requires a reconstruction in geographic ECEF coordinates, an additional georeferencing step converts the local point coordinates into global ECEF coordinates.

To execute the aforementioned subtasks, VisualSFM offers two interfaces that can be accessed in a programmable way. We implemented two steering algorithms, each of them using a different engine interface.

Commandline interface

One way to control the VisualSFM engine programmatically is by using its command line interface. The following listing shows the "generate", the "resume" and the "georeference" commands that are executed via the command line.

```
#generate new reconstruction
VisualSFM sfm+merge+pmvs+k=1024,800,1024,600 ./ ./output.nvm

#resume existing reconstruction
VisualSFM sfm+merge+resume+fixcam+pmvs ./input.nvm ./output.nvm

#georeference reconstruction
VisualSFM sfm+loadnvm+gcp ./input.nvm ./output.nvm
```

Listing 6.3: Commands to execute the VisualSFM engine via the command line.

The first command in the listing generates a new reconstruction. The "merge" keyword forces the SfM-engine to try to merge several smaller sparse reconstructions into a single bigger reconstruction in order to minimize the number of output point clouds. This is done before the dense reconstruction step that is triggered by the "pmvs" parameter. The last "k" attribute submits a fixed camera calibration to the engine. This speeds up the reconstruction process, because the engine doesn't need to recover the camera intrinsics because they are already given as input parameters (cf. Chapter 3 for more details).

The second command resumes an existing reconstruction. Before starting the reconstruction process, the original VisualSFM project file together with the images used in the previous reconstruction process must be exported from the OpenSfM database and saved into the task folder on the server's file system that contains the newly uploaded images. Additionally, a text file named *./input.nvm.txt* must be saved in the same directory containing a list of all input images for the "resume" task. The "resume" keyword forces the SfM-engine to only calculate missing information during the reconstruction process. This means for example, that already calculated feature correspondences are reused and not recalculated. The "fixcam" attribute makes sure that cameras reused from the existing reconstruction stay at the same location as before. This allows us to leave existing camera entries in the database untouched and only insert newly added cameras into the database. However, it is not guaranteed that the newly generated dense reconstruction contains the already stored points of the previous dense reconstruction. Therefore we have to drop the point clouds stored for the involved SfM-dataset and recompute the whole octree-based point cloud representation for the extended dense reconstruction.

The third command in Listing 6.3 georeferences the generated reconstruction. This process converts the local point coordinates into global coordinates in the geographic cartesian ECEF coordinate system. Unfortunately, via the command line interface it is not possible to use the GPS information stored in the EXIF data of the uploaded input images. Therefore, an alternative approach is to upload a ground-control-point-file (gcp-file) that contains at least three SfM-camera locations in geodetic coordinates. This file must be named *./input.gcp*. Since this approach is not very user-friendly, we finally decided to control VisualSFM using its socket interface described in the following section.

Socket interface

An alternative way to communicate with the VisualSFM engine is via its socket interface. As it turned out, this way is the favourable one, because it allows to use the GPS information stored in the input images for georeferencing, which is not possible when using the command line interface.

To use the socket interface to access the VisualSFM application, a VisualSFM instance must listen on a specific port to detect commands sent by our web application. In Section 6.3.3 we already described how a unique port is determined for each VisualSFM task process. The following listing shows the command that is used to start the engine in server mode such that it can handle received commands via the specified socket port.

```
#start VisualSFM in server mode
VisualSFM listen+log <port>
```

Listing 6.4: Command to start the VisualSFM engine in server mode.

However, during implementation we ran into troubles when we tried to start VisualSFM in server mode on a Linux-based operating system. The problem was that VisualSFM opens a GUI, even when it is started in server mode. To display a GUI in a Linux environment, access to the local X-server is needed. This access must be allowed manually on the OpenSfM server by executing the command *xhost localhost* with a logged-in user. As long as this user stays logged in, access to the local X-server is granted. Additionally, the default display must be exported using *export DISPLAY=:0*, because the user executing our web application is not logged in on the server's local terminal.

Our server application communicates with the VisualSFM application in server mode by sending command-IDs via the socket interface. These command-IDs uniquely identify menu-items of the VisualSFM menu. This means, that a command sent to the VisualSFM server emulates a click on a menu-item in the VisualSFM GUI. This concept allows us to access every feature of the VisualSFM application without having any limitations. Therefore, we ultimately decided to use this interface for SfM engine control in our final OpenSfM version. Figure 6.14 shows an example that illustrates the socket requests and responses needed for densely reconstructing a scene out of a number of input images.

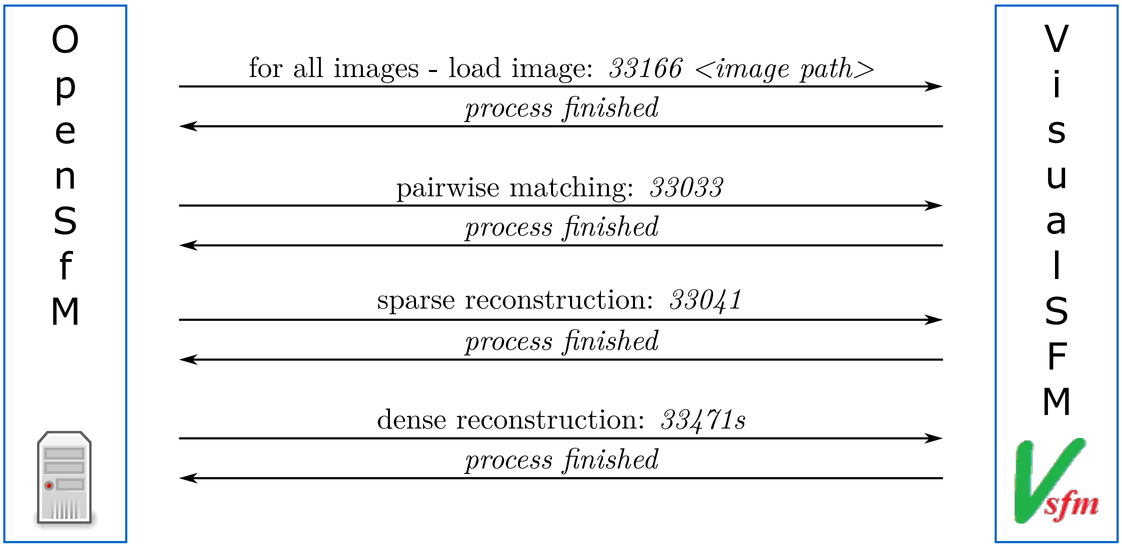

OpenSfM
for all images - load image: *33166 <image path>*
process finished
pairwise matching: *33033*
process finished
sparse reconstruction: *33041*
process finished
dense reconstruction: *33471s*
process finished
VisualSFM
Vsfm

Figure 6.14: Illustration of the socket communication between the OpenSfM server application and the VisualSFM engine used to generate a dense scene reconstruction out of input images. The italic letters denote the values that are sent via socket requests and responses between the two applications. The IDs in these values identify commands that are executed by VisualSFM.

6.3.5 Adapted PotreeConverter

The original PotreeConverter application that comes with the Potree renderer library generates several files in an output folder, each of them representing an octree node and storing its points. Our OpenSfM web application is designed around a database in the persistence layer. Therefore we decided to adapt the original PotreeConverter such that it stores the points of the octree nodes and their hierarchical relationship in database tables (cf. Section 6.4).

