Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Coloring Meshes of Archaeological Datasets

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik/Digitale Bildverarbeitung

eingereicht von

Michael Birsak
Matrikelnummer 0525386

an der
Fakultät für Informatik der Technischen Universität Wien

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

Wien, 10.05.2012

(Unterschrift Verfasser)

(Unterschrift Betreuung)

Coloring Meshes of Archaeological Datasets

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Michael Birsak
Registration Number 0525386

to the Faculty of Informatics
at the Vienna University of Technology

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

Vienna, 10.05.2012

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Michael Birsak
Wallensteinstraße 3/8
1200 Wien

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.

(Ort, Datum) (Unterschrift Verfasser)

Abstract

Archaeological monuments are nowadays preserved for future generations by means of digitization. To this end, laser scanners in conjunction with registered cameras are used to gather both the geometric and the color information. The geometry is often triangulated to allow a continuous mapping of the photos onto the geometry. The color mapping for high-quality reconstructions of parts of the models is not without problems. In practice, the photos overlap. Now, assuming that a particular triangle receives color information from just one photo, there is a huge number of possibilities to map the photos onto the triangles. This labeling problem is already covered in literature. There are also approaches for the leveling of the remaining seams that arise because of the different lighting situations during the exposure of the photos.

In this thesis, we improve common labeling approaches by the introduction of an accurate geometry-based occlusion detection. An octree is used to quickly filter out parts of the model that do not come into consideration for an occlusion anyway. The occlusion detection prevents texturing of parts of the model with image material that does not contain the expected region, but the colors of an occluder.

Further, a proposed approach for seam leveling on meshes is improved by the introduction of a new term into the least squares problem that corresponds to the proposed leveling approach. This new term penalizes big leveling function values and helps to keep the leveled color values in the valid range. For better filtering results, we improve the proposed calculation of a 1-pixel wide boundary around the leveled patches by the introduction of outline normals for a user-defined scale of the patches.

For easier manual editing of remaining artifacts in the photos, we introduce an application for the generation of alpha masks that indicate regions of the photos that are used for texturing of the 3D model.

For the high-performance visualization of 3D models with a huge amount of textures, we make use of virtual texturing. We present an application that generates the needed data structures *atlas* and *tile store* in significantly less time than existing scripts.

Finally, we show how all the mentioned functionalities are integrated into a visualization application that can support a graphic artist in the post-processing of a digitized archaeological monument.

Kurzfassung

Archäologische Monumente werden heutzutage mittels Digitalisierung für zukünftige Generationen aufbewahrt. Hierzu werden Laserscanner in Verbindung mit registrierten Kameras eingesetzt, um sowohl die Geometrie als auch die Farbinformation aufzunehmen. Die Geometrie wird oft trianguliert, sodass eine kontinuierliche Abbildung der Fotos ermöglicht wird. Die Abbildung der Fotos zur qualitativ hochwertigen Rekonstruktion von Teilen der Modelle ist nicht unproblematisch. In der Praxis überlappen sich die Fotos. Setzt man nun voraus, dass ein Dreieck des Modells die Farbinformation von nur einem Foto erhält, so gibt es eine riesige Anzahl an Möglichkeiten, die Fotos auf die Dreiecke abzubilden. Dieses Labeling-Problem wird bereits in der Literatur behandelt. Es existieren auch bereits Methoden um jene Bereiche, die von unterschiedlichen Fotos texturiert werden, farblich an einander anzupassen. Diese Stoßkanten entstehen durch unterschiedliche Beleuchtungssituationen während der Aufnahme der Fotos.

In dieser Diplomarbeit verbessern wir übliche Labeling-Methoden durch die Einführung einer akkuraten geometriebasierten Verdeckungserkennung. Ein Octree wird verwendet um schnell jene Teile des Modells auszufiltern, welche ohnehin nicht für eine Verdeckung in Frage kommen. Die Verdeckungserkennung verhindert, dass Teile des Modells Bildmaterial von Fotos erhalten, welche nicht die erwartete Region, sondern die Farben eines verdeckenden Objekts enthalten.

Weiters wird eine vorgeschlagene Methode zur farblichen Anpassung der Stoßkanten für Dreiecksnetze verbessert, indem ein neuer Term in das Kleinstquadrat-Problem hinzugefügt wird, welches zu der vorgeschlagenen Methode korrespondiert. Dieser neue Term benachteiligt große Funktionswerte der Leveling-Funktion und hilft, dass die angepassten Farbwerte nicht den gültigen Bereich verlassen. Für bessere Filterungsresultate verbessern wir die vorgeschlagene Berechnung eines ein Pixel breiten Randes um die farblich angepassten Bildregionen, indem wir die Normalvektoren für den Rand der Regionen berechnen, um so eine beliebige Skalierung der Regionen zu ermöglichen.

Zur einfacheren manuellen Bearbeitung verbleibender Artefakte in den Fotos stellen wir eine Applikation zur Generierung von Alpha-Masken vor. Diese Masken zeigen jene Regionen der Fotos, welche zur Texturierung des 3D-Modells verwendet werden.

Zur Visualisierung von 3D-Modellen mit großen Texturmengen verwenden wir Virtual Texturing. Wir präsentieren eine Applikation, welche die notwendigen Datenstrukturen *Atlas* und *Tile Store* in deutlich kürzerer Zeit als existierende Scripts generiert.

Zu guter Letzt zeigen wir, wie alle erwähnten Funktionalitäten in eine Visualisierungsapplikation integriert werden, welche einen Grafiker bei der Nachbearbeitung eines digitalisierten archäologischen Monuments unterstützen kann.

Acknowledgements

First, I want to thank Michael Wimmer, my supervisor, for his support. He always knew an answer to my questions, gave me valuable tips and replied quickly to my emails.

I also want to thank Murat Arikan for providing me with information about linear equation systems and checking my code for the leveling procedure.

A big thanks goes to my sister Vera who supported me with free coffee capsules and food for small technical jobs in her apartment.

I especially want to thank my parents Elisabeth and Franz for their amazing support during my study. They always motivated me in times of absent motivation and kept me afloat financially.

Contents

1 Introduction **1**
- 1.1 Motivation 2
- 1.2 Aim 5
- 1.3 Contributions 5
- 1.4 Structure 7

2 Related Work **9**
- 2.1 Labeling 9
- 2.2 Leveling 14
- 2.3 Virtual Texturing 16

3 Automatic workflow **19**
- 3.1 Labeling – MosaicBuilder 19
  - 3.1.1 Labeling overview 19
  - 3.1.2 Photo undistortion 23
    - 3.1.2.1 Riegl 23
    - 3.1.2.2 Adam Technology 25
    - 3.1.2.3 Conclusion 27
  - 3.1.3 α-expansion Graph Cuts 27
  - 3.1.4 Occlusion Detection 32
    - 3.1.4.1 Triangle sight frustum 34
    - 3.1.4.2 Edge sight triangle 37
  - 3.1.5 Shift Vectors 39
  - 3.1.6 Implementation 41
  - 3.1.7 Issues 42
- 3.2 Leveling – PoissonLeveler 43
  - 3.2.1 Leveling overview 43
  - 3.2.2 Solving the least squares problem 45
  - 3.2.3 Keeping the color values in range 47
  - 3.2.4 Filtering 49
  - 3.2.5 Issues 50

3.2.6 Implementation 52

4 Manual workflow 54
4.1 MaskDrawer 55
4.1.1 Overview 55
4.1.2 Implementation 55
4.2 VT-Tools 56
4.2.1 Atlas generation 57
4.2.2 Tile store generation 57
4.2.3 Update of atlas and tile store 58
4.2.4 Implementation 60

5 Results 61
5.1 Platform 61
5.2 Performance Tests 61
5.2.1 MosaicBuilder 61
5.2.1.1 Number of triangles 62
5.2.1.2 Number of labels 63
5.2.1.3 Upper threshold for octree cell side length 63
5.2.1.4 Max. number of iterations 65
5.2.2 PoissonLeveler 65
5.2.3 MaskDrawer 67
5.2.4 VT-Tools 67
5.2.4.1 Atlas generation 67
5.2.4.2 Tile store generation 68
5.2.4.3 Update procedure 68
5.3 Our applications in practice 69
5.3.1 Scanopy integration 69
5.3.2 Overall workflow example 70
5.3.2.1 Automatic Workflow 70
5.3.2.2 Manual Workflow 71
5.4 Ground Truth Test 73

6 Conclusion and Future Work 76

Bibliography 78

CHAPTER 1

Introduction

Cultural heritage is important to preserve tangible items as well as intangible attributes for future generations. Tangible items (e.g. buildings, monuments) are often too big to be saved in a secure environment like a display case in a museum in order to protect them from atmospheric conditions and natural breakup. Therefore, and due to the immense computational power of today's computers, those items are preserved in a digital manner. Laser scanners are used to produce 3D point clouds of the items of interest. Most of those items provide more interesting information than their simple geometry, namely their surface color. Texture mapping is the standard method to add details to a 3D model by projecting images representing the color information onto its surface. Some laser scanners allow the connection with a digital single lens reflex camera (*DSLR*). In case of the RIEGL VZ-400, which is shown in Figure 1.1, the camera is mounted on top of the laser scanner. It is important that the camera is well registered. During the registration process, the exact position and orientation of the camera relative to the coordinate system of the laser scanner is evaluated. Then, provided that the internal parameters of the camera (sensor dimensions, focal length, distortion) are known, an accurate back-projection of the images onto the scanned geometry is possible. For the color mapping for high-quality reconstructions of parts of the model, the registered camera is used for taking photos at every scan position in conjunction with scanning of the geometry. With a point cloud, it is not possible to map the photos onto the model in a continuous way. Therefore, often the point cloud is converted into a mesh. Because of the shape or size of the item, a single photo is usually insufficient to map color information onto the whole surface of the model. So when speaking about meshes, and assuming that every triangle of the mesh should receive its color information from exactly one photo, there have to be edges inside the model where regions, textured by different photos, adjoin to each other. Those edges express themselves as visible artifacts and are also referred to as *seams*.

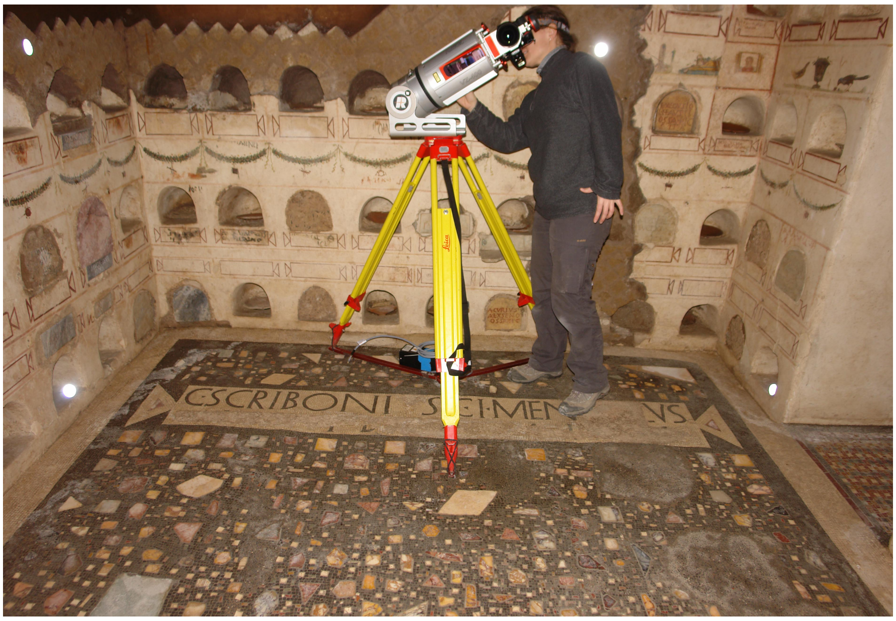


Figure 1.1: RIEGL VZ-400 laser scanner with top mounted Digital SLR. Image taken from [14].

The whole digitizing process leads to four major problems. The first problem is the finding of an optimal mapping of photos onto the mesh, so that as few seams as possible remain while at the same time choosing a high-quality photo for each triangle. In practice, camera registration errors lead to misalignments between the photos when they are mapped onto the model. A subproblem in the calculation of an optimal mapping is therefore the handling of these misalignments. The second major problem is the automatic adjustment in terms of color of the remaining seams. The third problem is the manual editing of the photos, which is often needed to handle remaining artifacts like highlights. Finally, the fourth problem is the high-performance visualization of a 3D model with a huge amount of textures.

All these problems are covered by our applications, which are altogether the topic of this thesis.

1.1 Motivation

At the Vienna University of Technology, the aim of the Terapoints project is the preservation of important archaeological monuments. This is done by laser-scanning the monuments, which yields huge point clouds. Further, photos of the monuments are taken for color mapping for high-quality reconstructions of parts of the models. Of course, the exact position and orientation of the camera during the shoot is stored for every photo.

To make a visualization of a 3D model possible, so that the photos can be mapped continuously onto it, the point cloud is triangulated inside applications like Geomagic. After the triangulation, the gathered photos are mapped onto the triangle mesh. Before the introduction of our labeling application that we will present in Section 3.1, this was done using the approach of Abdelhafiz [1], which we will describe in Section 2.1. Both in the approach of Abdelhafiz and ours, a particular triangle receives its color information from exactly one photo. When a triangulated model that was textured using such an approach is visualized, there are visible seams resulting from texturing the model surface with different photos. In Figure 1.2, such seams in the Centcelles cupola model are shown. The seams arise because of the different lighting situations during the exposure of the photos. This makes an editing of the photos absolutely necessary.

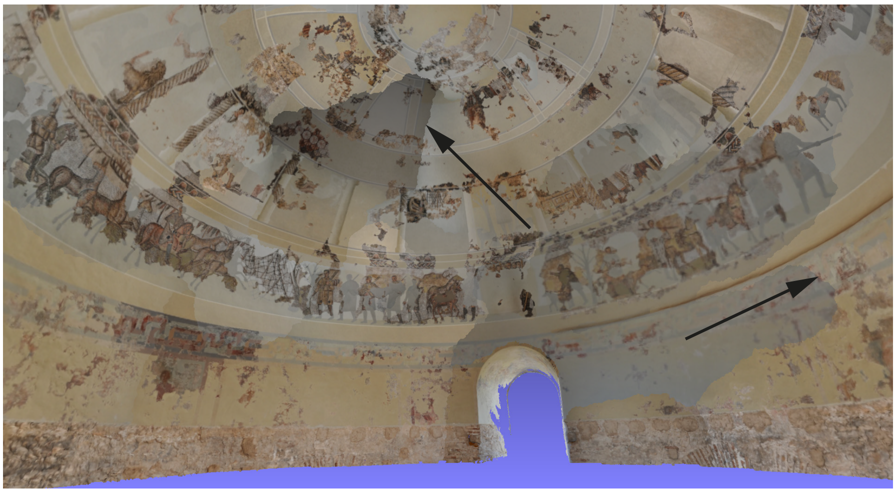


Figure 1.2: Seams in the Centcelles cupola model without further processing of the photos. The arrows show the regions where areas, getting color information from different photos, adjoin each other.

Recently, photogrammetry found its way into the Terapoints project. For this purpose, they use a commercial solution of the ADAM Technology company [30]. The reason for the usage of photogrammetry were issues with the camera-to-geometry registration. A disadvantage of the described digitizing approach of an archaeological monument using a laser scan and a DSLR is that the geometry and the color information are gathered by two different devices. Although the camera is registered, in practice the registration is never absolutely perfect. This leads to visible misalignments of the photos when they are mapped onto the model (see Figure 1.3). In contrast to the approach using a laser scanner and a DSLR, with photogrammetry only photos are taken of the item of interest. The model geometry is then entirely calculated using epipolar geometry methods [16]. The accurateness of the model is directly correlated to the image resolution of the camera. Because in photogrammetry the geometry of the model arises completely out of the image information, the camera-to-geometry registration is better in practice than with separate gathering of geometry and color. No matter which method is used, misalignments of photos can occur and have therefore to be handled.