We implemented our own converter that creates an octree hierarchy for each dense reconstruction generated by the VisualSFM application based on the implementation of the original PotreeConverter [87]. First of all, the axis-aligned bounding box of the point cloud is calculated. This bounding box is used to compute a spacing value that represents the minimum distance between two points in the octree's root level using Equation 6.3. If the user-defined diagonal fraction value is set to 0, the spacing value set in the front-end's upload mask is used. If this value is also 0, the value 250 is assumed as default diagonal fraction value and the spacing is computed using Equation 6.3.

$$spacing = \frac{bboxDiagonal}{diagFraction} \tag{6.3}$$

After that, each point is added one after another to the octree. Generally, when adding points to a non-leaf node, the above computed spacing that is halved with each octree level must be considered. This means that a point is only added to a non-leaf node if the distance to any other point in the node is larger than the spacing value of the node's octree level. If this is not the case, the point is forwarded to the next octree level. Points added to a leaf node are firstly stored in a point bucket until a bucket limit of 20000 is exceeded. If this happens, the leaf node is converted to a non-leaf node by adding eight child nodes to it and the points contained in its bucket are added to the (now non-leaf) node considering the aforementioned distance test.

Similar to the original PotreeConverter algorithm, we use sparse Cartesian 3D grids to optimize distance tests. Each node contains such a grid with equally sized cells having a side length equal to the spacing at the node's octree level. A point that is going to be inserted is assigned a 3D grid cell according to its position in space. It is just tested against points in the same grid cell and points contained in the neighbouring cells. This is necessary to reduce the number of distance tests and therefore to speed up the conversion process.

After every millionth processed point, the points are saved to a temporary database table that is dynamically generated for the conversion process. Additionally, the current octree node hierarchy is stored in the *octreenodehierarchy* table of the database. At the end of the conversion process, the temporary table is used to group the points together by their unique octree node index in order to generate a patch per octree node. These patches contain the eventual points and are stored in the *octreenode* table of the database. For more details see the next section.

6.4 The OpenSfM database

The database behind the OpenSfM web application is realized using *PostgreSQL* [41]. We decided to use this database due to its opensource status and its ability to efficiently store points with the help of the *pgPointcloud* extension [74].

Storing large point datasets in a database is a challenge in different respects. First of all, each point is usually described by several variables, which need to be accessed in an easy way. OpenSfM uses eight parameters to describe a point. Besides the three-dimensional position and and the four-dimensional RGBA colour parameters, an additional parameter contains the index of the octree-node the point belongs to. Furthermore, a point dataset consists of billions of points. Storing each of these points in a single database table entry would not be very efficient. Retrieving records from tables with a large number of entries is usually not very fast, due to the structure of database storage. Therefore, we group

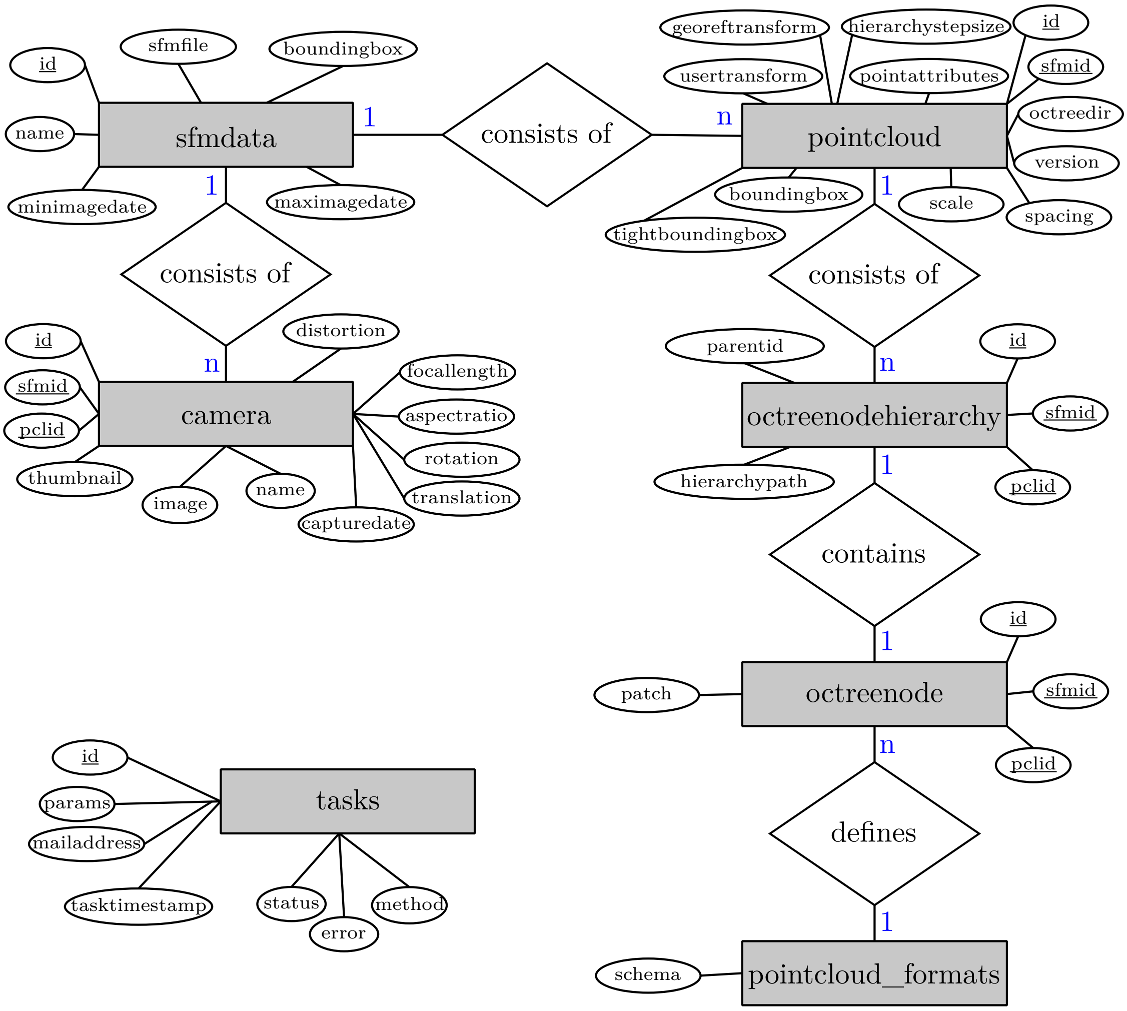


Figure 6.15: Entity-relationship diagram illustrating the OpenSfM database table layout.

the points together into patches and store them as individual entries in the database tables. This grouping step reduces the number of table entries significantly. Since we do not retrieve single points from the database, but whole octree nodes, this is not a limitation. We use a point patch to represent a single octree node containing all the points that belong to this node.

6.4.1 The database table layout

The entity–relationship diagram (ER-diagram) depicted in Figure 6.15 illustrates the table layout and relationships between the tables in the OpenSfM database scheme. The *sfmdata* entity represents a SfM-dataset that consists of 1..n SfM *camera* entities and 1..n *pointcloud* entities. Since each *pointcloud* entity is stored in the octree-based Potree format, each entity has a relationship to its corresponding octree nodes. A *octreenodehierarchy* table stores the octree hierarchy by using a “parentid” column that refers to a node’s parent node. To fulfill the Potree renderer’s loading requests, for each node entry a “hierarchypath” value is stored to identify the requested nodes. Each *octreenodehierarchy* entry refers to a *octreenode* table entry that stores the points of a node using a *PcPatch* typed column. The pgPointcloud extension introduces a PcPatch type that groups points together in a single table entry. The SQL commands in Listing 6.5 are used to generate the entries of the *octreenode* table. A temporary table *temp_points_tempTableId* is used to store the points during the octree generation process. This table is created and dropped during runtime using a *PL/pgSQL* database procedure. To uniquely identify it, its name contains the database sequence value “tempTableId”. At the end of this process, the points stored in the temporary table are grouped by their “OctreeNodeIndex” attribute in order to create the desired patches per octree node.

```sql
WITH toInsert AS (
SELECT PC_patch(point) AS patch, PC_Get(point,'OctreeNodeIndex') AS node
FROM (
SELECT PC_MakePoint(1, ARRAY[x,y,z,r,g,b,a,node]) AS point FROM
    TEMP_POINTS_tempTableId AS pcr
) table_point
GROUP BY PC_Get(point,'OctreeNodeIndex')
);

INSERT INTO octreenode (SELECT node, SFMID, PCLID, patch FROM toInsert);
```

Listing 6.5: SQL commands to group points by their corresponding octree node and insert them into the octreenode table.