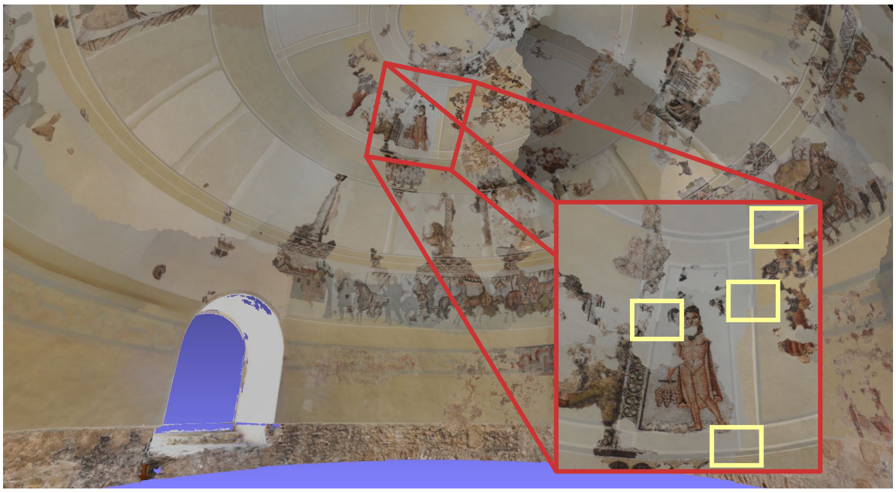


Figure 1.3: Misalignments of the photos in the Centcelles cupola model (yellow rectangles) that was digitized using a laser scanner for the geometry and a DSLR for the color information.

Another problem that arises in the post-processing of a 3D model of an archaeological monument are the low frame rates when visualizing a model with a huge amount of textures. Such a model can be the result of the digitizing process of a big item like a church of which high resolution photos are taken.

At the time this thesis was in progress, a graphic artist was editing the photos manually. This is a challenging task. To see all the artifacts (seams, misalignments) in action, the 3D model has to be visualized in an application like MeshLab [8]. When the big textures do not fit into the memory of the graphics card, a continuous reloading is necessary. This leads to slow frame rates and jerky movement through 3D space. This in turn leads to increased time durations until the model is positioned as desired to see the part of interest. After the model is positioned, and the edge with color differences is visible, the two photos belonging to the edge are loaded into an image editing application. The photos are edited manually and then the cycle starts again with the visualization of the model. This iterative process lasts until the result is satisfactory. We found that this work flow is unacceptable and decided to implement applications to ease the work of the graphic artist.

1.2 Aim

The aim of this thesis is the development of a set of applications to make texturing and visualization of a 3D model of an archaeological item as simple as possible. Beside simpleness, we place importance on accurateness and performance. This thesis does not aim to make the graphic artist jobless. In practice, it is nearly impossible to deliver a perfect result in a fully automatic way. One reason for this are the mentioned camera registration errors and the resulting misalignments in the model. These misalignments can sometimes only be reduced to a certain degree. Other visible artifacts like highlights that result from the usage of a flash light are currently not handled by our applications at all. Therefore, at least a small manual editing step by a human being in the form of a graphics artist is indispensable.

1.3 Contributions

The contributions of this thesis are:

- Implementation of four applications to simplify the post-processing of a 3D model that has been generated during the digitizing process of an archaeological item. We did not implement algorithms for conversion of a point cloud into a mesh in an automatic way. Our input is a triangulated model in conjunction with all the registered photos. For the triangulation of a point cloud, we recommend commercial solutions like Geomagic. We decided to implement one application for each of the following purposes:

    1. Labeling
       In the labeling stage, labels (in this case photos) are assigned to the triangles of the mesh. Every triangle receives its color information from exactly one photo. The result is therefore a mapping from a set of triangles onto a set of labels. Every triangle is then textured by the photo that corresponds to its assigned label. Because one particular triangle can usually be textured by more than just one photo, there is more than just one possibility to label the triangles. In practice, the number of possibilities is huge. The goal of the labeling stage is to minimize the number of edges where regions, textured by different photos, adjoin each other. Our approach is based on the findings of Lempitsky and Ivanov [18], and Gal et al. [13]. They describe the problem of finding an ideal mapping as a Markov Random Field energy optimization. Gal et al. further introduced shift vectors into the image space to compensate for the camera registration errors. We will show that our method further improves their results by the introduction of an octree into the labeling process. The octree contains all the triangles of the mesh and is

used for occlusion detection. This prevents texturing of surface areas with parts of the photos that do not contain the expected area, but the colors of an occluder. The occluder can be any part of the model (e.g. a wall) that was between the camera and the particular surface area during the exposure of the photo.

2. Leveling
   The leveling step is needed to automate the editing of the taken photos. This is done by the calculation of a 2D function that is added to the whole texture. Pérez et al. introduced a method to do image operations in the gradient domain [28]. Lempitsky and Ivanov adapted their findings to the problem of seam leveling on meshes. Their approach results in a least squares problem [18]. Our approach is based on the one of Lempitsky and Ivanov. Their least squares problem, however, has the drawback that it does not penalize too big function values. When leveling a disadvantageous original texture function, the color values can exceed the valid range. Therefore, we introduce a new term into the least squares problem that penalizes these big function values and keeps the color values in the valid range. Gal et al. further propose the calculation of a 1-pixel wide boundary around each patch in every photo [13]. When the textured model is viewed from a certain distance, where higher mipmap levels are needed, unleveled regions are filtered into the used areas. Therefore, we found that this boundary is insufficient and introduce the calculation of outline normals for every patch. These normals make a user-defined scale of the patches possible. The leveling values are linearly extrapolated along the outline normals. The upscaled patches then guarantee better filtering results.

3. Mask generation
   Masks support the graphic artist in the manual editing step. For every photo that is used inside the model, one mask is generated. A mask is an image with the same resolution as the corresponding photo. A black pixel inside a mask indicates that the corresponding pixel in the photo is unused. A white pixel belongs to a used pixel in the photo. With the masks it is obvious which parts of the photos are used and where the main focus during the manual editing step has to be. The work done for the mask generation has already been published in [3].

4. Visualization
   Because of the high amount of textures of the models, a performance boost of the visualization is necessary. Therefore, a sophisticated technique called *virtual texturing* is used. There is already a library for virtual texturing that also delivers some scripts to build the needed image structures called *atlas* and *tile store*. Those scripts are so slow that we decided to re-implement this

functionality in a faster way. Further, an update function is desirable to see the edited image parts immediately inside the 3D model. The work done for the visualization has already been published in [3].

- Testing of the implemented applications concerning accurateness and performance, as well as the comparison with existing solutions.

1.4 Structure

This thesis is structured in the following way:

1. Introduction
   This very chapter.

2. Related Work
   First, information about the automatic labeling is given. Labeling in this context is the finding of an optimal mapping of the photos onto the triangles. Second, we provide information about the already existing methods used for leveling. In the leveling process, one tries to do an adjustment in terms of color to get rid of the already mentioned seams. Last, information about virtual texturing used for high performance rendering of 3D models with a huge amount of textures is provided.

3. Automatic workflow
   We first describe the workflow of the graphic artist using our implemented tools for the automatic labeling and leveling. This directly correlates to the chronological order of the needed post-processing steps for a particular 3D model. Only an already labeled and leveled model is ready for the manual workflow, where the remaining seams and other visible artifacts are handled. We give a thorough description about our applications used for labeling and leveling.

4. Manual workflow
   After the automatic workflow, there are often manual editing steps necessary in order to get a satisfactory result. These manual editing steps were also improved via introduction of an application used for mask generation. Because the visualization of a 3D model is particularly important during and after the manual workflow, we provide the information of our application used for visualization improvement in this section.

5. Results
   In this chapter, the performance of our applications will be evaluated by means of some performance tests. These tests also estimate the impact of the different

parameters on the overall runtimes. The results of our programs for the automatic labeling and leveling are shown by the means of rendered images of an example model. We will further show, how all our applications were integrated into one big application so that thy can be used by the graphic artist in a user-friendly way.

6. Conclusion and Future Work
   We conclude this thesis by providing a summary of the work and give some ideas for future research.

CHAPTER 2

Related Work

This chapter aims to provide an overview of the existing techniques concerning labeling, leveling and virtual texturing.

2.1 Labeling

In the labeling procedures similar to our approach, every face F_i of the model gets a label P^j that corresponds to a particular photo. Assuming that there are K faces F_1 to F_K and N photos P^1 to P^N, a labeling is a mapping from the set of faces $\{F_1, ..., F_K\}$ onto the set of photos $\{P^1, ..., P^N\}$. A label P^i for a face F_j means that the face F_j is textured by the photo P^i. Obviously, not every photo can be used for every face, but some faces can be textured by more than just one photo. A simple method to find a valid mapping is also referred to as the best fragment approach [18]. The principle of the best fragment approach is the calculation of weights for the photos. Every photo gets a weight for every single face in the model. The smaller the weight, the better the photo is fitting. In the best fragment approach, every face is textured by the photo for which the weight is minimum. There are different methods to estimate the weight for a particular photo-face-pair. Lempitsky and Ivanov calculate the squared sine of the angle between the view vector and the face normal [18]. If the face is viewed perpendicularly from the front, the view vector and the face normal coincide. The squared sine of the angle between these vectors is 0.0, which denotes a perfect fit. If the face is viewed from behind, the weight would be a negative number. In this case a weight that states, that the face can not be textured with this photo, has to be used instead. When using the squared sine of the angle between the view vector and the face normal for the weight calculation, this can be e.g. 1.0. The best fragment approach is fast and easy, but in practice it produces many seams. To improve the result, Lempitsky and Ivanov consider the labeling problem as a Markov Random Field (*MRF*) energy optimization [20]. Therefore, they try to minimize the sum of two terms. The first term is the sum of all the weights that are also used in the best fragment approach. The second term belongs to all the seams occurring in a particular labeling and is a measure for the dissimilarity of the colors on either side. For the minimization process they use an algorithm called α-expansion Graph Cuts [5]. This algorithm needs more time than the best fragment approach but produces bigger homogeneous areas and less seams in the model.

Abdelhafiz follows a simpler approach for the labeling. His work is based on the calculation of the areas of the projected faces [1]. In simple terms, he counts the number of pixels of a projected face inside the photos. Then, he chooses the photo for the face for which the number of pixels is maximum. The result of this method might be similar to the best fragment approach of Lempitsky and Ivanov. To improve the result, he changes the labeling. For this, he iterates over all the vertexes in the mesh of the model. When a vertex is found that is adjacent to differently (in terms of different photos) textured faces, the photo is determined that is used for texturing of most of the adjacent faces. Let P^i be this photo. Now, all the faces that are not textured by P^i are projected into P^i. When their projections are entirely inside of P^i, the current labeling is changed so that all the faces around the current vertex get the label P^i. It is possible to do more than one iteration over all the vertexes to further improve the result. The modification of the initial labeling leads to bigger homogeneous areas. In Figure 2.1, this labeling refinement approach is illustrated. In contrast to the approach of Lempitsky and Ivanov, the color differences of the remaining seams are not considered.

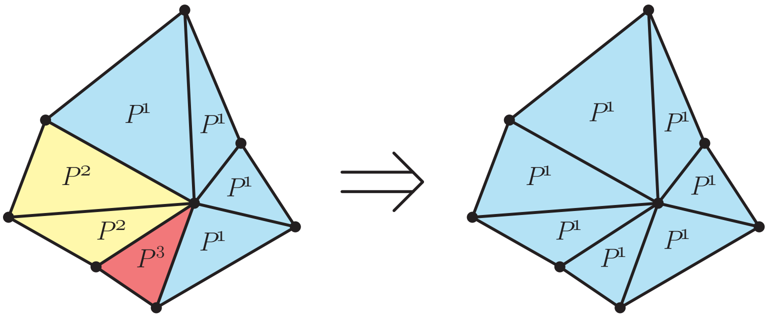


Figure 2.1: Labeling refinement approach of Abelhafiz [1]. Because P^1 is the most frequently label around the vertex in the middle (left side), and the other faces can be textured by P^1 as well, they also get the label P^1 (right side). Entries in the triangles correspond to their assigned label.

The approach of Gal et al. [13] is based on the one of Lempitsky and Ivanov [18]. They also consider the labeling problem as an MRF energy optimization. Their contribution is the introduction of shift vectors into the labeling process. The vectors are used to get rid of the camera registration errors that often arise in practice. They base their approach on the assumption, that registration errors are often just translational. With translational they mean, that when a triangle is projected into an image using erroneous camera parameters, the corresponding image material of the triangle can be found just by translating the projected triangle inside the image. Instead of just using the photos themselves, they use photo-vector-tuples as labels. A particular face is then labeled with the image region that results from projecting the face into the assigned photo and shifting this projection by the assigned shift vector. The shift vectors are $\in [0,1]^2$, therefore they correspond to a translation in texture space. The set of all possible shift vectors is a 2D continuum. However, to keep the labeling procedure combinatorial, it is necessary to do it in a hierarchical manner.

First, for all the photos the corresponding image pyramid is generated. It is not required to generate all possible levels up to a single pixel. The number of needed levels is directly linked to the magnitude of camera registration errors. In order to find the appropriate number of levels, the magnitude of camera registration errors has to be expressed in pixels. A maximum camera registration error of e.g. 10 pixels means, that the projected triangles have to be shifted by 10 pixels in the horizontal, and by 10 pixels in the vertical direction inside a photo to find the corresponding image material. When the maximum error is one pixel, only level 0 representing the full resolution images is needed. With two levels (level 0 and level 1) one can compensate for an error of three pixels. Levels 0, 1 and 2 are needed to compensate for a maximum camera registration error of seven pixels and so on. So, with a highest level number k a registration error of $2^{k+1} - 1$ can be compensated for. Let level 0 be the original photo and level n be the highest generated level in the image pyramid. A highest level number of n results in $n + 1$ Markov Random Field energy optimizations. In the first iteration, for all the photos level n of the corresponding image pyramid is considered. Not the photos themselves are used as labels, but for every of these photos nine labels are generated. These nine labels result from the conjunction of the photo and one of the vectors of $\{-1, 0, 1\}^2$ in pixel dimensions. A vector $(1, 1)$ at the current level states, that the projection would be shifted one pixel to the right, and one pixel up. Such a description is very inconvenient, since it is connected to the resolution of the photo at the current level. Therefore, the shift vectors are normalized to be $\in [0,1]^2$, similar to texture coordinates. The description in pixel dimensions is only used for explanation reasons.

After the first iteration, all labels that would be used for texturing the model are identified and used as seed labels for the next iteration for which layer $n - 1$ of all image pyramids is used. Every seed label again is used in conjunction with one of the vectors of $\{-1, 0, 1\}^2$ in pixel dimensions. Note here, that a shift vector $(1, 1)$ in pixel dimensions at level $n-1$ is just half the size of a shift vector $(1, 1)$ in pixel dimensions at level n, when they are both converted into texture space. After level $n-1$, level $n-2$ of the image pyramid is used and so on, until level 0 was used. For accuracy reasons, Gal et al. propose a further iteration where shift vectors with the length of only half pixels are used at the full resolution images.

The approach of Gal et al. can be easier explained by means of an example. Consider a set of poorly registered photos $\{P^1, ..., P^N\}$ with a resolution of 1024×768 pixels that are used for texturing of a 3D model consisting of the triangles $\{F_1, ..., F_K\}$. Assuming that the maximum camera registration error has a magnitude of 63 pixels with respect to the photos, we need the levels 0 to 5 of the image pyramid of every photo according to the mentioned formula $2^{k+1} - 1$. The resolution of the photos at level 5 is 32×24 pixels. For the first iteration in the approach of Gal et al., we generate $9N$ labels by connecting every photo of level 5 of the image pyramid with each of the shift vectors $\{-1, 0, 1\}^2$ in pixel dimensions. Because it is more convenient to normalize the shift vectors so that they correspond to texture space, and the photos have a resolution of 32×24 pixels at level 5, the resulting shift vectors are of the form (a, b) with $a \in \{-\frac{1}{32}, 0, \frac{1}{32}\}$ and $b \in \{-\frac{1}{24}, 0, \frac{1}{24}\}$. These labels are now used in the first iteration in a Markov Random Field energy optimization to receive a good labeling result. All the labels, that would be used for texturing of the model and are therefore not discarded in the first iteration, act as seed labels for the second iteration. In the second iteration, level 4 of the image pyramid of the photos is used. At level 4, the photos have a resolution of 64×48 pixels. Now the seed labels are connected with each of the shift vectors $\{-1, 0, 1\}^2$ in pixel dimensions. The shift vectors at level 4 are normalized again and are therefore of the form (a, b) with $a \in \{-\frac{1}{64}, 0, \frac{1}{64}\}$ and $b \in \{-\frac{1}{48}, 0, \frac{1}{48}\}$. Because the seed labels already contain a shift vector, the shift vectors are accumulated. Consider a label $(P^1, (\frac{1}{32}, \frac{1}{24}))$ that was not discarded in the first iteration and is therefore used as seed label for the second iteration. When this label is connected with the shift vector $(\frac{1}{64}, \frac{1}{48})$, the result is the label $(P^1, (\frac{3}{32}, \frac{3}{64}))$ that is now used for the second iteration. The resulting labels are then used again in a Markov Random Field energy optimization to receive a good labeling result. The labels that were not discarded then act as seed labels for the third iteration. The whole procedure is continued exactly the same way until the full resolution photos at level 0 of the image pyramids are reached. After the iteration for level 0 was carried out, one additional iteration is done by connecting the seed labels with the shift vectors $\{-0.5, 0, 0.5\}^2$ in pixel dimensions, which equals a shifting of only half pixels. The labels that result from this last iteration are then the final labels that are used for texturing of the model.