When generating a table with a PcPatch typed column, the column type must refer to an entry in the *pointcloud_formats* table that describes the format of a point stored within a patch using a XML schema document. We use a schema that defines a point by its three-dimensional 4-byte unsigned integer position attribute, its four-dimensional 1-byte unsigned char RGBA colour attribute and an additional 1-byte octree-node-index that is used to group the points of the same octree node together into patches. Point clouds usually consist of a large amount of data. To deal with such a large amount within a database, the pgPointcloud extension uses compression algorithms to reduce the required storage size.

An additional *tasks* entity, which is not related to the aforementioned entities, represents a task queue containing jobs that are processed by the back-end. This table is polled by the server to detect new tasks, as already mentioned in Section 6.3.

CHAPTER 7

Results

This chapter shows the results of our implementation by demonstrating the features of the OpenSfM application by means of a workflow example. Additionally, we present some benchmarking results and performance numbers that can be used to prove the usability of our system.

7.1 System

Since the OpenSfM system is based on a client-server architecture, two workstations were needed during the implementation and testing work.

The development workstation was a system composed of an AMD Phenom II X6 1055T processor, 16 GiB RAM and an AMD Radeon R9 270 GPU with 2 GiB GDDR5 video memory that runs a Windows 8.1 x64 operating system. This workstation was used to access the client application during the implementation work and for the performance tests documented below. Furthermore, it hosted the server application during the development process.

The finally released web application is hosted together with the OpenSfM database on a server machine located at the Institute of Computer Graphics at the TU Wien that is running Ubuntu 14.04 x64 on an Intel Core I7-2600K processor, 16 GiB RAM and a CUDA-capable NVIDIA GeForce GTX 580 GPU with 1.5 GiB GDDR5 video memory. The following performance tests are also executed on the release system hosted by this server hardware.

7.2 Performance tests

7.2.1 SfM-datasets

We tested our OpenSfM system with two datasets, each of them generated out of uploaded input images. To get comparable values, the upload time was not considered in the measurements. The smaller “OldMill” dataset consists of 1,587,152 points generated out of 97 of the 101 uploaded images. The georeferenced images were taken using the camera of a Sony Xperia Z1 Compact smartphone. The larger “Favoritenstraße” dataset with 6,468,933 points was reconstructed from 268 of 352 input photographs. The photos of this dataset were taken using a Canon EOS 5D DSLR camera. For georeferencing, an additional gcp-file was uploaded containing GPS coordinates of 4 input photographs. Figure 7.1 depicts five image samples from each of the used photo sequences together with the resulting point cloud renderings.

7.2.2 Dataset generation performance

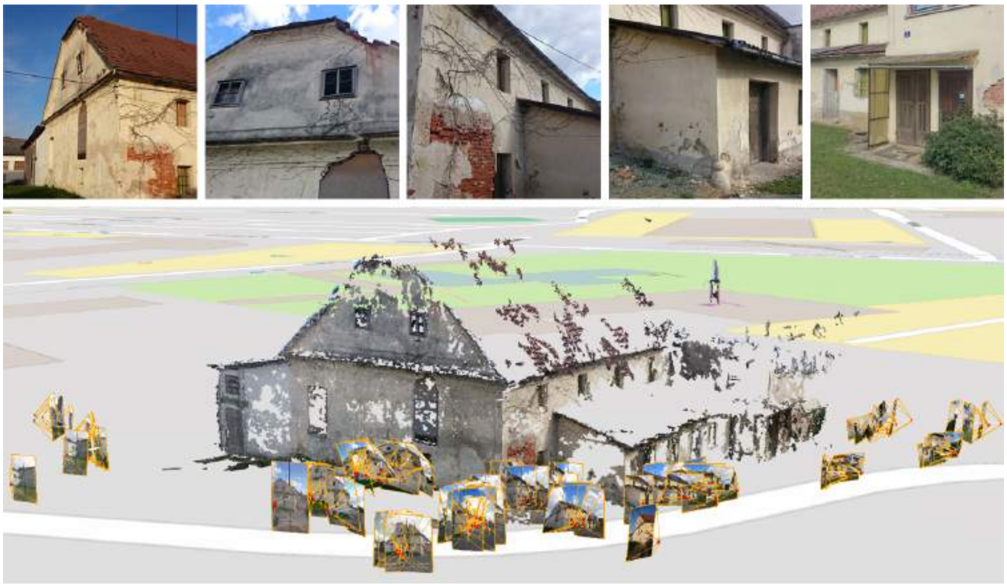
Dataset	#PCLs	spacing	depth	#point	#img	SfM	Octree	Total
OldMill	2	0.072 0.016	5 4	1,382,619 204,533	97	28m	2m	30m
Favoritenstraße	1	0.394	8	6,468,933	268	207m	28m	235m

Table 7.1: Performance values of the SfM-dataset generation process.

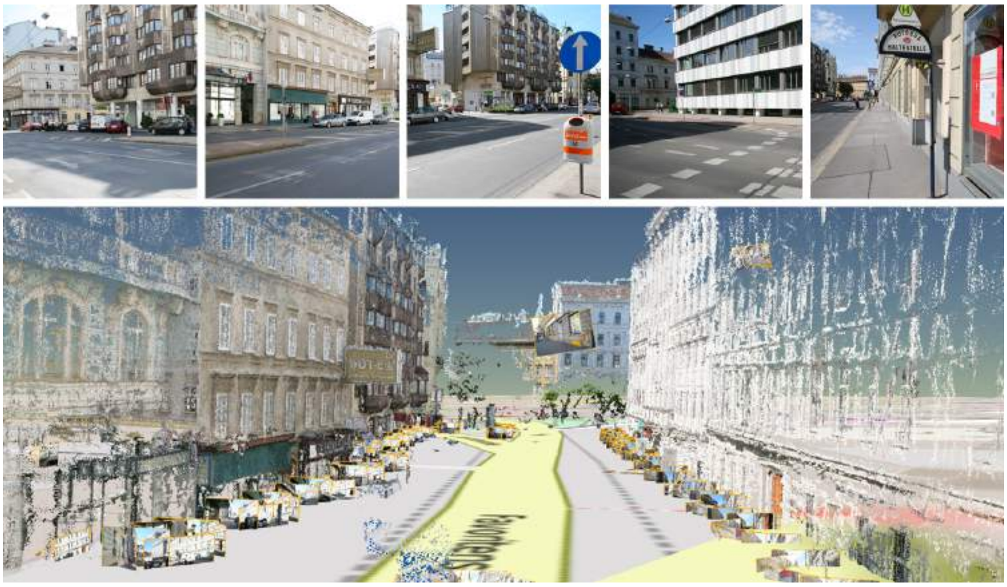
Table 7.1 shows the results of the performance analysis of our back-end SfM-dataset generation process. These values are snapshots that are taken during one single reconstruction iteration and not averaged results of multiple reconstruction runs. The SfM reconstruction process took advantage of the CUDA capability of the server’s GPU and uses it for feature detection and matching.

The “OldMill” dataset consists of two reconstructed point clouds containing in total 1,587,152 points. It took around 30 minutes to generate the 3D reconstruction on the aforementioned server workstation and store it in the OpenSfM database hosted on the same machine. Approximately 28 minutes were needed for the reconstruction process, which uses 97 input images to generate the point cloud representations. Here especially the computation of the dense scene reconstruction using the CMVS algorithm by Furukawa and Ponce (cf. Section 3.1.3) needed a lot of computation time. The remaining processing time was used for generating the octree-based point cloud representation that was stored in the database.