Another method dealing with image-to-geometry misregistrations was presented by Dellepiane et al. [10]. They also try to find an optimal mapping of photos onto a meshed model. They estimate the regions on the model, where two or more projected photos overlap. To decide, which photo shall be used for texturing of a particular triangle, they calculate weights based on the weighting system of [6]. Simply using the “best” photo for every triangle would lead to visible discontinuities in the final texture because of the misregistrations. Therefore, they do the following. For all overlapping image pairs, they project the first photo onto the model, and then back-project it into the second photo.

This is also done the other way round, to project the image information of the second photo into the first one. Then, they calculate the so called optical flow, which is basically a set of displacement vectors between corresponding pixels. This information is then used for warping of the image information in the overlapping regions. The resulting texture of the model then has no visible discontinuities at edges where regions adjoin, that get its color information from different photos.

Musialski et al. presented a method for the generation of high-quality approximated façade ortho-textures based on a set of perspective photos [27]. Based on structure-from-motion and using the perspective photos as input, they calculate a point cloud representing the scene geometry. They assume the façade to be planar. Therefore, they introduce a plane that best approximates the position of the façade. The plane is rasterized so that each element of the raster represents a single pixel of the final image. The taken photos are then back-projected onto this plane. The labeling part of their approach is the decision which of the pixels in the resulting image gets its color information from which photo. For every pixel, they use the "best" photo as the origin of the color information. Photos, that correspond to camera positions with a small distance to the raster element on the plane and almost perpendicular viewing angle are preferred. Occlusions are also handled. Occluding objects express themselves as geometry points in front of the façade plane. For every camera position, these points are splatted onto the plane in a projective way. The result of this splatting operation is then used to generate masks for the photos that encode the areas corresponding to occluding objects. Beside this method, they also allow the insertion of explicitly modeled objects into the scene. These objects are then also used for mask generation to encode the occluded regions in the photos. Certainly, pixels in the photos, that belong to occluding objects, are penalized in the labeling operation. For further improvement of the result, they allow the user to interactively select regions in the final image in order to texture them with another photo.

In contrast to our mesh-based approach, Pintus et al. directly work on (extremely dense) point clouds [29]. The result is an assignment of colors to the points. The origin of the color information is a set of high-resolution photos. In contrast to our approach where every basic element (in our case triangles) gets its color information from a single photo, they assign colors to points, that are weighted sums of several pixels. First, they re-organize the point stream in order to minimize the access operations to invisible points with respect to a particular photo. For visibility calculation, they use the depth buffer. They project the visible points onto the image plane of the particular camera and fill the regions in the depth buffer having no depth value via an interpolation approach. This guarantees that the depth information represents not only a set of projected points, but an entire surface. Then, they assign weights to the pixels in every photo. The weights are calculated based on the visibility of the projected points, which can be easily evaluated using the values of the depth buffer. Because a simple usage of the calculated values leads to visible seams in the rendered point cloud, they first look for edges in the estimated weighting function. Based on this information, they do a distance transform on the edge maps and multiply the distance field with the weighting function. This guarantees a smooth transition between different photos. To avoid blurring, they do multi-band blending. Finally, adaptive point cloud refinement is carried out in regions, where more points are needed for mapping of high-frequency image information.

2.2 Leveling

In the leveling stage, one tries to get rid of the visible artifacts in the model where regions that are textured by different photos adjoin each other. These seams arise because of the different lighting situations during the exposure of the photos.

There is much literature available concerning leveling. However, most of it does not directly deal with leveling on meshes, but only covers approaches used for stitching together two or more planar photos to get a panorama image. Approaches proposed in [24], [12] and [9], which are also referred to as optimal seam methods, search for a curve in the overlapping region of two photos, and use the curve as the border between the two photos. An adaption of such methods to the leveling on meshes is difficult, since the region where the photos adjoin is already defined by the edge that corresponds to the differently textured triangles. The photos could be distorted so that the estimated curve is mapped to the edges in the mesh. A disadvantage of such approaches are, that no colors are changed. They are therefore not suited for photos that are taken under significantly different lighting conditions.

Other approaches blend between the images under consideration to get a smoother intersection [31]. Blending could also be adapted to meshes so that the gradient of the blending function would be applied perpendicularly to the edge of the mesh. Assuming that one of the photos is significantly lighter or darker than the other, no sharp edge would be visible, but the intensity difference of the photos attracts attention at some distance from the edge. Furthermore, a misalignment of the photos produces ghosting artifacts when they are simply blended.

A better idea than blending or searching for an optimal seam in order to stitch two images together is to keep the variation of the images, namely their gradients, while bringing their intensity levels onto the same level. This approach has its origin in the paper of Pérez et al. [28]. They introduce a new way of image editing that is based on the manipulation of gradients instead of colors themselves. One of their proposed applications is seamless cloning of an object of one image into another image. This is done by keeping the gradients of the copied object nearly unmodified while adjusting the boundary colors to the surrounding colors of the new background. The solution of this operation results in a Poisson equation with Dirichlet boundary conditions.

Levin et. al propose two similar methods for image stitching [19]. The first method stitches two images together by minimizing the differences between the gradients of the input images and the resulting image. For the second method they first stitch the gradient fields of the input images, and then calculate the resulting image by minimizing the differences between the gradients of the stitched gradient fields and the resulting image. The approach of Levin et. al is highly related to [28].

Lempitsky and Ivanov adapted the findings of [28] to the problem of seam leveling on meshes [18]. Instead of solving a Poisson equation, they approximate the solution by solving a least squares problem. A disadvantage of their method is a missing mechanism to keep the calculated color values in the valid range.

A very interesting approach for leveling of photos that were taken using only one significant light source, e.g. a flash light, was proposed by Dellepiane et. al [10]. Such photos are often unevenly illuminated and suffer under artifacts like specular highlights or shadows. The only constraint of the proposed method is, that the position of the light source in relation to the optical center of the camera keeps unchanged. In case of conventional cameras this is no problem, since the flash light is either built in, or in case of a DSLR it is mounted on top and therefore moves with the camera. Their contribution is the introduction of the, as they call it, color correction space. The color correction space coincides with the view frustum space of the camera. Every point in it corresponds to an affine transformation used for color correction. These affine transformations are evaluated in an empirical way at some discrete points of the view frustum, and are then interpolated for points in between. For every pixel of the photo that should be color corrected, the corresponding depth information is needed. The depth information can be evaluated by taking two or more photos of the scene from different positions and using shape-from-stereo methods. In order to get a color corrected photo, every pixel has to be transformed the following way. Using the depth information, the point in the scene can be found that was projected onto the pixel. This point corresponds to a particular affine transformation in the color correction space, that has then to be used for color correction. In their paper, they also show how the color correction space can be established for a particular camera-light source pair using an empirical approach. The pleasant fact about the color correction space is, that it has to be established for a particular camera-light source combination only once. Beside the color correction space, Dellepiane et al. also show how other image artifacts introduced by flash light (e.g. specular highlights, shadows) can be removed. They evaluate the exact position of the flash light in relation to the optical center of the camera. Provided that the scene geometry, the position of the camera and the position of the light source is known, which is the case in the mentioned setup, shadow mapping can be used to find shadowed regions in the photo. The specular highlights in the photo, or at least candidate pixels, can be found by the introduction of the Phong model.

Our approach is based on the approach of Lempitsky and Ivanov [18]. In contrast, we introduce a further term into the least squares problem that pays attention on the final color values so that they reside in the valid range.

2.3 Virtual Texturing

Texture mapping was pioneered in 1974 by Edwin Catmull in his Ph.D. thesis [7]. It was a big invention, and is used in nearly all fields of computer graphics like visualization, real-time rendering and computer games. Without it, also the digitized models for cultural heritage would look quite boring. Although these models can be colored using texture mapping, the amount of textures is so big that they do not fit entirely into the memory of a commercially available graphics card. This problem calls for an out-of-core solution. Since its introduction there has been heavy research regarding texture mapping, but most publications for out-of-core rendering deal with geometry complexity [15]. Dealing with huge 3D models is important too, but in our case the huge amount of textures is the limiting factor for visualization.

Virtual texturing solves the problem of huge texture data sets. It was presented by Mittring in [25] and covered in detail by J. Mayer in [23]. It is a highly sophisticated technique that is used for rendering of very detailed scenes whose textures do not fit into the memory of the graphics card. The basic concept of virtual texturing is to stream only those parts to the graphics card that are currently visible. Of course, such a scene can be visualized without virtual texturing as well, but the frame rate would probably suffer extremely. Without virtual texturing, every texture would be streamed to the graphics card as a whole, regardless of the visible areas. Because not all textures fit into the memory of the graphics card, this leads to permanent reloading. The result will be slow frame rates and jerky movement through 3D space. When using virtual texturing, two important data structures are needed.

The first of these data structures is the so-called *atlas*. The atlas is a single texture that consists of all the textures that belong to the virtually textured models in the scene. It must meet some requirements regarding its size. In the simplest case, when no bilinear filtering is used, its side length has to be power of two. There is no theoretical limit for the maximum size of the atlas. The maximum size is essentially bound by the maximum texture size that is allocatable on the graphics card. In the LibVT, the library for virtual texturing that was developed by Mayer, the maximum atlas size is 128k^2 . Because such a big image is very unhandy, for example to load it into an image editing application for observation reasons, it can be stored in 4, 16, 64 or more files on hard disc. In Figure 2.2, an atlas consisting of 70 single images is shown. Every image has a resolution of 4256×2832 pixels. The resulting atlas has a side length of 32k. Of course, the texture coordinates of the virtually textured models have to be altered to reference the correct positions inside the atlas.

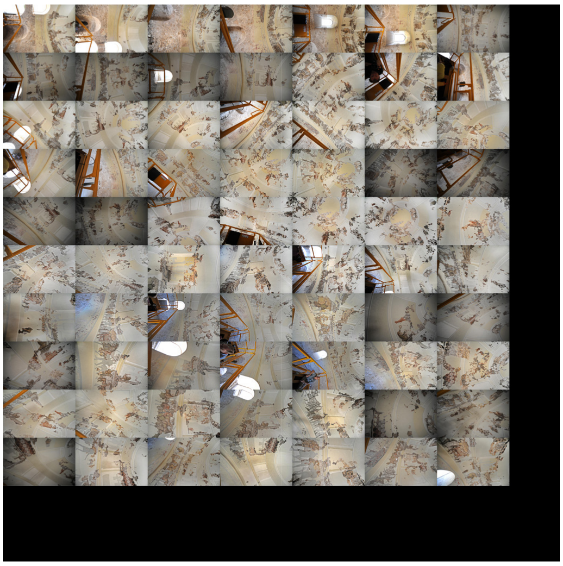


Figure 2.2: Atlas used for virtual texturing consisting of 70 images. The images have a resolution of 4256×2832 pixels each. The atlas has a side length of 32k.

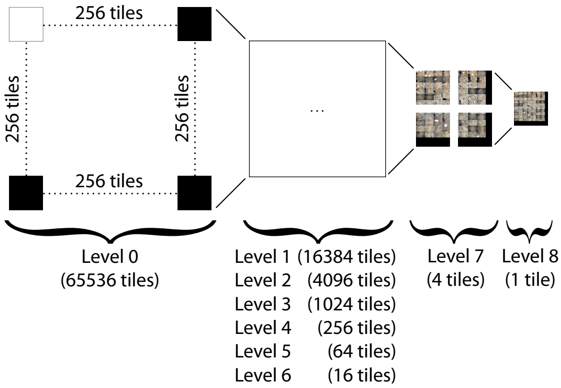


Figure 2.3: Tile store used for virtual texturing consisting of all in all 87381 tiles. It is based on the atlas of Figure 2.2. The tiles have a side length of 128 each.

After the generation of the atlas, the second data structure needed for virtual texturing – the so-called *tile store* – can be generated. In Figure 2.3, the tile store that is based on the atlas of Figure 2.2 is illustrated. The basic element of the tile store is a so-called *tile*. A tile is a small texture. “Common tile-sizes range from 64^2 up to 256^2 pixels” [23]. These tiles are the atomic elements that are streamed to the graphics card. The structure of the tile store is directly related to mipmapping [32]. In mipmapping, not only the full resolution of a single texture is used, but several versions with lower resolution. These versions originate in scaling down the original image to half size again and again until only one pixel resides. For the rendering process, the version that best fits the requirements (or the two versions, in case of trilinear filtering), is used to deliver the color for a single pixel on the screen. Similar to mipmapping, the tile store consists of several resolution levels. Level 0 represents the atlas at its full resolution and is constructed by cutting the atlas in equally sized tiles. After that, the atlas is scaled down to half size to deliver the image data for level 1. Level 1 is then constructed by cutting this scaled version of the atlas into equally sized tiles. The tiles have the same resolution at every level. Assuming that the atlas has a side length of 32k, and the tiles have a resolution of 128^2, there would be 65536 tiles at level 0 of the tile store, 16384 tiles at level 1 and so on. In contrast to mipmapping, the atlas is not scaled down to one pixel, but to the size of a single tile. Continuing our example, at level 8 there would be only one tile representing the whole atlas.

The rendering of the virtually textured models needs two passes. In the first pass, a special shader is used in order to find the needed tiles for the current frame. In every pixel of the resulting image, the exact coordinate of the tile that is needed to deliver the color information for that pixel is encoded. This image is then read back to the CPU to estimate the needed tiles. The needed tiles are then read from the medium where they reside (e.g. hard disc), compacted and streamed to the GPU. On the GPU, they are stored in the so-called *physical texture*. There is also a texture on the GPU that acts as a map for the physical texture. This *pagetable* is updated so that the location of the tiles in the physical texture can be easily evaluated. In the second render pass, another special shader is used to render the final frame. This shader uses the pagetable in order to find the needed tiles in the physical texture. These tiles are then used as the origin of the texture information needed to render the final frame.

CHAPTER 3

Automatic workflow

In this chapter, we will present our applications used for automatic labeling and leveling. Because until this thesis was in progress, the simple approach of Abdelhafiz [1] was used for labeling, we decided to implement a more sophisticated labeling mechanism in a first step. In a second step, the labor-intensive image editing was the target of improvement.

In Section 3.1, we will describe the *MosaicBuilder*, which is our application used for automatic labeling. We will first formulate the labeling problem. Then, we will speak about the significance of photo undistortion and the different distortion models we were faced with during our labeling experiments. After that, we will show how the energy term which is the result of the labeling problem formulation can be minimized. We will further show how occlusion detection was introduced into the labeling procedure. Finally, we will talk about the implementation of our application.

In Section 3.2, we will describe the *PoissonLeveler*, which is our application used for automatic leveling. We will first formulate the leveling problem. After that, we will show how the least squares problem which is the result of the leveling problem formulation can be solved. In a section speaking about the divergence of color values, we will introduce a new term into the least squares problem to keep the color values in range. Then, we will show how the filtering results for the visualization of a model that is textured using a set of leveled photos can be improved. Finally, we will take a look at the implementation of our application.

3.1 Labeling – MosaicBuilder

3.1.1 Labeling overview

Our approach for labeling is based on the approaches of Lempitsky and Ivanov [18], and Gal et al. [13]. Both papers consider the labeling problem as a Markov Random

Field (*MRF*) energy optimization. Lempitsky and Ivanov formulate the energy term as follows. Let $\{F_1, ..., F_K\}$ be the set of faces in a model and let $\{P^1, ..., P^N\}$ be the set of registered photos that are used for texturing of the model. The resulting labeling, which is also referred to as a *texture mosaic*, is then defined by a labeling vector $\mathbf{M} = (m_1, m_2, ..., m_K)$, where $m_1, m_2, ..., m_K \in \{0, 1, ..., N\}$. An element m_i in $\mathbf{M}$ states that the face F_i is textured by the photo P^{m_i}. The best-fragment approach that we described in Section 2.1 can therefore be written as $\mathbf{M} = (m_1, m_2, ..., m_K)$, where every $m_i = \arg\min_j w_i^j$. Here, the term w_i^j is the cost to texture the face F_i with the photo P^j. The term w_i^j is also referred to as the *data cost*. As proposed in [18], we calculate the cost value w_i^j as $\sin^2 \alpha$, where α is the angle between the view vector and the corresponding face normal. In Figure 3.1 this scenario is illustrated.

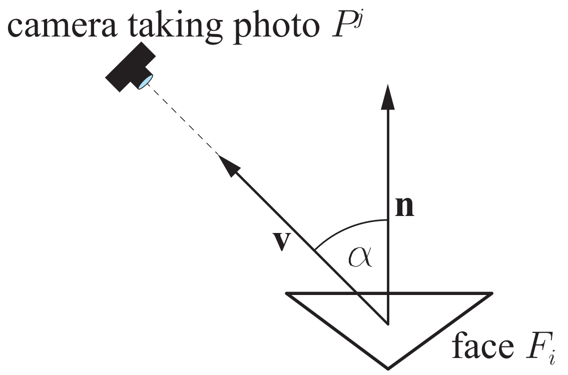


Figure 3.1: A face F_i of the model with its normal vector **n** and view vector **v** to the camera taking the photo P^j. The cost value w_i^j is calculated by $\sin^2 \alpha$.

In future releases of our application, we want to consider even more metrics to calculate the weight for a particular photo-face-pair. In [6], Callieri et al. calculate the quality of image material on a per-pixel basis. For a particular photo, they compute masks using different metrics. These masks are then multiplied in order to get a final mask so that every pixel in the final mask encodes the quality of the corresponding pixel in the photo. Although we need weights not for the pixels of the photos but for all photo-face-pairs, it would be no problem to use the masks in our approach. The weight for a particular photo-face-pair would be simply computed by projecting the face into the photo, and considering the values in the final mask of the photo at the pixels the face is projected onto. Some of the masks that are computed in [6], which we can also think of to use them in our approach, are:

- **Border Mask.** Every pixel in the border mask stores the distance of the pixel to both the image borders and discontinuities in the depth map. Higher distances correspond to better image material.

- **Focus Mask.** In the focus mask, the value of each pixel is a measure for the focusing. Higher values correspond to better image material.
- **Stencil Mask.** Often the user wants to exclude portions of the photos which are then not considered in the labeling procedure. The stencil masks are provided by the user and encode the areas that can be used for texturing.

The best fragment approach does not take the color differences of the photos into account. The result are many seams in the final texture, which are shown in Figure 3.2. Therefore, another cost value is introduced, which accounts for the seams. Consider two adjacent faces F_i and F_j of the mesh, that share an edge E_{ij}. Given a labeling vector **M**, the cost produced by this edge is calculated by:

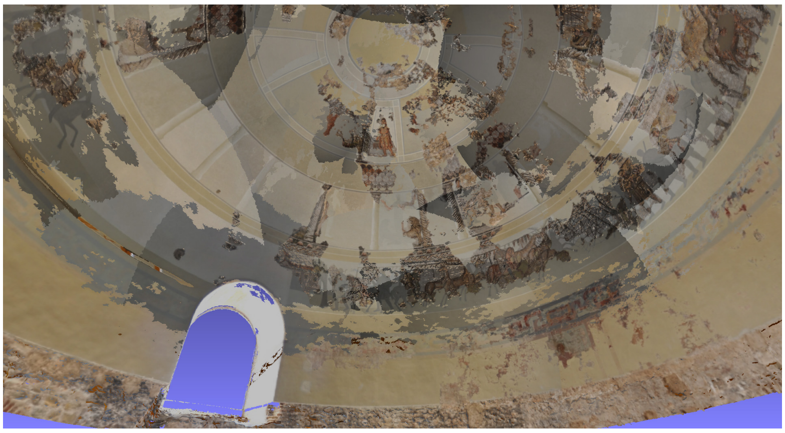
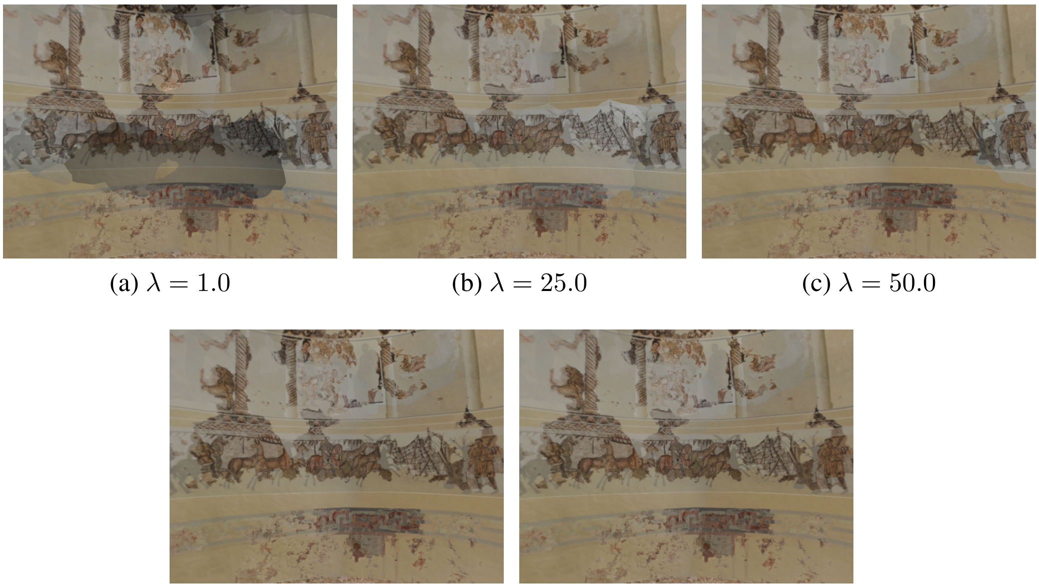


Figure 3.2: Result of the labeling using the best fragment approach. Every face is textured by the "best" photo, which leads to many seams.

$$w_{i,j}^{m_i,m_j} = \int_{E_{ij}} d(Pr_{m_i}(X), Pr_{m_j}(X))dX \tag{3.1}$$

In Equation 3.1, Pr_i is a projection operator for the photo P^i. The operator $d(.,.)$ returns the distance between two color samples. Similar to Lempitsky and Ivanov, we use a Euclidean distance between RGB values. The minimum distance between two color samples is therefore 0.0 if the colors are identical. The maximum distance is $\sqrt{3}$, corresponding to the distance between a white and a black pixel in normalized RGB space. The photos are not continuous functions, so the integral of Equation 3.1 must be discretized. It is therefore a sum of distances between color values along the projected edge. Certainly, this sum is 0.0 when the faces F_i and F_j sharing an edge E_{ij} are textured by the same photo. The term $w_{i,j}^{m_i,m_j}$ is also referred to as the *smoothness cost*.



(a) $\lambda = 1.0$ (b) $\lambda = 25.0$ (c) $\lambda = 50.0$

(d) $\lambda = 75.0$ (e) $\lambda = 100.0$

Figure 3.3: Impact of the parameter λ on the final texturing result. Note how the quality of image material per face decreases with increasing λ, while the transitions become smoother.

Let $\mathcal{N}$ be the set of adjacent faces in a model. Then, the final energy term that has to be minimized can be written as:

$$E(\mathbf{M}) = \sum_{i=1}^{K} w_i^{m_i} + \lambda \sum_{\{F_i, F_j\} \in \mathcal{N}} w_{i,j}^{m_i, m_j} \tag{3.2}$$

The value λ in Equation 3.2 is typically ≥ 0.0. It defines the degree of penalizing of edges, shared by faces that are textured by different photos. If 0.0 is chosen as the value for λ, the minimization of Equation 3.2 degrades to the best fragment approach, and every face is textured by the “best” photo. With increasing λ, the importance of quality of image material used for the faces decreases, because the whole effort goes into the establishment of smooth transitions between areas that receive its color information from different photos. This behavior is illustrated in Figure 3.3. The used model is the Centcelles cupola model.

In Figure 3.4, the energy values that were the result of approximately solving Equation 3.2 for the Domitilla cubiculum model are shown. The term data cost refers to the first term of the equation, while smoothness cost refers to the second term. To be comparable, we divided the resulting values of the second term by λ. Note how the data cost increases with increasing λ, while the smoothness cost decreases. With increasing λ, color differences between images that are used for texturing of adjoining faces are more and more penalized. Assuming that a region is already textured, the neighbor region would rather be textured by an image with similar colors, than by an image corresponding to a perpendicular view onto the region of interest, but having completely different colors.

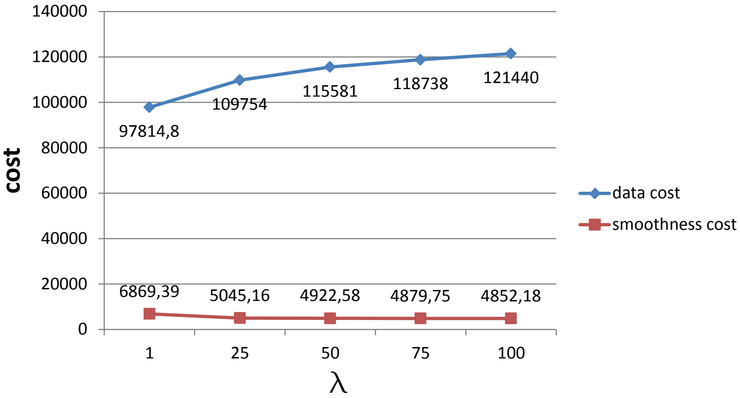


Figure 3.4: Impact of the parameter λ on the resulting cost values. Data cost refers to the first sum of Equation 3.2, smoothness cost to the second one.

3.1.2 Photo undistortion

An essential part of the labeling process is the projection of triangles into the photos so that the corresponding image material needed for texturing is found. It is important that the distortion parameters of the camera are correctly incorporated into the projection, otherwise an inaccurate texturing of the model is the result. During our labeling experiments, we were faced with different types of distortion models. Obviously, the Riegl company uses a different distortion model than Adam Technology. Because the undistortion of the photos is important to ensure an adequate labeling result, we will provide their methods for projection with incorporated image undistortion in the following.

3.1.2.1 Riegl

The Centcelles cubiculum model that we already showed in figures in prior sections was digitized using a Riegl laser scanner and a DSLR. There is also a software package that is shipped with every terrestrial Riegl laser scanner, called RiScan Pro. This software package is then used for managing of the data that is gathered during the scans. With every new scan project, an XML-based project file is generated that contains information about all the scan positions, taken photos and also the distortion parameters of the used camera(s). The used projection method is well documented in a file that is generated in conjunction with every project file. This file is the document type definition for the project file. The projection method and the way how the distortion parameters are incorporated are similar to the projection method that is implemented in the OpenCV library.

The internal parameters of the camera are contained in the camera matrix **A**, which is shown in Equation 3.3.

$$\mathbf{A} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{3.3}$$

The parameters f_x and f_y are the focal length of the camera in pixels in horizontal and respectively vertical direction. Note that the cells on the sensor of the camera corresponding to the pixels of the taken image do not have to be perfect squares. The parameter f_x refers to the width of the pixels on the sensor, while the parameter f_y refers to the height. Therefore, f_x and f_y do not have to be the same value. c_x and c_y are the coordinates of the principal point in pixels with respect to the upper left corner of the image.

Assuming that a perfect distortion-free camera is used, the homogeneous coordinates $(u^{'}, v^{'}, w^{'})^{\top}$ in pixels resulting from the projection of a point $\mathbf{p} = (X, Y, Z, 1)^{\top}$ that is in world space into the image plane of the camera is calculated by the formula shown in Equation 3.4.

$$\begin{pmatrix} u^{'} \\ v^{'} \\ w^{'} \end{pmatrix} = \mathbf{A} \cdot [\mathbf{R}|\mathbf{t}] \cdot \mathbf{p} \tag{3.4}$$

A is the camera matrix that was shown in Equation 3.3. $[\mathbf{R}|\mathbf{t}]$ is a matrix that results from concatenating the rotation matrix $\mathbf{R} \in \mathbb{R}^{3,3}$ and the translation vector $\mathbf{t} \in \mathbb{R}^{3,1}$. The matrix $[\mathbf{R}|\mathbf{t}]$ describes the transformation from world space into the view space of the camera. The inhomogeneous coordinates $(u, v)^{\top}$ in pixels of the projected point inside the image is then simply computed by division of $u^{'}$ and $v^{'}$ by $w^{'}$, so that $(u, v)^{\top} = (^{u^{'}}/_{w^{'}}, ^{v^{'}}/_{w^{'}})$.

In practice, lenses are never distortion-free. Therefore, we have to take distortion into account. In RiScan Pro, there are all in all six coefficients to model the distortion. The four coefficients k_1, k_2, k_3 and k_4 account for the radial distortion, while the two coefficients p_1 and p_2 account for the tangential distortion. The radial coefficients are used a bit different than in the underlying OpenCV model.

The calculation of (u_d, v_d), which are the coordinates of a projected point with incorporated distortion, is then done by the formulas shown in Equation 3.5 and Equation

3.6. The variables x and y are calculated by $({}^{u-c_x}/_{f_x})$ and $({}^{v-c_y}/_{f_y})$ respectively.

$$u_d = u+x\cdot f_x\cdot(k_1\cdot r^2+k_2\cdot r^4+k_3\cdot r^6+k_4\cdot r^8)+2\cdot f_x\cdot x\cdot y\cdot p_1+p_2\cdot f_x\cdot(r^2+2\cdot x^2) \quad (3.5)$$

$$v_d = v+y\cdot f_y\cdot(k_1\cdot r^2+k_2\cdot r^4+k_3\cdot r^6+k_4\cdot r^8)+2\cdot f_y\cdot x\cdot y\cdot p_2+p1\cdot f_y\cdot(r^2+2\cdot y^2) \quad (3.6)$$

The last missing parameter that has to be explained is r^2. In RiScan Pro, there are two versions of calculating r^2. In order to know which version has to be used, the *version* tag has to be evaluated for the corresponding camera calibration in the project file. If the version tag is 0 or it is missing at all, r^2 is calculated by x^2+y^2. If the version tag is 1, r^2 is calculated by $\arctan(\sqrt{x^2+y^2})$. Riegl proposes that the second version gives better results when using lenses with very large field of view.

3.1.2.2 Adam Technology

As already stated in Section 1, beside laser-scanning also photogrammetry in the form of the commercial software 3DM Analyst by Adam Technology is used for digitizing of archaeological items in the Terapoints project. In contrast to the Riegl company, where the camera model used for projection is fully documented, there is no such documentation from Adam Technology for the models produced by 3DM Analyst. In a first step, we tried to use the same projection methods and undistortion as it is done in RiScan Pro. This, however, led to catastrophic results. Then we tried the approach that is also proposed in [1], which delivered satisfactory labelings. Because there is no documentation from Adam Technology for the used camera model and undistortion methods implemented in the 3DM Analyst software, we use exactly the same camera model and undistortion methods that are proposed in [1] for the models produced by 3DM Analyst. We will provide these methods in the following.

In order to find the image coordinates in pixels of a point $\mathbf{p} = (X, Y, Z, 1)^\top$ that is in world space, it is first only transformed into view space using Equation 3.7. This is in contrast to RiScan Pro, where the point in world space is directly transformed into pixel coordinates. $[\mathbf{R}|\mathbf{t}] \in \mathbb{R}^{3,4}$ is again the matrix that transforms the point from world space into the view space of the camera.

$$\mathbf{p}_{\text{view}} = \begin{pmatrix} x_{\text{view}} \\ y_{\text{view}} \\ z_{\text{view}} \end{pmatrix} = [\mathbf{R}|\mathbf{t}] \cdot \mathbf{p} \quad (3.7)$$

In the appproach proposed in [1], the point is then transformed into image space in millimeters with respect to the image center because all the distortion parameters are incorporated in this space. The transformation is done using Equation 3.8.

$$\mathbf{p}_{\text{image}} = \begin{pmatrix} x_{\text{image}} \\ y_{\text{image}} \end{pmatrix} = -c \cdot \begin{pmatrix} x_{\text{view}}/z_{\text{view}} \\ y_{\text{view}}/z_{\text{view}} \end{pmatrix} + \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \tag{3.8}$$

In Equation 3.8, c is the focal length in millimeters. The minus sign in front of c is needed because in the projection model proposed in [1] the origin of the image is assumed to be in the lower left corner. Therefore, we are looking in the negative Z-direction of the coordinate system of the camera. This is different to RiScan Pro, where the origin of the image is assumed to be in the upper left corner. The values x_0 and y_0 in Equation 3.8 are the coordinates of the principal point of the camera sensor in millimeters with respect to the lower left corner of the image sensor.