The “Favoritenstraße” dataset was reconstructed out of 268 input images and contains a point cloud with 6,468,933 points. The reconstruction and preprocessing process took approximately 4 hours, where 3.5 hours were used for reconstruction and half an hour for the preprocessing and database storage. It can be seen that the computation time increases with the number of input images and the number of reconstructed points.



(a) “OldMill” dataset: 97 of 101 input images were used to generate two point clouds containing in total 1,587,152 points.



(b) “Favoritenstraße” dataset: 268 of 352 input images were used to generate a point cloud containing 6,468,933 points.

Figure 7.1: Collages of the two datasets used for our performance tests.

7.2.3 Rendering performance

In Table 7.2 the results of our rendering tests are listed. All tests were done using Firefox 45 on the client machine mentioned above. As the results of the generation performance

Dataset	#points	#rendered points	#rendered nodes	fps	fps (cams off)
OldMill	1,587,152	174,819	29	38	54
		1,454,680	209	32	35
Favoritenstraße	6,468,933	350,182	44	24	48
		2,516,114	355	17	20

Table 7.2: Performance values of the SfM rendering process.

tests, the rendering test results are snapshots that represent single measurement values. We measured the frames per second (fps) for two different view points for each of the two examined datasets using adaptive point size and interpolated RGB point colours. The resolution of the rendered canvas was 1680x900.

For the "OldMill" dataset we've chosen the view points such that in one case approximately 175k points in 29 octree nodes and in the other case 1.5M points in 209 nodes were rendered. We measured acceptable 38 respectively 32 frames per second. When hiding the 97 rendered SfM camera markers together with their image planes, the frame rate increased to 54 respectively 35 frames per second.

The "Favoritenstraße" dataset was rendered from a view point far away from the dataset such that all SfM cameras of the large dataset and a sparse octree level were visible. Rendering 350k points in 44 visible octree nodes and 268 camera markers together with their image planes resulted in 24 frames per second. When switching off the camera representations the frame rate doubled to 48 frames per second. When moving closer to the dataset, more octree nodes and points got rendered and the frame rate decreased. We observed a frame rate of 17 frames per second when rendering 2.5M points in 355 nodes. The frame rate increased just slightly to 20 frames per second when switching off the camera illustrations, since from the observed point of view only a small number of SfM cameras were visible and therefore rendered.

7.3 The OpenSfM workflow in practice

This section explains three common use cases for our OpenSfM system accessible via `opensfm.cg.tuwien.ac.at` in more detail.

7.3.1 Explore OpenSfM datasets

The possibly most common use case for our system is simply browsing through SfM-datasets placed on the interactive globe rendered in the browser window. Datasets can either be searched for by manually navigating through the map using panning and zooming or by using the geocoder in the top left corner to search for specific geographic locations.

When the user clicks a red dataset marker in the overview mode, an infobox appears in the bottom-right corner displaying information about the dataset behind the selected marker. However, if more than one dataset is placed at the mouse-click location, a reselection is needed to uniquely identify a dataset. A double-click lets the application load the selected dataset from the database and switches the front-end into viewing mode.

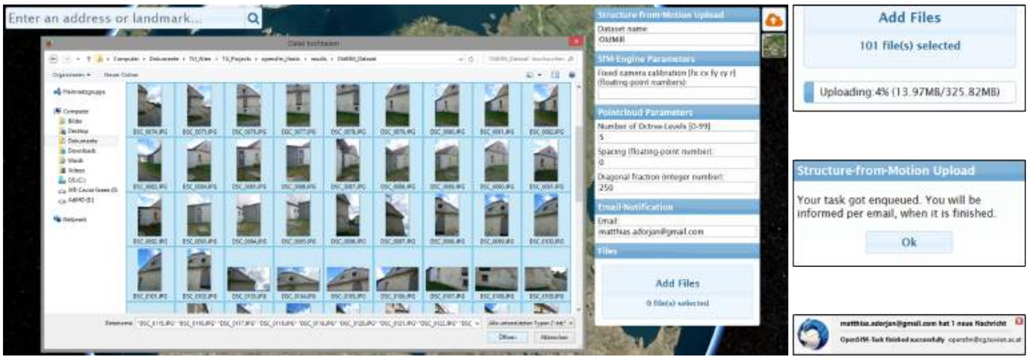
In viewing mode, the SfM-dataset can be explored in more detail. The rendered SfM cameras can be hidden/shown with the help of the small "render" menu visible below the geocoder in the left part of the screen. Double-clicking a rendered SfM camera image plane, the application switches to photo mode and moves the viewing camera in front of the selected plane, such that it is displayed in fullscreen. The thumbnail image is replaced by its corresponding high-quality image. By switching the SfM camera on and off, the user can compare the reconstructed point cloud with the rendered image plane. Another double-click on the photograph returns to view mode, where the user can explore the reconstructed 3D scene with the free-flight viewing camera. He can change the point cloud appearance by choosing between three point-size algorithms. "Fixed" point size renders the points in the size that can be adjusted with the "point size" menu item. "Attenuated" point size rendering uses the value defined by "point size" menu item to specify the point radii in scene coordinates. "Adaptive" point size avoids holes in the scene by adjusting the point sizes according to the point's octree level. Points in higher levels are usually rendered larger than points in lower and denser levels. Additionally, the user can switch between the default RGB material that represents the scene colour derived from the photographs and the tree material that visualizes the rendered octree levels by assigning a different colour to each level. Furthermore, the user can choose from three different rendering qualities. "Square" renders each point as a small square and "Circle" renders small circles. In "Interpolation" mode colours between neighbouring points are interpolated such that especially high frequency point cloud areas where points may overlap are rendered correctly.

The download possibility allows the user to use the VisualSFM project file, the input images and the corresponding point cloud files for further processing tasks independently of our OpenSfM system.

7.3.2 Add new datasets

If a user cannot find a specific 3D reconstruction in our system, but has enough photographs of the original scene, he can upload these photos in order to generate a new reconstruction and add it to our SfM system. In such a way our database can steadily grow and our system can offer free access to a wide variety of different 3D scene reconstructions spread over the whole world. Figure 7.2 depicts the workflow for adding a new SfM-dataset to the OpenSfM system.

To add new SfM-datasets to the system, the user has to complete the SfM upload form accessible via the top-most menu item of the menu bar on the right border of the



Structure-from-Motion Upload

Dataset name: OldMill

SfM-Engine Parameters

Fixed camera calibration [fx cx fy cy r] (floating-point numbers):

Pointcloud Parameters

Number of Octree-Levels [0-99]: 5

Spacing (floating-point number): 0

Diagonal fraction (integer number): 250

Email-Notification

Email: matthias.adorjan@gmail.com

Files

Add Files

0 file(s) selected

Add Files

101 file(s) selected

Uploading:4% (13.97MB/325.82MB)

Structure-from-Motion Upload

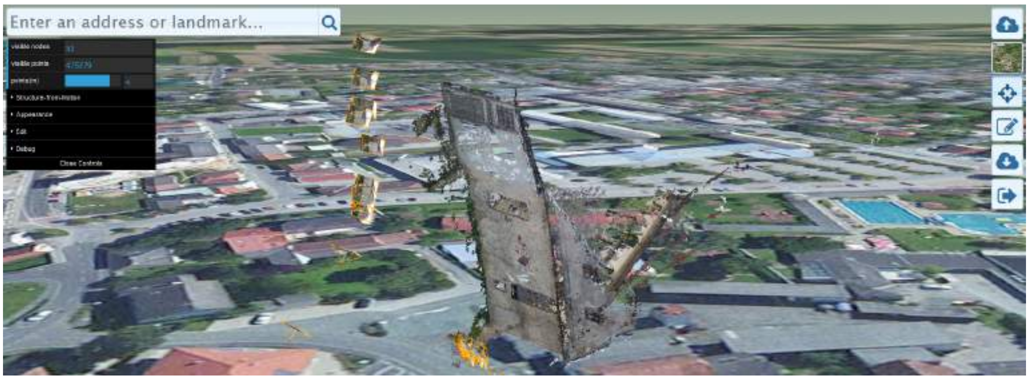
Your task got enqueued. You will be informed per email, when it is finished.