Now the distortion parameters are taken into account. At first, the overall distortion is calculated. Then, the image point $\mathbf{p}_{\text{image}}$ is adapted using this distortion. In this distortion model, there are four coefficients k_1, k_2, k_3 and k_4 accounting for the radial distortion. The calculation of the radial distortion is shown in Equation 3.9. The calculation of the needed parameter r^2 is shown in Equation 3.10.

$$\boldsymbol{\delta}_{\text{radial}} = (k_1 \cdot r^2 + k_2 \cdot r^4 + k_3 \cdot r^6 + k_4 \cdot r^8) \cdot \mathbf{p}_{\text{image}} \tag{3.9}$$

$$r^2 = (\mathbf{v}_{\text{offset}})^\top \cdot \mathbf{v}_{\text{offset}}, \text{ with } \mathbf{v}_{\text{offset}} = \mathbf{p}_{\text{image}} - \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \tag{3.10}$$

The tangential distortion is modeled using two coefficients p_1 and p_2. The calculation is shown in Equation 3.11.

$$\boldsymbol{\delta}_{\text{tangential}} = \begin{pmatrix} p_1 \cdot (r^2 + 2 \cdot (x_{\text{image}})^2) + 2 \cdot p_2 \cdot x_{\text{image}} \cdot y_{\text{image}} \\ p_2 \cdot (r^2 + 2 \cdot (y_{\text{image}})^2) + 2 \cdot p_1 \cdot x_{\text{image}} \cdot y_{\text{image}} \end{pmatrix} \tag{3.11}$$

The affinity distortion is modeled using two coefficients b_1 and b_2. The calculation is shown in Equation 3.12.

$$\boldsymbol{\delta}_{\text{affinity}} = \begin{pmatrix} b_1 \cdot x_{\text{image}} + b_2 \cdot y_{\text{image}} \\ 0 \end{pmatrix} \tag{3.12}$$

The final image distortion vector $\boldsymbol{\delta}$ is now just the sum of all the single distortion vectors, so that $\boldsymbol{\delta} = \boldsymbol{\delta}_{\text{radial}} + \boldsymbol{\delta}_{\text{tangential}} + \boldsymbol{\delta}_{\text{affinity}}$.

The undistorted image coordinate in millimeters with respect to the image center is now computed by the equation shown in Equation 3.13.

$$\mathbf{p}_{\text{undistorted\_mm}} = \begin{pmatrix} x_{\text{undistorted\_mm}} \\ y_{\text{undistorted\_mm}} \\ z_{\text{undistorted\_mm}} \end{pmatrix} = \mathbf{p}_{\text{image}} - \boldsymbol{\delta} \tag{3.13}$$

Now we can compute the final undistorted coordinates of the projected point in pixels with respect to the upper left corner of the image (see Equation 3.14). The parameters *pixWidth* and *pixHeight* refer to the width and respectively to the height of a pixel on the image sensor of the camera in millimeters. The parameters W and H refer to the width and respectively to the height of the photo in pixels.

$$\mathbf{p}_{\text{undistorted\_pixels}} = \begin{pmatrix} x_{\text{undistorted\_mm}} / \textit{pixWidth} + W/2 \\ H/2 - y_{\text{undistorted\_mm}} / \textit{pixHeight} \end{pmatrix} \tag{3.14}$$

3.1.2.3 Conclusion

In the proposed methods for projection and undistortion, we only explained how to find the image coordinates of a single point in space considering distortion. Certainly, it is possible to account for the distortion during the labeling process, to find the correct texture coordinates for the projected points, but using the original undistorted photos for texturing of the final model. A disadvantage of this approach is that only the linear distortion is considered during the visualization. Since the texture coordinates are interpolated linearly along the triangles of the mesh, the nonlinear distortion is not considered. Therefore, we first generate the undistorted versions of the original photos, and then do the labeling using these undistorted photos.

So far, we know how the problem of labeling can be formulated so that the quality of the image material for every face as well as the transitions between the faces are considered. Further, we know how the geometry can be projected into the photos so that the distortion is considered. Now the question is, how a minimization of Equation 3.2 can be calculated, or at least approximated. Lempitsky and Ivanov propose a method called α-expansion Graph Cuts. This method is explained in the following section.

3.1.3 α-expansion Graph Cuts

In 2001, Boykov et al. introduced a method for the fast approximate energy minimization via graph cuts [5]. In their paper, they describe the algorithm only for the labeling of image pixels. Nonetheless, it can easily be adapted to meshes, as we will see in this section. Because the notation used in [5] is different to ours, we will stick to our notation for consistency to explain their approach.

A big problem of finding a global minimum of an arbitrary energy function is that also the minimization of energy functions following a simple pattern tends to be NP-hard. This was also proved in [5]. Therefore, the goal is finding a local minimum of the energy function shown in Equation 3.2. A labeling vector **M** is a local minimum of the energy function $E(.)$ if

$$E(\mathbf{M}) \leq E(\mathbf{M}') \text{ for any } \mathbf{M}' \text{ “near to” } \mathbf{M}. \tag{3.15}$$

$\mathbf{M}'$ refers to a labeling that can be reached by a single *move* from $\mathbf{M}$. In the approach of Boykov et al., such a move is either an α-expansion or an α-β-swap. Since we only considered the version using α-expansions, we will only describe this part. For the explanation what exactly an α-expansion is, they consider a labeling as a partition of the labeled elements into groups with the same label. In our case, this would be as follows. A particular labeling, corresponding to a labeling vector $\mathbf{M}$, can be viewed as a partition of all faces of the model. This partition is defined as $\mathbf{F} = \{\mathcal{F}^i | 1 \leq i \leq N\}$, where $\mathcal{F}^i$ is the subset of faces with the photo P^i assigned as their label. For a particular label P^α, a move from the partition $\mathbf{F}$ to another partition $\mathbf{F}'$ is called an α-expansion if $\mathcal{F}^\alpha \subset \mathcal{F}^{\alpha\prime}$ and $\mathcal{F}^{l\prime} \subset \mathcal{F}^l$ for any label $P^l \neq P^\alpha$. In simple terms, an α-expansion leads to an increase of the area in the mesh textured by P^α. Every face that was labeled by P^α does not change, but at least one face that was not labeled by P^α gets P^α assigned as its new label. In Algorithm 3.1, the basic structure of the α-expansion algorithm is outlined.

```
   input: Set of faces $\{F_1, ..., F_K\}$, set of labels $\{P^1, ..., P^N\}$
 1 Start with an arbitrary labeling vector $\mathbf{M}$;
 2 success ← true;
 3 while success == true do
 4     success ← false;
 5     foreach label $P^\alpha \in \{P^1, ..., P^N\}$ do
 6         Find $\hat{\mathbf{M}} = \arg\min E(\mathbf{M}')$ among $\mathbf{M}'$ within one $\alpha$-expansion of $\mathbf{M}$;
 7         if $E(\hat{\mathbf{M}}) < E(\mathbf{M})$ then
 8             $\mathbf{M} \leftarrow \hat{\mathbf{M}}$;
 9             success ← true;
10         end
11     end
12 end
13 Return $\mathbf{M}$;
```

Algorithm 3.1: Basic structure of the α-expansion algorithm, as it is described in [5]

Boykov et al. call the steps 4 to 11 a *cycle*, the steps 6 to 10 an *iteration*. In every cycle, the algorithm iterates over all labels in a predefined or an arbitrary order. In every iteration, the goal is to find a labeling vector $\hat{\mathbf{M}}$, for which the energy is smaller than for the current labeling vector $\mathbf{M}$. Only when at least one better labeling vector was found during a cycle, the algorithm resumes with another cycle. If no better labeling vector can be found during a cycle, the current labeling vector $\mathbf{M}$ is returned.

With the Algorithm 3.1, we have a powerful tool to calculate a local minimum of the energy function shown in Equation 3.2. The remaining problem is finding the optimal α-expansion in step 6 for a given labeling vector $\mathbf{M}$. As a solution, Boykov et al.

propose a method using graph cuts [5]. In order to explain this approach, it is necessary to understand the principles of graph cuts. Therefore, we will provide the essential information about graph cuts, as it is also done in [5], now.

Graph Cuts. Let $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ be a weighted graph consisting of a set of vertexes $\mathcal{V}$ and a set of edges $\mathcal{E}$, so that every edge $e \in \mathcal{E}$ has a weight assigned. In order to perform a graph cut, it is important that exactly two different vertexes $v_\alpha, v_{\bar{\alpha}} \in \mathcal{V}$ are considered that are called *terminals*. A cut $\mathcal{C} \subset \mathcal{E}$ is a subset of edges so that no path exists between the two terminal vertexes v_α and $v_{\bar{\alpha}}$. Further, this subset has to be minimal, which means that the addition of a single edge would establish a path between the terminals. The result of a cut of a graph $\mathcal{G}$ is a graph $\mathcal{G}(\mathcal{C}) = \langle \mathcal{V}, \mathcal{E} - \mathcal{C} \rangle$. The cost of a graph cut, defined as $|\mathcal{C}|$, is simply the sum of the assigned weights of all the edges $e \in \mathcal{C}$. Let $\mathbf{C} = \{\mathcal{C} \mid \mathcal{C}$ is a cut of $\mathcal{G}$ separating v_α and $v_{\bar{\alpha}}\}$ be the set of all possible graph cuts separating the two terminals v_α and $v_{\bar{\alpha}}$. The *minimum cut* problem is finding the cut $\mathcal{C}_{min}$ = arg min $|\mathcal{C}|$ among all graph cuts $\mathcal{C} \in \mathbf{C}$. Boykov et al. showed and proved in [5] that finding the optimal α-expansion in step 6 of Algorithm 3.1 is equivalent to finding the minimum cut of an appropriately defined two-terminal graph. The layout of this graph will be explained in the following.

Constructing an appropriate graph. For the explanation how a graph appropriate to encode the layout of a particular mesh must look like, we will consider the small mesh illustrated in Figure 3.5. It consists of only four faces. Every face has a label assigned. In the figure, $F_i(P^j)$ denotes that the label P^j is assigned to the face F_i.

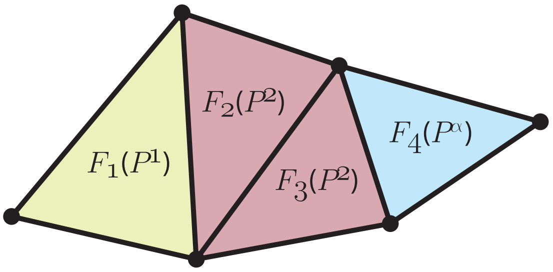


Figure 3.5: Example mesh with three different labels assigned to the triangles. $F_i(P^j)$ denotes that the label P^j is assigned to the face F_i. The label P^α refers to the current label of step 6 of Algorithm 3.1.

In Figure 3.6, the graph corresponding to the mesh in Figure 3.5 is illustrated. The layout of this graph is directly related to the current labeling vector $\mathbf{M}$ and the current label P^α. Therefore, the layout changes after every iteration, because the current label P^α changes. In the graph, there are different types of nodes. For every face in the mesh, there is a node representing this face in the graph. In Figure 3.6, these nodes are illustrated by the triangles denoted by F_1 to F_4. Further, there are two nodes representing the mentioned terminal nodes v_α and $v_{\bar{\alpha}}$. For every neighboring pair of faces that is labeled differently with respect to the current labeling vector $\mathbf{M}$, a further node is introduced. In [5], such a node is called an *auxiliary node*. In Figure 3.6, there are two auxiliary nodes denoted by a and b. Boykov et al. differentiate between two types of edges in the graph, called *t-links* and *n-links* [5]. Every node corresponding to a face is connected to both terminal nodes by one t-link each. A face F_i is therefore connected to the terminal node v_α via t-link $t^{v_\alpha}_{F_i}$ and to the terminal node $v_{\bar{\alpha}}$ via t-link $t^{v_{\bar{\alpha}}}_{F_i}$. Two neighboring and equally labeled faces F_i and F_j are connected by an n-link $e_{\{F_i,F_j\}}$. The introduction of an auxiliary node $a_{\{F_i,F_j\}}$ is a little bit more sophisticated. Consider two neighboring faces F_i and F_j that are differently labeled. Then, the introduced auxiliary node $a_{\{F_i,F_j\}}$ is connected via t-link $t^{v_{\bar{\alpha}}}_{a}$ to the terminal node $v_{\bar{\alpha}}$, via n-link $e_{\{F_i,a\}}$ to the face node F_i and via n-link $e_{\{a,F_j\}}$ to the face node F_j.

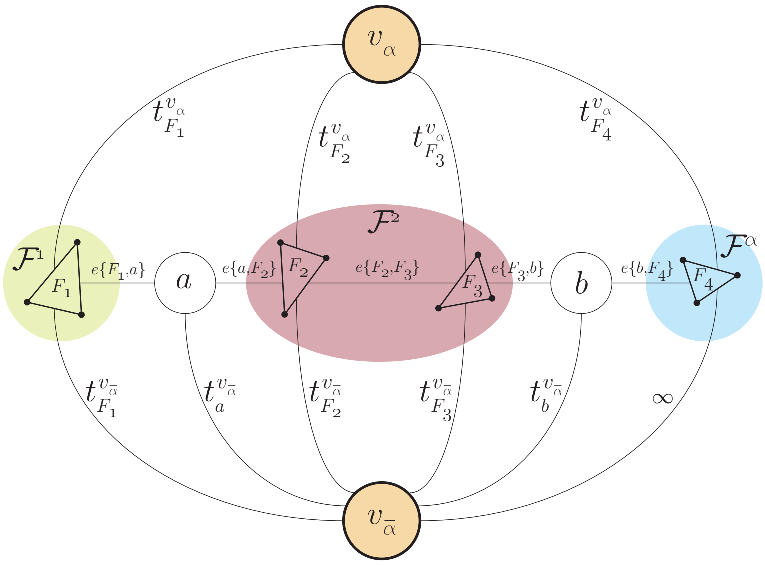
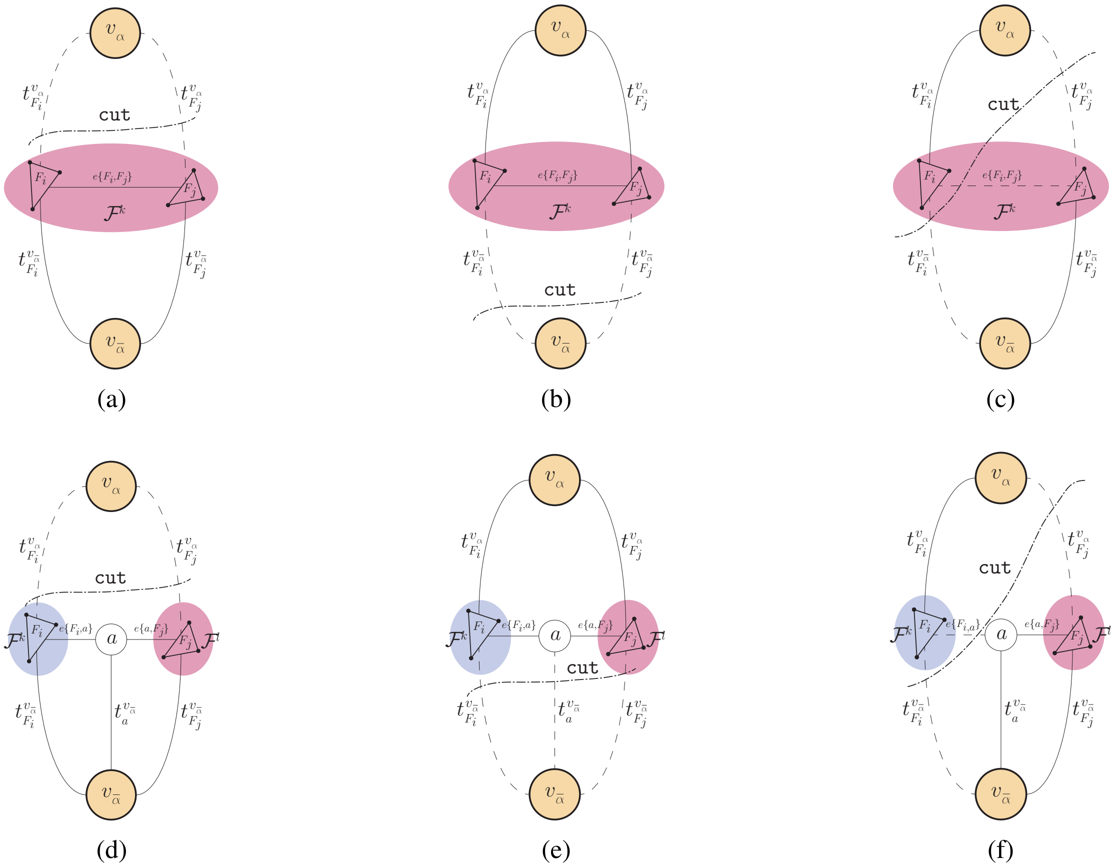


Figure 3.6: Graph that corresponds to the mesh shown in Figure 3.5. The layout is based on the graph shown in [5].