Ok

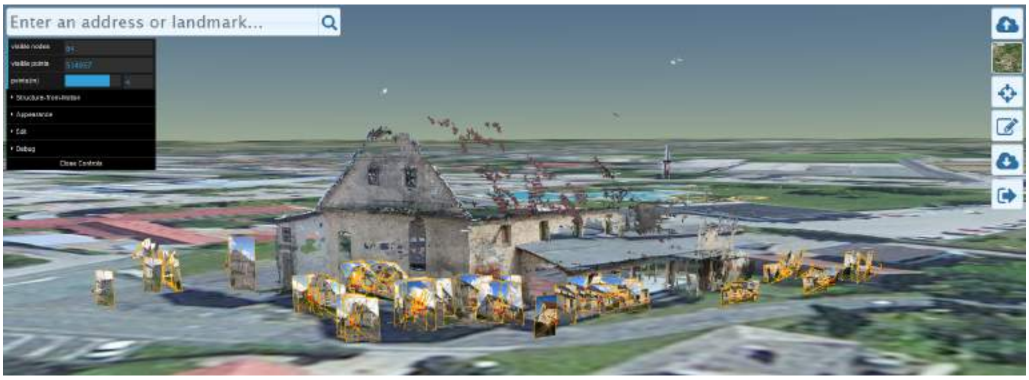
matthias.adorjan@gmail.com hat 1 neue Nachricht

OpenSfM-Task finished successfully openshm@cg.tuwien.ac.at

(a) First, the user selects the input images and fills out the upload form. When submitting the form, the upload process starts. After uploading a popup appears that informs the user that his task got enqueued in the servers job queue. The server application notifies the user per mail about the successful task processing.



(b) The user can return to our application by following the hyperlink in the notification mail. When loading the dataset there can exist georeferencing errors like the one seen in this picture...



(c) ...however, these errors can be easily corrected using the editmode of the OpenSfM front-end.

Figure 7.2: The typical workflow of adding a new dataset to our application.

application. With the help of this form the user can adjust the reconstruction process by defining fixed camera calibration parameters, letting the SfM engine at the background only recover the camera's extrinsics and the 3D reconstruction and by configuring the octree generation process by supplying a maximum octree depth and user defined spacing parameters. The committed mail address is used just for email notification at the end of the reconstruction process. "Add files" opens the file browser that lets the user choose a number of images to upload. Having correctly filled the whole form, a "Submit" button starts the upload process indicated by a progressbar. After the upload has finished, a message informs the user that he gets notified per mail as soon as the reconstruction process has finished.

The mail notification can contain either a success message or an error message. Errors can occur if not enough images are uploaded or if the images are not suitable images for reconstruction. One reason for this is that they do not overlap enough. A success message contains a link to the OpenSfM webpage that lets the user easily return to our system and explore his generated 3D scene reconstruction. However, in some cases it can happen that the georeferencing done using the GPS data stored in the EXIF part of the uploaded images is inaccurate and the resulting dataset is not aligned correctly as can be seen in Figure 7.2b. The edit mode lets the user correct such errors by transforming the point cloud such that it is at last located at its right position on the globe's surface.

7.3.3 Extend existing datasets

Another use case covers the extension of existing datasets. If a user explores a dataset that is not complete, he can add additional images in order to improve the appearance of the 3D reconstruction and add further missing parts to it.

This process can be triggered in the same way as the creation of new datasets. However, to add images to an existing dataset the user must be in the view mode of the targeted scene. Being in view mode, he can open the SfM upload form and complete it in the same way as in the use case described in Section 7.3.2. The reconstruction process at the back-end now triggers a "resume" command that adds the images to the existing dataset. The user gets again informed per mail when the "resume" process has finished and he can return to the OpenSfM webpage to explore the improved SfM-dataset.

7.4 Limitations

Since our application at its current state is just the result of a proof of concept, there exist several limitations. The main issues are described in the following subsections.

7.4.1 Rendering

We have shown that it is possible to combine two independent WebGL based rendering engines such that the produced output can be explored in an interactive way. However, when implementing our front-end we have not attached great importance to rendering performance, as can be seen in Section 7.2.3. Especially rendering the camera illustrations and their image planes is a heavy weight on the performance, because we compute them using an individual camera projection matrix for each camera that is multiplied in each frame.

Additionally, the integration of the ThreeJS-based rendering engine into the CesiumJS rendering pipeline limits the rendering possiblities of the Potree point cloud renderer. Algorithms that require multiple render passes, like high-quality splatting, cannot be used in the current version of OpenSfM. Furthermore, newer Potree features like Eye-Dome-Lighting (EDL) are not tested and implemented in the OpenSfM front-end.

7.4.2 Point cloud processing

The PotreeConverter was successfully adapted to the new database persistence layer. The processing performance is acceptable, however, we haven't optimized it for benchmarking and testing against the original converter. A drawback of this point cloud preprocessing is that the whole octree needs to be recomputed if an existing SfM-dataset is extended. This is required because not all points of the original SfM-dataset are contained in the extended dataset. The recomputation process needs additional computation time and is not always necessary, especially when the extended point cloud is similar to the original one.

7.4.3 Stability and flexibility

The VisualSFM application used by our back-end is not always stable and can crash unpredictably, especially when a lot of input images are used. It can occur that the user waits several hours for a success result from our back-end and finally receives an error message because of an unsuccessful CMVS processing step. This can be very annoying, because our system doesn't offer a partial rerun of the reconstruction process, which means that all images have to be uploaded again and an additional full reconstruction process has to be started in the hope for a successful outcome. Sometimes it can even happen that the SfM engine crashes silently such that our back-end is not informed about the fail. In such cases the back-end process waits for a response until an administrator manually kills the VisualSFM or CMVS process on the server machine.

7.4.4 Georeferencing and edit mode

Georeferencing 3D reconstructions by using the GPS data stored in the input images is not very accurate. Due to this inaccuracy and missing GPS information, the georeferencing algorithm implemented in VisualSFM and used by our application can produce completely wrong results as can be seen in Figure 7.2b. With the help of the edit mode the user can correct these misalignments to a certain degree. Highly accurate georeferencing and user-defined aligning as it is needed for geodesy and surveying is not possible with our implementation. However, for our intended purpose as a virtual tourism application for point cloud exploration the accuracy is sufficient.

CHAPTER 8

Conclusion

We introduced a structure-from-motion system based on the idea of collaborative projects like OpenStreetMap. The client-server system called OpenSfM consists of a freely accessible front-end that acts as a virtual tourism client allowing the exploration and editing of SfM-datasets and a back-end that communicates with the front-end and processes uploaded data that is finally stored in the underlying database.

The freely accessible virtual tourism client offers four different system modes. The overview mode lets the user navigate on an interactive globe, rendered with the help of the CesiumJS library, on which the SfM-datasets are placed either by manually zooming and panning using mouse interactions or by jumping to locations on the globe using a geocoder that translates postal addresses into geographic coordinates. Double-clicking a dataset marker placed on the globe's surface enters the system's view mode. This mode allows the exploration of the georeferenced SfM-dataset consisting of the 3D scene reconstruction represented as point clouds rendered with the help of the Potree library and corresponding SfM camera positions, orientations and photos. Selecting one of the rendered photographs lets the front-end switch to the photo mode, which displays a high-quality version of the image in fullscreen mode, which allows a comparison of the photo and the reconstructed point representation. Furthermore, an edit mode allows the correction of georeferencing and reconstruction errors.

Besides the exploration features, our front-end allows the upload of images into the system. These images are used by the back-end to either generate new SfM-datasets, when done in overview mode, or extend existing SfM-datasets when uploading in the view mode of the corresponding dataset. For the generation of georeferenced, dense scene reconstructions, the state-of-the-art SfM engine VisualSFM is used together with the clustered multi-view stereo algorithm CMVS. The reconstruction and georeferencing process is controlled via socket communication with the VisualSFM application, which runs on the same server machine as our back-end application. VisualSFM georeferences the reconstructed point clouds using the GPS data stored in the uploaded input images. The generated point datasets are preprocessed and converted into an octree-based representation that can be interpreted by the Potree point cloud renderer at the front end.

In summary, our system allows the gathering of SfM-datasets that represent different sights or landmarks, but also just locally famous buildings, placed all over the world by combining different state-of-the-art techniques at the persistence, processing and visualization layer. Those datasets can be explored in an interactive way by every user who accesses our virtual tourism client using a web browser. Furthermore, the download possibility allows any user to save the SfM-datasets and point clouds on his local file system for subsequent offline processing.

8.1 Future work

Our system shows that it is possible to establish a user-friendly, free and fully accessible structure-from-motion platform that anyone with access to the Internet can use to reconstruct large 3D scenes without any technical background knowledge and share the reconstructions with other OpenSfM users. However, integrating complex algorithms into a user-friendly system comes with several limitations as mentioned in Section 7.4. Therefore one of the tasks of the future work on our system will be to overcome those limitations.

Rendering-specific shortcomings like unsupported high-quality point splatting and missing Potree features like Eye-Dome-Lighting (EDL) should be tackled first. However, this task could be hard and may require a lot of implementation work because our system ultimately uses two nearly independent renderer implementations that maintain their own rendering states, which is not very helpful when realizing special rendering effects.

Furthermore, the integration of the SfM engine at the back-end can be more flexible. Instead of restarting the whole reconstruction process in error cases, rerunning just individual steps of the reconstruction pipeline would significantly increase the usability of our system. Additionally, the SfM-data extension algorithm can be improved by implementing a heuristic that punctually adds points to the existing octree-based representation that fill missing areas in the point clouds instead of completely dropping existing reconstructions and replacing them with the newly generated ones.

Improving the accuracy of georeferencing would be another important task for future work on our system. This improvement could be achieved by using additional control points that refer to visible points in the input images instead of inaccurate camera locations stored in the EXIF tags of the input images. VisualSFM supports such control points as input gcp-files. However, the manual generation of such gcp-files is not very user-friendly. Therefore, an interface to pick the points in the uploaded images at our front-end would be desirable.

An additional useful feature would be morphing between photographs in photo mode, like realized in the Photo Tourism application [97]. Also merging the overview mode with the view mode would be another desirable improvement. Then instead of rendering single

SfM-datasets in view mode, SfM-datasets would be rendered depending on the camera's view point and direction directly in overview mode allowing to navigate seamlessly between neighbouring datasets.

Bibliography

[1] Sameer Agarwal, Yasutaka Furukawa, Noah Snavely, Ian Simon, Brian Curless, Steven M Seitz, and Richard Szeliski. Building rome in a day. *Communications of the ACM*, 54(10):105–112, 2011.

[2] Sameer Agarwal, Noah Snavely, Steven M Seitz, and Richard Szeliski. Bundle adjustment in the large. In *Computer Vision–ECCV 2010*, pages 29–42. Springer, 2010.

[3] Analytical Graphics Inc. (AGI) and open-source community. CesiumJS. URL: `http://cesiumjs.org/` (visited on March 22, 2016), 2016.

[4] DP Andrews, J Bedford, and PG Bryan. a Comparison of Laser Scanning and Structure from Motion as Applied to the Great Barn at Harmondsworth, UK. *ISPRS-International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 1(2):31–36, 2013.

[5] Jamal Jokar Arsanjani, Alexander Zipf, Peter Mooney, and Marco Helbich. An Introduction to OpenStreetMap in Geographic Information Science: Experiences, Research, and Applications. In *OpenStreetMap in GIScience*, pages 1–15. Springer, 2015.

[6] Sunil Arya, David M Mount, Nathan S Netanyahu, Ruth Silverman, and Angela Y Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.

[7] Johannes Bauer, Niko Sunderhauf, and Peter Protzel. Comparing several implementations of two recently published feature detectors. In *Proc. of the International Conference on Intelligent and Autonomous Systems*, volume 6, 2007.

[8] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *Computer vision–ECCV 2006*, pages 404–417. Springer, 2006.

[9] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173–189, 1972.

[10] Yochai Benkler. Freedom in the commons: Towards a political economy of information. *Duke Law Journal*, pages 1245–1276, 2003.

[11] Wasim Ahmad Bhat and SMK Quadri. A Quick Review of On-Disk Layout of Some Popular Disk File Systems. *Global Journal of Computer Science and Technology*, 11(6), 2011.

[12] François Blais. Review of 20 years of range sensor development. *Journal of Electronic Imaging*, 13(1), 2004.

[13] Blizzard. World of Warcraft subscriber base reaches 12 million worldwide. URL: `http://us.blizzard.com/en-us/company/press/pressreleases.html?id=2847881` (visited on December 11, 2015), 2010.

[14] Matthew Brown and David G Lowe. Automatic panoramic image stitching using invariant features. *International journal of computer vision*, 74(1):59–73, 2007.

[15] Martin Byröd and Karl Åström. Bundle adjustment using conjugate gradients with multiscale preconditioning. In *British Machine Vision Conference*, 2009.

[16] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: Binary robust independent elementary features. *Computer Vision–ECCV 2010*, pages 778–792, 2010.

[17] Błażej Ciepłuch, Ricky Jacob, Peter Mooney, and Adam Winstanley. Comparison of the accuracy of OpenStreetMap for Ireland with Google Maps and Bing Maps. In *Proceedings of the Ninth International Symposium on Spatial Accuracy Assessment in Natural Resuorces and Enviromental Sciences 20-23rd July 2010*, page 337. University of Leicester, 2010.

[18] Douglas Comer. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[19] Patrick Cozzi. Graphics Tech in Cesium - Renderer Architecture. URL: `http://cesiumjs.org/2015/05/15/Graphics-Tech-in-Cesium-Architecture/` (visited on March 19, 2016), 2015.

[20] Patrick Cozzi. Graphics Tech in Cesium - Rendering a Frame. URL: `http://cesiumjs.org/2015/05/14/Graphics-Tech-in-Cesium/` (visited on March 19, 2016), 2015.

[21] Patrick Cozzi and Kevin Ring. *3D engine design for virtual globes*. CRC Press, 2011.

[22] Frédéric Devernay and Olivier D. Faugeras. Automatic calibration and removal of distortion from scenes of structured environments. In *SPIE's 1995 International Symposium on Optical Science, Engineering, and Instrumentation*, pages 62–72. International Society for Optics and Photonics, 1995.

[23] C. Brown Duane. Close-range camera calibration. *Photogram. Eng. Remote Sens*, 37:855–866, 1971.

[24] Jakob Engel, Thomas Schöps, and Daniel Cremers. LSD-SLAM: Large-scale direct monocular SLAM. In *Computer Vision–ECCV 2014*, pages 834–849. Springer, 2014.

[25] Remondino Fabio. From point cloud to surface: the modeling and visualization problem. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 34(5):W10, 2003.