For two neighboring faces, the possible cuts are shown in Figure 3.7. In the top row of the figure, the possibilities to separate the terminal nodes for equally labeled faces are shown. In the bottom row, the valid cuts for differently labeled faces are shown. Obviously, for every face node F_i, one of the t-links $t^{v_\alpha}_{F_i}$ or $t^{v_{\bar{\alpha}}}_{F_i}$ has to be part of the cut $\mathcal{C}$ in order to receive a valid cut. A t-link $t^{v_\alpha}_{F_i} \in \mathcal{C}$ states that the face F_i gets the label P^α. A t-link $t^{v_{\bar{\alpha}}}_{F_i}$ states that the label of face F_i is not changed. The weights that are assigned to the edges of the graph are shown in Table 3.1. Boykov et al. show in their paper that the lowest energy labeling within a single α expansion move from the current labeling vector $\mathbf{M}$ is $\hat{\mathbf{M}} = \mathbf{M}^{\mathcal{C}}$, where $\mathcal{C}$ is the minimum cut on $\mathcal{G}_\alpha$ [5].



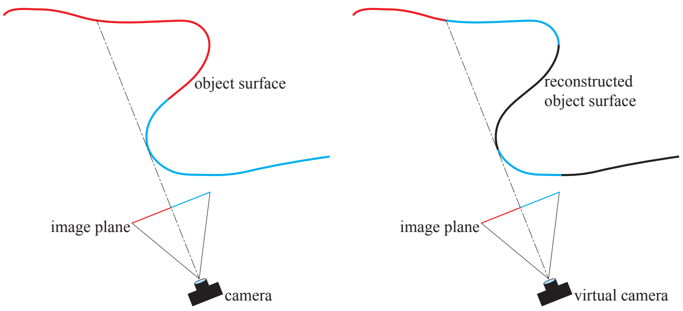
(a) (b) (c)

(d) (e) (f)

Figure 3.7: Possible cuts for equally labeled (a, b, c) and differently labeled faces (d, e, f). The illustrations are based on the ones shown in [5].

edge	**weight**	**for**
$t_{F_i}^{v_{\bar{\alpha}}}$	∞	$F_i \in \mathcal{F}^\alpha$
$t_{F_i}^{v_{\bar{\alpha}}}$	$w_i^{m_i}$	$F_i \notin \mathcal{F}^\alpha$
$t_{F_i}^{v_\alpha}$	w_i^α	$F_i \in \{F_1, ..., F_K\}$
$e_{\{F_i, a_{\{F_i,F_j\}}\}}$	$\lambda \cdot w_{i,j}^{m_i,\alpha}$	$\{F_i, F_j\} \in \mathcal{N}, m_i \neq m_j$
$e_{\{a_{\{F_i,F_j\}}, F_j\}}$	$\lambda \cdot w_{i,j}^{\alpha,m_j}$	
$t_a^{v_{\bar{\alpha}}}$	$\lambda \cdot w_{i,j}^{m_i,m_j}$	
$e_{\{F_i,F_j\}}$	$\lambda \cdot w_{i,j}^{m_i,\alpha}$	$\{F_i, F_j\} \in \mathcal{N}, m_i = m_j$

Table 3.1: Weights that are assigned to the edges for the minimum cut problem, as it is shown in [5].



(a) Exposure (b) Back-projection

Figure 3.8: Incorrectly back-projected image information when no occlusion detection is carried out. The taken image (a) is back-projected onto surface areas that to do not correspond to the image information (b).

3.1.4 Occlusion Detection

With the approach of Boykov et al. [5] that we explained in the last section, we have a powerful tool for (approximately) solving Equation 3.2 in order to come to a good labeling result. This is exactly the method for labeling that was proposed by Lempitsky and Ivanov in [18]. Their approach, however, gets into difficulties when complex geometry is considered. In their paper, they just show the approach using nearly convex items like a statue head and a model of the earth ball. In the fields of cultural heritage, however, often far more complex items have to be textured. A missing link in the approach of [18] is occlusion detection. Without occlusion detection, it is possible that wrong image material is back-projected onto the digitized model. This issue is shown in Figure 3.8 for a very simplistic scene.

The scene consists of an object that changes its color in the area that is invisible to the camera. In Figure 3.8a, the exposure of the photo is visualized. The image material in the left half of the photo corresponds to the red part of the object. In the right half of the photo, the image material corresponds to the blue part of the object. When no occlusion detection is used for the back-projection, no calculations are carried out whether there was an object in between the camera and the area of interest or not. Without occlusion detection, a particular face is textured by a photo when just the two following criteria are met: The face must be observed from the front and the projection of the face must be entirely inside the photo. If these are the only criteria that decide whether a particular face should be textured by a photo or not, the issue shown in Figure 3.8b can arise. In the figure, the blue image material of the photo is back-projected onto all the faces observed from the front and lying entirely inside the photo. We were first faced with the mentioned issues when we were labeling a model of a cubiculum of the Domitilla catacombs. However, we can not show images of this model for copyright reasons. What we can say is that the results were absolutely unacceptable because of the incorrectly textured faces. In the model, faces in the inside of the cubiculum received color information from photos that just contained image material of an exterior wall. Therefore, we introduce an accurate occlusion detection method into the proposed approach of [18].

Occlusion detection is necessary to decide whether or not parts of a photo correspond to a particular surface area of the digitized object. There are different methods for occlusion detection available. Callieri et al. calculate depth masks for the photos [6]. The occlusion detection for a particular triangle-photo-pair is then simply performed by projection of the triangle into the photo. If the depth values of the triangle are bigger than the values in the depth mask, there is an occlusion and the photo can not be used for texturing of the triangle. The accurateness of this approach is directly linked to the resolution of the depth masks, which is usually the same as the resolution of the photos. When texturing very dense meshes or when a triangle is observed under a very flat angle, it can happen that two neighboring vertexes are mapped onto the same pixel of a particular depth map. It is therefore possible that a neighboring vertex overwrites the depth value of a particular vertex for this pixel because of its smaller distance to the viewpoint. This depth value would indicate an occlusion of the vertex and therefore of the triangle in the photo. To be independent of the resolution of the photos, we use a conservative geometric approach.

Consider a triangle F_i of the model. The most naive way to decide whether a particular photo P^j really contains the image material needed to texture F_i, and not the colors of an occluder, would be to iterate over all the other triangles. For every of these triangles, one would have to evaluate if it is between the position of the camera corresponding to P^j and the triangle F_i. If the triangle is in between, it occludes F_i and therefore P^j can not be used to texture F_i. This approach sounds easy, but the complexity is unacceptable. To do such an occlusion detection for the whole set of photos $\{P^1, ..., P^N\}$, one would need to iterate over all the photos. Then, for each of these photos, an iteration over the whole set of triangles $\{F_j \in \{F_1, ..., F_K\}, F_j \neq F_i\}$ would be necessary for every single triangle F_i the occlusion detection is carried out for. The overall complexity would be therefore $\mathcal{O}(NK^2)$.

To reduce the complexity, we introduce an octree into the occlusion detection algorithm to reach a complexity of $\mathcal{O}(NK \log{(K)})$. This octree is used to quickly filter out all the triangles that do not come into consideration for an occlusion of a particular triangle. The octree extents directly correlate to the axis aligned bounding box of the model. The triangles of the model are not grouped together, but are directly inserted into the octree one by one. We introduced a parameter into our application that acts as upper threshold for the cells of the octree. Only when all three side lengths of a particular cell are smaller than this threshold, no further division into octants is performed. These small cells are the leaves of the octree and store the triangle indexes of those triangles that lie (partially) inside. In our performance tests we found that approximately the diameter of an average triangle of the model is a good value for the upper threshold (see Section 5.2.1.3 for details). A triangle is not always positioned in space, so that it fits entirely into one of the leaf cells. For those cases, more than just one leaf cell holds the index of a particular triangle.

Our occlusion detection algorithm is based on the view frustum culling algorithm proposed in [21]. For the actual occlusion detection, we introduce two geometrical objects. The first of these objects we called *triangle sight frustum.* Such a triangle sight frustum is used for the calculation whether a particular triangle of the mesh is occluded by another triangle concerning a particular photo. The second of the introduced geometrical objects we called *edge sight triangle*, which is used in a similar way, but is only used for the detection of an occlusion of an edge of the mesh. These geometrical objects and their usage will be explained in the following sections.

3.1.4.1 Triangle sight frustum

The triangle sight frustum is a geometrical object defined by four points in 3D space. Three of these points are defined by the vertexes of the triangle for which an occlusion shall be calculated. The fourth point of the triangle sight frustum is defined by the position of the camera that corresponds to the photo under consideration. The four points span four triangles. The result is a volume that looks more like a pyramid than a frustum. Because of the analogy to a conventional camera view frustum, we chose the word "triangle sight frustum" though. In Figure 3.9 such a triangle sight frustum is illustrated.

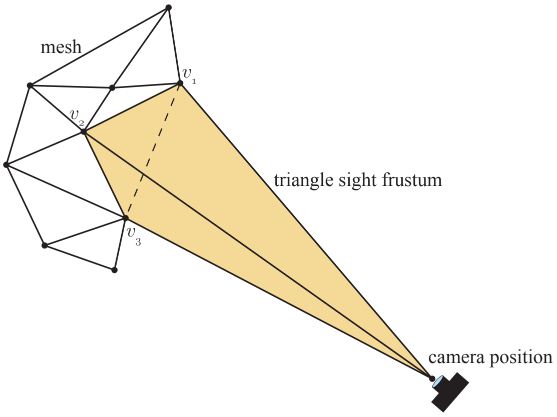


Figure 3.9: Triangle sight frustum used for occlusion detection concerning a single triangle of the mesh.

The principle of the triangle sight frustum is simple. Let F_i be a triangle of the model for which an occlusion in the photo P^j shell be detected and let T_i^j be the resulting triangle sight frustum. Then, F_i is occluded in P^j if any other triangle $F_k \neq F_i$ of the mesh intersects with T_i^j. An intersection in this context only exists if at least a part of the occluding triangle F_k is inside of T_i^j. If any triangle $F_k \neq F_i$ only touches one of the triangles of T_i^j, no intersection occurs. If already a touch of the triangles of T_i^j would be classified as an intersection, the neighbor triangles of F_i would always occlude F_i, since they share an edge with F_i and therefore they always intersect with T_i^j.

In order to rapidly find the faces of the model that even come into consideration for an occlusion of F_i in P^j, the octree is taken into account. First, T_i^j and the box representing the whole octree are tested for intersection. In principle, this first test can be omitted, since the octree measurements are defined by the axis aligned bounding box of the model and therefore there is always an intersection between T_i^j and the box representing the whole octree. After this first intersection test, T_i^j and the boxes representing the eight child nodes of the root of the octree are tested for intersection. Depending on the structure of the model, this is the first part of our occlusion detection algorithm where a bigger part of the set of triangles can be filtered out immediately by a simple intersection test between T_i^j and some axis aligned boxes. In contrast to the mentioned naive approach, where an intersection test is needed for every triangle $F_k \neq F_i$, here we can exclude a bulk of triangles just by testing T_i^j and some axis aligned bounding boxes for intersection. If T_i^j intersects with a cell of the octree, T_i^j and all the cells representing the child nodes of the current cell are tested for intersection. This process is continued in a recursive way until either all parts of the octree are filtered out, and no occlusion is detected, or some cells representing the leaf nodes of the octree storing the triangle indexes remain. Before we will show how the intersection test between T_i^j and the triangles in the remaining octree cells is performed, we will show the very efficient intersection test for the axis aligned boxes representing the cells of the octree.

We define the normal vectors of the four triangles of the triangle sight frustum to point into the inside of the frustum. Then, a point is inside the frustum if it is in front of all the triangles with respect to their normals. Therefore, if this point is behind just one of the triangles, it is definitely outside the whole frustum. When we do not exactly want to know if the point coincides with one of the triangles, but only whether it is in front or behind, the triangles can be treated as planes. In our case, we do not have points for testing, but axis aligned boxes. In order to test such a box and a plane for intersection, it would be therefore sufficient to find the vertex of the box whose signed distance to the plane is maximum. This vertex is called the *positive vertex*, or just p-vertex [21]. If the p-vertex of the box is behind the plane, the whole box is behind the plane. The vertex, whose signed distance to the plane is minimum, is the *negative vertex*, or just n-vertex [21].

In Figure 3.10, the possible relations of a box and a plane are illustrated. The green box is completely behind the plane, since its positive vertex is behind the plane. The yellow box gets intersected by the plane, since its positive vertex is in front of it, but the negative vertex is behind it. Because both the positive and the negative vertex are in front of the plane, the red box is completely in front of it.

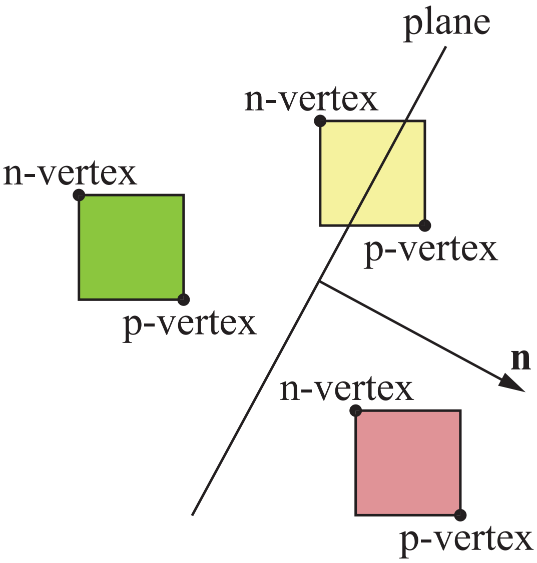


Figure 3.10: Possible relations of a box and a plane. The green box is behind the plane, the yellow box gets intersected and the red box is entirely in front of the plane. The illustration is based on the one shown in [21].

In order to test a triangle sight frustum and an axis aligned box for intersection, we can therefore just find the p-vertex of the box and test it with the planes defined by the triangles of the frustum for intersection. If the p-vertex is behind just one of the planes, the box is definitely outside. If the p-vertex is in front of all the planes, the box is at least partly inside of the frustum. Using the n-vertex, we could further evaluate if the box is partly or entirely inside the triangle sight frustum. Since in our case it is unimportant whether a box representing a cell of the octree is entirely or only partly inside the triangle sight frustum, we only need the p-vertex. The calculation of the p-vertex, given the two vertexes defining the axis aligned box and the normal of the plane, is shown in Algorithm 3.2.

When all parts of the octree that do not intersect with T_i^j are filtered out, there may remain some cells of the octree representing some leaf nodes. When no parts remain, there is no occlusion and F_i can be textured by P^j. The triangle indexes stored in the remaining cells correspond to triangles of the mesh for which an intersection with T_i^j is possible. In order to perform a fast intersection test for all these possibly occluding triangles, we first test if any vertex of these triangles is inside of T_i^j. If this is the case, F_i is definitely occluded and can not be textured by P^j. If no vertex is inside, an occlusion is still possible. A simple test between the vertexes of the triangle under consideration

```
Input: Axis aligned box box, plane normal $\boldsymbol{n}$
$\boldsymbol{p} \leftarrow (box.x_{min}, box.y_{min}, box.z_{min})$;
if $\boldsymbol{n}.x \geq 0$ then
    $\boldsymbol{p}.x \leftarrow box.x_{max}$;
end
if $\boldsymbol{n}.y \geq 0$ then
    $\boldsymbol{p}.y \leftarrow box.y_{max}$;
end
if $\boldsymbol{n}.z \geq 0$ then
    $\boldsymbol{p}.z \leftarrow box.z_{max}$;
end
Return $\boldsymbol{p}$;
```

Algorithm 3.2: Calculation of the positive vertex $\boldsymbol{p}$ of an axis aligned box with respect to a plane with normal vector $\boldsymbol{n}$.

and the planes defined by the triangles of T_i^j is insufficient. Only a real triangle-triangle intersection test can give insight into the situation then. For this we use the approach of Möller [26]. If there is an intersection between one of the triangles T_i^j and any triangle of the remaining octree cells, an occlusion of F_i exists and P^j can not be used to texture the triangle F_i. If there is no intersection, there is no occlusion and P_j contains the correct image material to texture F_i.