[26] Facebook. Facebook Reports Fourth Quarter and Full Year 2014 Results. URL: `http://investor.fb.com/releasedetail.cfm?ReleaseID=893395` (visited on December 11, 2015), 2014.

[27] Olivier D. Faugeras. What can be seen in three dimensions with an uncalibrated stereo rig? In *Computer Vision—ECCV'92*, pages 563–578. Springer, 1992.

[28] Olivier D. Faugeras, Q-T. Luong, and Stephen J. Maybank. Camera self-calibration: Theory and experiments. In *Computer Vision—ECCV'92*, pages 321–334. Springer, 1992.

[29] Raphael A. Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[30] Andrea Forte and Amy Bruckman. Why do people write for Wikipedia? Incentives to contribute to open–content publishing. In *Proceedings of 41st Annual Hawaii International Conference on System Sciences (HICSS)*, pages 1–11, 2008.

[31] Apache Software Foundation. Apache Maven. URL: `https://maven.apache.org/` (visited on March 22, 2016), 2016.

[32] Witold Fraczek. Mean sea level, GPS, and the geoid. *ArcUsers Online*, 2003.

[33] Yasutaka Furukawa, Brian Curless, Steven M Seitz, and Richard Szeliski. Towards internet-scale multi-view stereo. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 1434–1441. IEEE, 2010.

[34] Yasutaka Furukawa and Jean Ponce. Accurate, dense, and robust multiview stereopsis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(8):1362–1376, 2010.

[35] Riccardo Gherardi, Michela Farenzena, and Andrea Fusiello. Improving the efficiency of hierarchical structure-and-motion. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 1594–1600, 2010.

[36] Enrico Gobbetti and Fabio Marton. Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics*, 28(6):815–826, 2004.

[37] Michael Goesele, Noah Snavely, Brian Curless, Hugues Hoppe, and Steven M Seitz. Multi-view stereo for community photo collections. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8. IEEE, 2007.

[38] Michael F Goodchild. Citizens as sensors: the world of volunteered geography. *GeoJournal*, 69(4):211–221, 2007.

[39] Mark Graham. Cloud collaboration: Peer-production and the engineering of the internet. In *Engineering earth*, pages 67–83. Springer, 2011.

[40] Khronos Group. WebGL. URL: `https://www.khronos.org/webgl/` (visited on March 22, 2016), 2016.

[41] The PostgreSQL Global Development Group. PostgreSQL. URL: `http://www.postgresql.org/` (visited on March 21, 2016), 2016.

[42] GulpJS. GulpJS. URL: `http://gulpjs.com/` (visited on March 22, 2016), 2016.

[43] Antonin Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD Rec.*, 14(2):47–57, 1984.

[44] Mordechai Haklay et al. How good is volunteered geographical information? A comparative study of OpenStreetMap and Ordnance Survey datasets. *Environment and planning. B, Planning & design*, 37(4):682, 2010.

[45] Mordechai Haklay and Patrick Weber. Openstreetmap: User-generated street maps. *Pervasive Computing, IEEE*, 7(4):12–18, 2008.

[46] Klaus Häming and Gabriele Peters. The structure-from-motion reconstruction pipeline - a survey with focus on short image sequences. *Kybernetika*, 46(5):926–937, 2010.

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

[48] Richard Hartley et al. In defense of the eight-point algorithm. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 19(6):580–593, 1997.

[49] Richard Hartley, Rajiv Gupta, and Tom Chang. Stereo from uncalibrated cameras. In *Computer Vision and Pattern Recognition, 1992. Proceedings CVPR'92., 1992 IEEE Computer Society Conference on*, pages 761–764. IEEE, 1992.

[50] Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision.* Cambridge university press, 2003.

[51] Richard I Hartley. Estimation of relative camera positions for uncalibrated cameras. In *Computer Vision—ECCV'92*, pages 579–587. Springer, 1992.

[52] Benjamin Mako Hill. *Essays on volunteer mobilization in peer production.* PhD thesis, Massachusetts Institute of Technology, 2013.

[53] National Imagery and Mapping Agency. Department of Defense World Geodetic System 1984: Its Definition and Relationships With Local Geodetic Systems, Third edition. Technical report, National Imagery and Mapping Agency, 2000.

[54] Instagram. Celebrating a community of 400 million. URL: `http://blog.instagram.com/post/129662501137/150922-400million` (visited on December 13, 2015), 2015.

[55] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Proceedings of the fourth Eurographics symposium on Geometry processing*, volume 7, 2006.

[56] Nabeel Younus Khan, Brendan McCane, and Geoff Wyvill. SIFT and SURF performance evaluation against various image deformations on benchmark dataset. In *Digital Image Computing Techniques and Applications (DICTA), 2011 International Conference on*, pages 501–506. IEEE, 2011.

[57] Donald E Knuth. Big omicron and big omega and big theta. *ACM Sigact News*, 8(2):18–24, 1976.

[58] Alfred Leick, Lev Rapoport, and Dmitry Tatarnikov. *GPS satellite surveying.* John Wiley & Sons, 2015.

[59] Josh Lerner and Jean Triole. The simple economics of open source. Technical Report 2, National Bureau of Economic Research, 2002.

[60] Bo Leuf and Ward Cunningham. *The Wiki Way: Quick Collaboration on the Web.* Addison-Wesley Professional, 2001.

[61] H.C. Longuet Higgins. A Computer Algorithm for Reconstructing a Scene from Two Projections. *Nature*, 293:133–135, 1981.

[62] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[63] Stephen Malys, John H Seago, Nikolaos K Pavlis, P Kenneth Seidelmann, and George H Kaplan. Why the Greenwich meridian moved. *Journal of Geodesy*, 89(12):1263–1272, 2015.

[64] Dennis D McCarthy. IERS technical note 21. *US Naval Observatory*, 1996.

[65] Donald JR Meagher. *Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer.* Electrical and Systems Engineering Department Rensseiaer Polytechnic Institute Image Processing Laboratory, 1980.

[66] Microsoft. Azure. URL: `https://azure.microsoft.com/de-de/` (visited on January 7, 2016).

[67] Microsoft. Silverlight. URL: `http://www.microsoft.com/silverlight/` (visited on January 7, 2016).

[68] Michael Montemerlo, Sebastian Thrun, Daphne Koller, Ben Wegbreit, et al. FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *AAAI/IAAI*, pages 593–598, 2002.

[69] MrDoob. ThreeJS. URL: `http://threejs.org/` (visited on March 22, 2016), 2016.

[70] Netcraft. October 2015 Web Server Survey. URL: `http://news.netcraft.com/archives/2015/10/16/october-2015-web-server-survey.html` (visited on December 9, 2015), 2015.

[71] Andreas Nuchter, Hartmut Surmann, Kai Lingemann, Joachim Hertzberg, and Sebastian Thrun. 6D SLAM with an application in autonomous mine mapping. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 2, pages 1998–2003. IEEE, 2004.

[72] OpenStreetMap. OpenStreetMap. URL: `http://www.openstreetmap.org` (visited on November 30, 2015).

[73] OpenStreetMap. OpenStreetMap Wiki. URL: `http://wiki.openstreetmap.org` (visited on December 13, 2015).

[74] pgPointcloud. pgPointcloud. URL: `https://github.com/pgpointcloud/pointcloud` (visited on March 21, 2016), 2016.

[75] Frederik Ramm and Jochen Topf. *OpenStreetMap: Die freie Weltkarte nutzen und mitgestalten.* Lehmanns Media, 2010.

[76] Richard H Rapp. Geometric geodesy: Part I. *Lecture Notes*, 1991.

[77] Bernhard Reitinger, Christopher Zach, and Dieter Schmalstieg. Augmented reality scouting for interactive 3d reconstruction. In *Virtual Reality Conference, 2007. VR'07. IEEE*, pages 219–222. IEEE, 2007.