The whole occlusion detection algorithm for a particular triangle F_i and a photo P^j is shown in Algorithm 3.3.

Occlusion detection using the triangle sight frustum is needed when calculating $w_i^{m_i}$ in Equation 3.2. This is the cost to texture the triangle F_i with the corresponding image region in photo P^{m_i}. When the occlusion detection algorithm detects an occlusion of F_i in P^{m_i}, the maximum value for $w_i^{m_i}$ has to be estimated, indicating that P^{m_i} can not be used to texture F_i.

3.1.4.2 Edge sight triangle

While a triangle sight frustum is used to detect an occlusion of a triangle, an edge sight triangle is used to detect the occlusion of an edge. Consider an edge E_{ij} of the mesh that is shared by the two triangles F_i and F_j, and a photo P^k used for texturing. Then, the corresponding edge sight triangle R_{ij}^k is defined by the two vertexes that correspond to the edge E_{ij}, and the position of the camera corresponding to P^k. In Figure 3.11 an edge sight triangle is illustrated.

In order to detect an occlusion of E_{ij} in P^k, the corresponding edge sight triangle R_{ij}^k is tested for intersection with all the triangles of the mesh that come into consideration for an intersection. To quickly filter out all the triangles that do definitely not intersect

```
Input: Octree node node, photo $P^j$, triangle $F_i$ for which an occlusion in $P_j$ shall
       be detected, triangle sight frustum $T_i^j$
if $T_i^j$ intersects $node.boundingBox$ then
    if node is a leaf node then
        foreach triangle index k stored in node do
            if any vertex $v$ of triangle $t_k$ is inside $T_i^j$ then
                Return OCCLUSION;
            end
            if triangle $t_k$ intersects with any triangle of $T_i^j$ then
                Return OCCLUSION;
            end
        end
        Return NO_OCCLUSION;
    else
        foreach child node $c$ of node do
            result ← Return value of Algorithm 3.3 using $c$, $P^j$, $F_i$ and $T_i^j$ as
            arguments;
            if result $=$ OCCLUSION then
                Return OCCLUSION;
            end
        end
    end
else
    Return NO_OCCLUSION;
end
```

Algorithm 3.3: Our algorithm for detection of occluded triangles using an octree.

with R_{ij}^k, the octree is used again. This is done in a similar way to the approach that was used for the triangle sight frustum. For the intersection test between R_{ij}^k and an axis aligned box representing a node of the octree, the p-vertex of the box is calculated. If the p-vertex is behind R_{ij}^k with respect to its normal, there is no intersection. If the p-vertex is in front of R_{ij}^k, a second test has to performed to decide whether also the n-vertex is in front of R_{ij}^k. When the p-vertex and the n-vertex are lying on different sides of R_{ij}^k, the intersection tests need to be continued with all the boxes representing the octants of the current node of the octree. When the leaf nodes of the octree are reached, a full triangle-triangle intersection test is performed using the approach of Möller [26].

In contrast to a triangle sight frustum $T_i^{m_i}$ that is used in the calculation of the data cost $w_i^{m_i}$ for a triangle F_i and a photo P^{m_i}, there are two edge sight triangles $R_{ij}^{m_i}$ and $R_{ij}^{m_j}$ needed in the calculation of the smoothness cost $w_{i,j}^{m_i,m_j}$ for two adjoining triangles

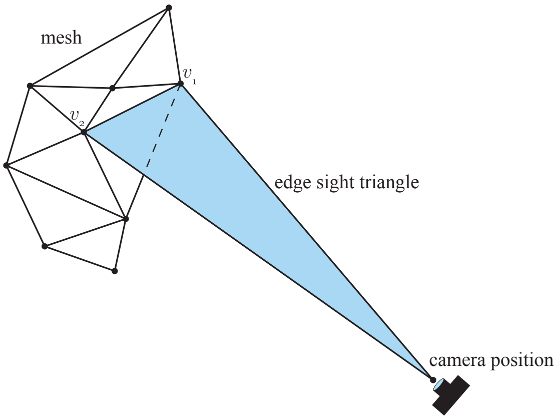


Figure 3.11: Edge sight triangle used for occlusion detection concerning a single edge of the mesh.

F_i and F_j and two photos P^{m_i} and P^{m_j}. When just one of these edge sight triangles intersects with any triangle of the mesh, the maximum value for $w_{i,j}^{m_i,m_j}$ has to estimated, indicating that the color difference between the adjoining faces is maximal.

Discussion. With our approach for occlusion detection, it is now possible to map a set of registered photos to very complex objects that are far away from convexity. The labeling process is fully automatic, only the parameter λ in Equation 3.2 has to be chosen to either prefer high quality of image material or smooth transitions between regions that receive color from different photos.

Using the octree for fast occlusion detection, we also got a satisfactory labeling result for the Domitilla cubiculum model which was inaccurately labeled without occlusion detection. Unfortunately, we can not show an image of the model for copyright reasons.

A remaining problem in the approach of Lempitsky and Ivanov [18] are the mentioned camera registration errors that lead to visible misalignments of the projected photos in the textured model.

3.1.5 Shift Vectors

To account for the camera registration errors, we consider the approach of Gal et al. [13], which was also described in Chapter 2. In the approach, shift vectors are introduced into the labeling procedure. Instead of just using the photos themselves as labels, each label is a tuple consisting of a particular photo and a shift vector. A triangle with such an assigned label is then textured by the image region that results from projecting the triangle into the photo, and shifting the projection along the shift vector inside the photo.

In our labeling experiments using this approach, we got satisfactory as well as catas-

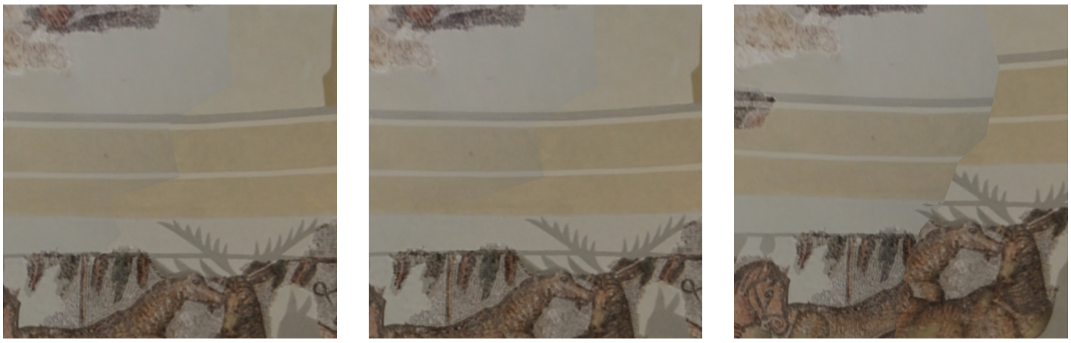

(a) (b) (c)

Figure 3.12: Illustration of the results using the labeling approach of Gal et al. [18] with introduced shift vectors to account for camera registration errors. (a) Without shift vectors. (b) 2 levels (3 pixels). (c) 6 levels (63 pixels).

trophic results. This is shown in Figure 3.12. The used model is a small section of the Centcelles cupola. The left side is textured by a different photo than the right side. In Figure 3.12a, the result of the labeling without shift vectors is visualized. In the figure, small misalignments between the photos are visible. Especially the horizontal lines do not perfectly adjoin. For the labeling result shown in Figure 3.12b, the approach of Gal et al. with two levels of the image pyramid of the photos was used. With two levels, a maximum camera registration error of 3 pixels can be accounted for. Obviously, the labeling result is better than without shift vectors. What we observed when using the approach of Gal et al., was that the labeling result is very sensitive to the choice of the maximum number of levels of the image pyramids. When we labeled the model using the approach of Gal et al. with six levels of the image pyramid of the photos, the result was not satisfactory. This issue is shown in Figure 3.12c. The misalignments of the photos became even bigger. When using the approach of Gal et al., it is therefore very important that the number of levels directly correlates to the maximum magnitude of camera registration errors.

Beside the problem with the adequate choice of number of levels of the image pyramids of the photos, the approach of Gal et al. is only suitable for very small models with just a few photos. The reason for this is the exploding number of labels. Already in the first iteration, the number of labels is nine times the number of input photos. In the absolute worst case, the number of labels then increases by a factor of 9 for every following iteration. When labeling a model with a huge number of triangles and also a huge number of photos, the labeling procedure can take a very long time. In our labeling experiments, we labeled the Centcelles cupola model consisting of more than 4 million triangles in conjunction with 70 input photos. We chose a maximum level number of 5 to account for a maximum camera registration error of 63 pixels with respect to the input photos. The labeling procedure took about 5 days, and then the result was not satisfactory anyway because of an inappropriate choice for the maximum number of levels.

The approach of Gal et al. is based on the assumption, that camera registration errors are often just translational. Therefore, it would be sufficient to shift the projection of a particular triangle inside the photos to find its corresponding image material needed for texturing. Our labeling tests using this approach showed us that this is obviously often not sufficient. An option could be to assign the shift vectors not to the triangles, but only to the edges or to the vertexes of the mesh. When the projections of the vertexes of a particular triangle would be shifted by different vectors, this would also allow scaling and rotating of the triangle.

3.1.6 Implementation

Our MosaicBuilder is implemented in C++. For all image operations, we use the OpenCV library in the version 2.3. In Figure 3.13, an overview of the labeling procedure as it is performed by our labeling application is given. At the moment, our application only supports OBJ as the format for triangulated models. Both the input model and the labeled output model are restricted to the OBJ format. The photos can be in any format that is also supported by the OpenCV library. When the photos are read in, they are undistorted and stored in the directory where also the labeled model will be stored.

After the model is loaded and the undistorted photos are generated, the normal vectors of the triangles of the model are calculated. It is important that the normal vectors are calculated before the octree is constructed, because they are needed for the fast evaluation of intersections between the triangles and the cells of the octree.

The octree is constructed in a way so that only the indexes of the triangles are stored in the leaf nodes. For fast insertion of a triangle into the octree, at first the axis aligned bounding box of the triangle and the cells of the octree are tested for intersection. This test is based only on comparisons and requires no further calculations. Only when the bounding box of the triangle and a particular cell intersect, the triangle itself and the cell are tested for intersection.

When the octree is constructed, the actual labeling procedure via Markov Random Field energy optimization is performed. The octree is used for occlusion detection, as it is described in Section 3.1.

For the actual labeling step, we use the gco-v3.0 library developed by Olga Veksler and Andrew Delong [5] [17] [4]. This library implements the α-expansion Graph Cuts algorithm that we described in an earlier section.

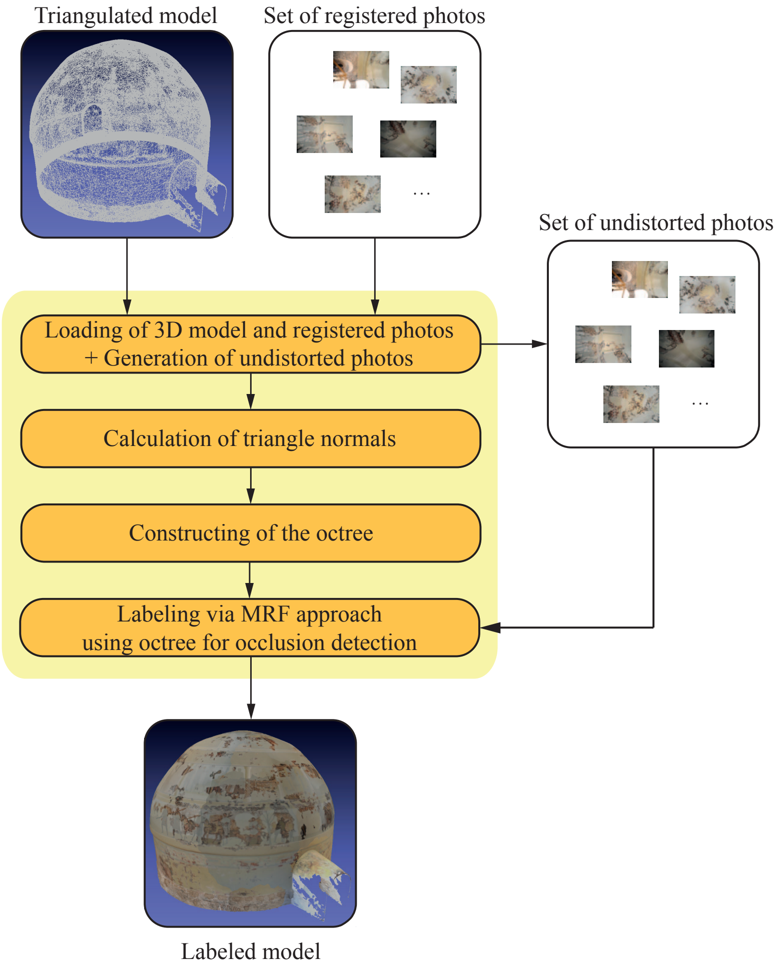

Triangulated model
Set of registered photos
...
Set of undistorted photos
...
Loading of 3D model and registered photos
+ Generation of undistorted photos
Calculation of triangle normals
Constructing of the octree
Labeling via MRF approach
using octree for occlusion detection
Labeled model

Figure 3.13: Overview of the labeling procedure as it is performed by our MosaicBuilder.

3.1.7 Issues

Regardless of which value is chosen for λ in Equation 3.2, or if occlusion detection is performed or not, in practice there remain seams in the model where regions that get color information from different photos adjoin. In Figure 3.14, those seams are shown for the Centcelles cupola model that was textured using our *MosaicBuilder*. As already mentioned, these seams arise because of the different lighting situations during the exposure of the photos. For a high-quality model, those seams have to be handled. Because a manual editing as it was done by the graphic artist until this thesis was in progress is unacceptable, we will show our automatic approach for leveling in Section 3.2.

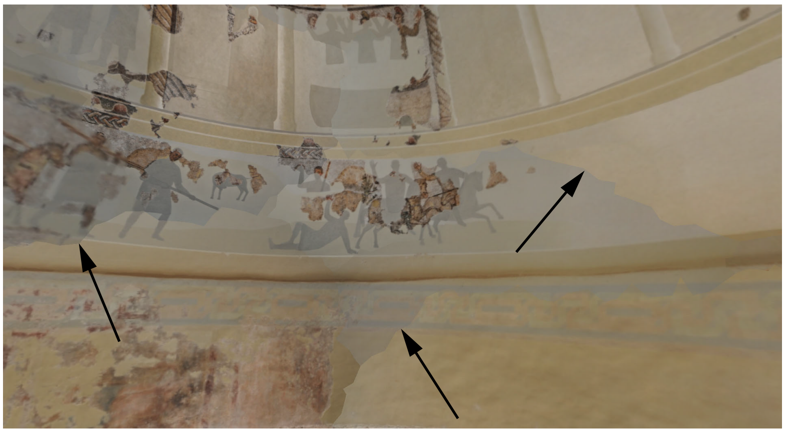


Figure 3.14: Remaining seams in the model that was textured using our *MosaicBuilder*.

3.2 Leveling – PoissonLeveler

3.2.1 Leveling overview

The result of the labeling procedure described in Section 3.1 is a labeling vector **M** that defines a mapping of the set of triangles of the model onto the set of photos used for texturing. If the labeling approach of Gal et al. [13] is used, every label further contains a shift vector that tells how to shift the projected face inside the assigned photo. Whatever method is used, we now have a set of connected components $\{C_1, ..., C_T\}$, where each component consists of a set of connected triangles that receive their color information from one particular photo. Assuming that the photos are continuous functions for each color channel (R, G and B), the mapping of just one color channel of the photos onto the 3D model with respect to the labeling vector **M** results in a piecewise continuous function f on the mesh M. Only at the edges where two different components C_i and C_j adjoin, there are points of discontinuity. What we are looking for in the leveling procedure is a piecewise smooth leveling function g that meets the two following criteria:

1. The magnitude of the gradients of the leveling function g is minimal.
2. The jumps of the leveling function g are equal to the negative jumps of f.

The first criteria is important to preserve the high frequencies of the function f. The second criteria guarantees that the points of discontinuity are smoothed out at the edges where two connected components C_i and C_j adjoin. For the leveling using photos with three color channels (R, G and B), three separate leveling functions have to be calculated. In the following, we will describe the leveling only for a single channel.