[78] Microsoft Research. Photosynth. URL: `http://photosynth.net/` (visited on January 6, 2016).

[79] Microsoft Research. Spin. URL: `http://research.microsoft.com/en-us/projects/spin/default.aspx` (visited on January 6, 2016).

[80] George Ritzer and Nathan Jurgenson. Production, Consumption, Prosumption The nature of capitalism in the age of the digital 'prosumer'. *Journal of consumer culture*, 10(1):13–36, 2010.

[81] D.P. Robertson and R. Cipolla. Structure from Motion. URL: `http://mi.eng.cam.ac.uk/~cipolla/publications/contributionToEditedBook/2008-SFM-chapters.pdf` (visited on November 30, 2015), 2008.

[82] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *Computer Vision–ECCV 2006*, pages 430–443. Springer, 2006.

[83] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. ORB: an efficient alternative to SIFT or SURF. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2564–2571. IEEE, 2011.

[84] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 343–352. ACM Press/Addison-Wesley Publishing Co., 2000.

[85] Hanan Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[86] Claus Scheiblauer. *Interactions with Gigantic Point Clouds*. PhD thesis, Vienna University of Technology, 7 2014.

[87] Markus Schütz. Potree. URL: `http://potree.org/` (visited on February 2, 2016).

[88] Markus Schütz and M Wimmer. Rendering Large Point Clouds in Web Browsers. *Proceedings of CESCG*, pages 83–90, 2015.

[89] Graham Sellers, Juraj Obert, Patrick Cozzi, Kevin Ring, Emil Persson, Joel de Vahl, and J. M. P. van Waveren. Rendering Massive Virtual Worlds. In *ACM SIGGRAPH 2013 Courses*, SIGGRAPH '13, 2013.

[90] Sudipta N Sinha, Johannes Kopf, Michael Goesele, Daniel Scharstein, and Richard Szeliski. Image-based rendering for scenes with reflections. *ACM Trans. Graph.*, 31(4):100, 2012.

[91] Sudipta N Sinha, Drew Steedly, and Richard Szeliski. Piecewise planar stereo for image-based rendering. In *ICCV*, pages 1881–1888, 2009.

[92] Sudipta N Sinha, Drew Steedly, and Richard Szeliski. A multi-stage linear approach to structure from motion. In *Trends and Topics in Computer Vision*, pages 267–281. Springer, 2012.

[93] Chester C Slama, Charles Theurer, Soren W Henriksen, et al. *Manual of photogrammetry*, volume 4. American Society of photogrammetry, 1980.

[94] N. Snavely, S. M. Seitz, and R. Szeliski. Modeling the World from Internet Photo Collections. *International Journal of Computer Vision*, 80(2):189–210, November 2008.

[95] Noah Snavely. Bundler. URL: `http://www.cs.cornell.edu/~snavely/bundler/` (visited on January 6, 2016).

[96] Noah Snavely, Rahul Garg, Steven M Seitz, and Richard Szeliski. Finding paths through the world's photos. In *ACM Transactions on Graphics (TOG)*, volume 27, page 15. ACM, 2008.

[97] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo tourism: Exploring photo collections in 3D. In *SIGGRAPH Conference Proceedings*, pages 835–846, New York, NY, USA, 2006. ACM Press.

[98] Noah Snavely, Steven M Seitz, and Richard Szeliski. Skeletal graphs for efficient structure from motion. In *CVPR*, volume 1, page 2, 2008.

[99] Noah Snavely, Ian Simon, Michael Goesele, Richard Szeliski, and Steven M Seitz. Scene reconstruction and visualization from community photo collections. *Proceedings of the IEEE*, 98(8):1370–1390, 2010.

[100] J. Sanz Subirana, J.M. Juan Zornoza, and M. Hernández-Pajares. Transformations between ECEF and ENU coordinates. URL: `http://www.navipedia.net/index.php/Transformations_between_ECEF_and_ENU_coordinates` (visited on February 15, 2016). Technical University of Catalonia, Spain.

[101] 3D Systems. 3D Scanners: A guide to 3D scanner technology. URL: `http://www.rapidform.com/3d-scanners/` (visited on November 24, 2015).

[102] Richard Szeliski. *Computer vision: Algorithms and applications*. Springer Science & Business Media, 2010.

[103] The Spring Team. Spring Framework. URL: `http://spring.io` (visited on March 22, 2016), 2016.

[104] Tubefilter. YouTube now gets over 400 hours of content uploaded every minute. URL: `http://www.tubefilter.com/2015/07/26/youtube-400-hours-content-every-minute/` (visited on December 13, 2015), 2015.

[105] Kathleen Tuite, Noah Snavely, Dun-yu Hsiao, Nadine Tabing, and Zoran Popovic. PhotoCity: Training experts at large-scale image acquisition through a competitive game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1383–1392. ACM, 2011.

[106] Government U.S. GPS: Accuracy. URL: `http://www.gps.gov/systems/gps/performance/accuracy/` (visited on December 13, 2015).

[107] Government U.S. GPS: Space segment. URL: `http://www.gps.gov/systems/gps/space/` (visited on December 13, 2015).

[108] Jakob Voss. Measuring wikipedia. In *Proceedings of the ISSI 2005*, 2005.

[109] Hoang-Hiep Vu, Patrick Labatut, Jean-Philippe Pons, and Renaud Keriven. High accuracy and visibility-consistent dense multiview stereo. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34(5):889–901, 2012.

[110] Likun Wang, Denis Tremblay, Bin Zhang, and Yong Han. Fast and Accurate Collocation of the Visible Infrared Imaging Radiometer Suite Measurements with Cross-Track Infrared Sounder. *Remote Sensing*, 8(1):76, 2016.

[111] MJ Westoby, J Brasington, NF Glasser, MJ Hambrey, and JM Reynolds. 'Structure-from-Motion'photogrammetry: A low-cost, effective tool for geoscience applications. *Geomorphology*, 179:300–314, 2012.

[112] Wikimedia. Report Card August 2015. URL: `reportcard.wmflabs.org/` (visited on December 16, 2015), 2015.

[113] Wikimedia. Wikipedia Statistics, October 2015. URL: `https://stats.wikimedia.org/EN/Sitemap.htm` (visited on December 13, 2015), 2015.

[114] Wikimedia. Wikipedia:About. URL: `https://en.wikipedia.org/wiki/Wikipedia:About` (visited on January 3, 2016), 2016.

[115] Wikimedia. Wikipedia:Five pillars. URL: `https://en.wikipedia.org/wiki/Wikipedia:Five_pillars` (visited on January 3, 2016), 2016.

[116] Wikipedia. Wikipedia. URL: `http://en.wikipedia.org` (visited on November 30, 2015).

[117] Michael Wimmer and Claus Scheiblauer. Instant Points: Fast Rendering of Unprocessed Point Clouds. In *SPBG*, pages 129–136. Citeseer, 2006.

[118] Stephen J Wright and Jorge Nocedal. *Numerical optimization*, volume 2. Springer New York, 1999.

[119] Changchang Wu. SiftGPU: A GPU implementation of scale invariant feature transform (SIFT). URL: `http://cs.unc.edu/~ccwu/siftgpu/` (visited on January 20, 2016), 2007.

[120] Changchang Wu. VisualSFM: A visual structure from motion system. URL: `http://ccwu.me/vsfm/` (visited on January 20, 2016), 2011.

[121] Markus Ylimäki, Juho Kannala, Jukka Holappa, Sami S Brandt, and Janne Heikkilä. Fast and accurate multi-view reconstruction by multi-stage prioritised matching. *IET Computer Vision*, 2015.

[122] Zhengyou Zhang. A flexible new technique for camera calibration. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(11):1330–1334, 2000.