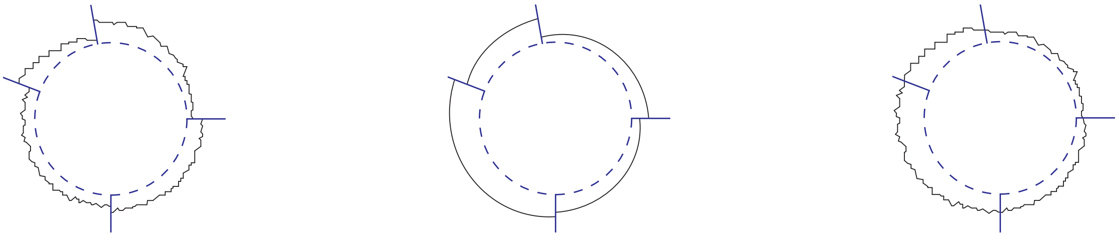

(a) (b) (c)

Figure 3.15: Illustration of the leveling procedure on a circle for a single color channel. Intensity values are encoded by the distance perpendicular to the circle. (a) The original function f having points of discontinuity (indicated by the radial line segments). (b) The calculated piecewise smooth leveling function g. (c) Sum of f and g. Discontinuities are smoothed out while at the same time high frequencies are preserved. The images are based on the ones shown in [18].

In Figure 3.15, the principle of the leveling procedure for a single color channel is illustrated.

Our approach for the leveling process is based on the one of Lempitsky and Ivanov [18]. They calculate the leveling function g only at the vertexes of the mesh and then interpolate the function values along the edges and triangle surfaces. For the explanation of their approach, they consider the set $\mathcal{M}$ containing all the (i, j)-pairs prescribing that at least one triangle adjacent to the vertex V_i is part of the connected component C_j. For each of these (i, j)-pairs, the corresponding leveling function value g_i^j is computed. As an example, consider a vertex V_1 that is part of three connected components C_1, C_2 and C_3. By definition, the connected components are textured by different photos. When the vertex V_1 is projected into these photos, the intensity values of the pixels V_1 is projected into, may differ. These intensity values define the original texture function f. At V_1 we have therefore three different function values. Similar to the notations in [18], we define f_i^j to be the original texture function value at the vertex V_i for the connected component C_j. In our example we have therefore the function values f_1^1, f_1^2 and f_1^3 at the vertex V_1. The differences between these function values then lead to the point of discontinuity at V_1. So three leveling function values g_1^1, g_1^2 and g_1^3 have to be calculated at the vertex V_1. These different leveling function values are necessary to smooth out the discontinuities at V_1.

Similar to [18], we denote $\mathcal{L}$ to be the set of (i, j)-pairs prescribing that in the mesh there is an edge E_{ij} formed by the vertexes V_i and V_j.

Now the leveling function g, computed at the vertexes of the mesh, can be approximated by the minimization of the following least-squares energy function:

$$\sum_{\substack{(i_1,j)\in\mathcal{M}\\(i_2,j)\in\mathcal{M}\\(i_1,i_2)\in\mathcal{L}}} (g^j_{i_1} - g^j_{i_2})^2 + \lambda \sum_{\substack{(i,j_1)\in\mathcal{M}\\(i,j_2)\in\mathcal{M}}} (g^{j_1}_i - g^{j_2}_i - (f^{j_2}_i - f^{j_1}_i))^2 \tag{3.16}$$

The first term of 3.16 approximates the first condition we demanded of the leveling function g. This condition is the minimality of the magnitude of the gradients. The first term corresponds to all the edges $E_{i_1 i_2}$ in the mesh whose vertexes V_{i_1} and V_{i_2} are part of the same connected component C_j.

The second term of 3.16 approximates the second demanded condition of the leveling function g. To ensure smooth transitions at the points of discontinuity, the jumps of g need to be the negative jumps of f. The second term corresponds to all the vertexes V_i in the mesh, where two connected components C_{j_1} and C_{j_2} adjoin.

Lempitsky and Ivanov recommend a large value for the parameter λ (e.g.100), since the second term is a hard constraint [18]. Further, they propose the usage of a sparse solver in order to calculate appropriate values for the variables g^j_i. We also use a sparse solver in our application. The approach that we use to solve the least-squares problem is explained in the next section by the means of an example.

3.2.2 Solving the least squares problem

Consider the little mesh shown in Figure 3.16. It consists of four triangles. The left two

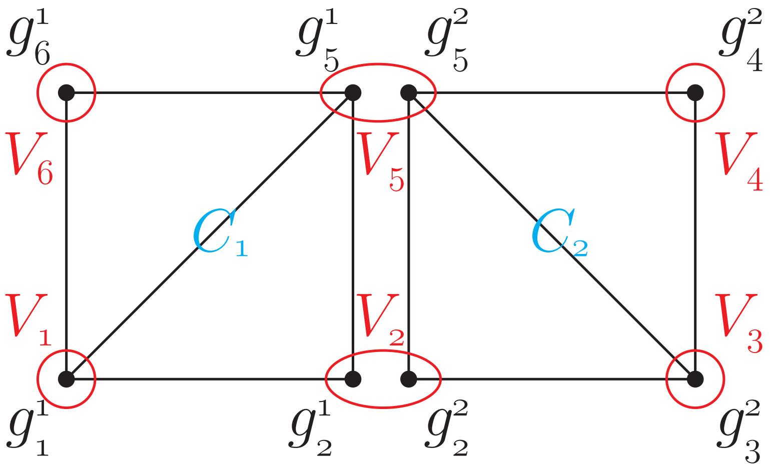


Figure 3.16: Example mesh with two connected components C_1 and C_2 to explain the leveling procedure. All the values g^j_i correspond to the computed leveling function values at the vertexes V_i.

triangles are textured by one photo, and therefore they are part of the same connected component C_1. The right two triangles are textured by another photo and are therefore part of another connected component than the left two triangles. In the example, this connected component is denoted by C_2. In the mesh, there are six vertexes V_1 to V_6. The vertexes are highlighted by red circles. The black dots correspond to the leveling function values g_i^j that are computed by solving the least squares problem. Note that the two edges in the illustration between V_2 and V_5 correspond to just one physical edge in the example mesh. C_1 and C_2 are therefore adjacent areas. For explanation reasons there are two edges drawn in, because in the least squares problem the physical edge is considered two times, one time for C_1 and one time for C_2. The fully expanded least-squares problem for the example mesh in Figure 3.16 is:

$$\begin{gathered}(g_1^1 - g_2^1)^2 + (g_2^1 - g_5^1)^2 + (g_5^1 - g_6^1)^2 + (g_6^1 - g_1^1)^2 + (g_1^1 - g_5^1)^2 + \\ (g_2^2 - g_3^2)^2 + (g_3^2 - g_4^2)^2 + (g_4^2 - g_5^2)^2 + (g_5^2 - g_2^2)^2 + (g_3^2 - g_5^2)^2 + \\ \lambda[(g_2^1 - g_2^2 - (f_2^2 - f_2^1))^2 + (g_5^1 - g_5^2 - (f_5^2 - f_5^1))^2]\end{gathered} \tag{3.17}$$

Now the question is how this term can be minimized. To do this, we will first take a look at the definition of a least squares problem. All the definitions are taken from [22]. In a least squares problem, one tries to find $\boldsymbol{g}^*$, which is a local minimizer for $F(\boldsymbol{g}) = \frac{1}{2}\sum_{i=1}^{m}(f_i(\boldsymbol{g}))^2 = \frac{1}{2}\|\boldsymbol{f}(\boldsymbol{g})\| = \frac{1}{2}\boldsymbol{f}(\boldsymbol{g})^\top \boldsymbol{f}(\boldsymbol{g})$, where $\boldsymbol{f} : \mathbb{R}^n \mapsto \mathbb{R}^m$, is a given vector function, and $m \geq n$. In particular, it is a linear least squares problem if the vector function $\boldsymbol{f}$ can be written as $\boldsymbol{f}(\boldsymbol{g}) = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{g}$, where the vector $\boldsymbol{b} \in \mathbb{R}^m$ and matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ are given.

For the leveling procedure, we have to solve a linear least squares problem, because the term shown in 3.17 can be written as $\boldsymbol{f}(\boldsymbol{g})^\top \boldsymbol{f}(\boldsymbol{g})$ with

$$\boldsymbol{f}(\boldsymbol{g}) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \sqrt{\lambda}(f_2^1 - f_2^2) \\ \sqrt{\lambda}(f_5^1 - f_5^2) \end{pmatrix} - \begin{pmatrix} -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & -\sqrt{\lambda} & \sqrt{\lambda} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \sqrt{\lambda} & -\sqrt{\lambda} & 0 \end{pmatrix} \begin{pmatrix} g_1^1 \\ g_2^1 \\ g_2^2 \\ g_3^2 \\ g_4^2 \\ g_5^2 \\ g_5^1 \\ g_6^1 \end{pmatrix} \tag{3.18}$$

The solution of the leveling procedure is therefore just finding an appropriate $\boldsymbol{g}^*$ satisfying $\boldsymbol{A}\boldsymbol{g}^* \simeq \boldsymbol{b}$. This linear equation system can be solved using standard approaches like QR factorization. As it was also proposed by Lempitsky and Ivanov in [18], the usage of a sparse solver is highly recommended. Naturally, the memory consumption of the matrix $\boldsymbol{A}$ increases quadratically with the number of vertexes in the mesh. The matrix $\boldsymbol{A}$ corresponding to a mesh with a vertex count of $300{,}000$ would have at least $300{,}000^2$ entries. Using double precision (64 bit) for storing the elements, this would lead to a memory consumption of 670.55 GB.

Discussion With the proposed leveling method, an adequate leveling function can be calculated up to an additive constant. This additive constant can e.g. be set to an appropriate value to maintain the mean gray value of the current color channel. One problem that we observed with the leveling approach of Lempitsky and Ivanov is the divergence of the leveled color values. When the leveling procedure leads to a range for the color values of e.g. $[-0.37, 1.67]$, there is no additive constant that would avoid clamping. Therefore, we adapted the least squares problem of Lempitsky and Ivanov.

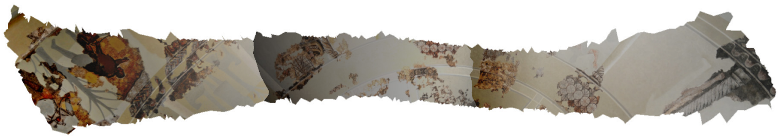
3.2.3 Keeping the color values in range

Because of the shown issues, a strategy is necessary to avoid the divergence of the leveled color values. We decided to adapt the least-squares problem shown in 3.16. A disadvantage of this least-squares problem is the absence of a term that penalizes big leveling function values. It only places importance on small differences between adjacent g_i^j and small color differences between adjoining connected components C_i.

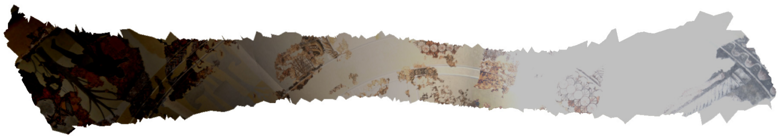
Therefore, we introduce a new term into the least-squares problem so that big leveling function values are penalized. This term can not fully avoid that the leveled color values exceed the valid range, but we observed a much better behavior using the new term. Our adapted least-squares problem is shown in 3.19.

$$\sum_{\substack{(i_1,j)\in\mathcal{M}\\(i_2,j)\in\mathcal{M}\\(i_1,i_2)\in\mathcal{L}}} (g_{i_1}^j - g_{i_2}^j)^2 + \lambda \sum_{\substack{(i,j_1)\in\mathcal{M}\\(i,j_2)\in\mathcal{M}}} (g_i^{j_1} - g_i^{j_2} - (f_i^{j_2} - f_i^{j_1}))^2 + \mu \sum_{(i,j)\in\mathcal{M}} (g_i^j)^2 \tag{3.19}$$

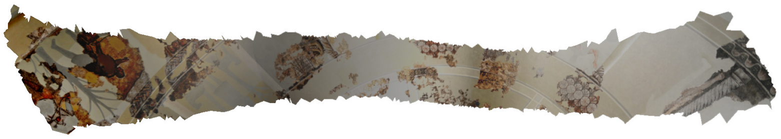
Discussion We tested the new term extensively in our leveling experiments. We observed that the least-squares problem reacts very sensitive to the parameter μ that controls the contribution of the third term. When a value of 100.0 was chosen for the parameter λ, a value of 0.01 for μ already brought considerable results. When the value for μ was chosen too high, the whole leveling procedure completely failed, and all the g_i^j got a value of 0.0. Nonetheless, when using the third term carefully, and an appropriate value is chosen for μ, our adapted least-squares problem can deliver better results than the original approach of Lempitksky and Ivanov proposed in [18]. In Figure



(a)



(b)



(c)

Figure 3.17: Comparison of the results of different leveling approaches using a small part of the Centcelles cupola model. (a) Originally textured model. (b) Approach of Lempitsky and Ivanov [18]. (c) Our approach with introduced third term into the least-squares problem of [18] for penalizing of too big leveling function values.

3.17, our approach using the third term for penalizing of too big leveling function values is demonstrated. The used model is a small part of the Centcelles cupola textured by three different photos. Relative to the model, the intensity values in each photo increase from the left to the right side. Originally, we had the problem of diverging color values in a model of the Domitilla catacombs. For copyright reasons, we can not show the mentioned issues using screenshots of this model. Therefore, we adapted the photos of the Centcelles cupola to deliver the same results.

Note the big loss of contrast on the left and right side of the model when there is no penalizing of big function values (Figure 3.17b). In contrast, when our third term is introduced into the least-squares problem, and big leveling function values are penalized, the seams are smoothed out while at the same time the contrast of the overall texture is not lost (Figure 3.17c).

The reason for the divergence of the color values, as it is shown in Figure 3.17b, is the original texture function f. We observed a diverging behavior when f had nearly the structure of a sawtooth function (see Figure 3.17a) so that the image intensity increases along each of the adjoining connected components C_i.

In Table 3.2, the ranges of the intensity values before and after the leveling procedure for the model shown in Figure 3.17 are shown for each color channel. The values correspond to normalized intensity values in RGB color space. While the values greatly

	Original range	After approach of [18]	After our approach
Red channel	$[0, 0.976471]$	$[-0.449405, 1.28369]$	$[-0.00177239, 0.963588]$
Green channel	$[0, 0.960784]$	$[-0.467519, 1.29606]$	$[-0.00133750, 0.943179]$
Blue channel	$[0, 0.929412]$	$[-0.456940, 1.34194]$	$[-0.00172197, 0.924188]$

Table 3.2: Color ranges before and after the leveling procedure using the approach proposed in [18] and using our approach with the introduced third term for penalizing of big leveling function values.

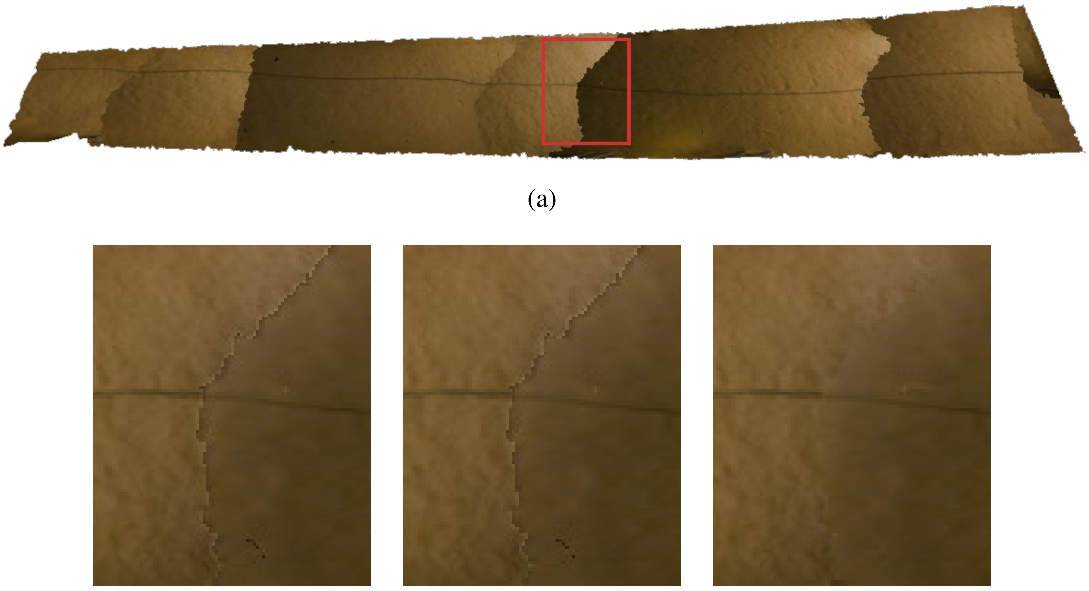
leave the valid range $[0,\ 1]$ when doing the leveling procedure by solving the least-squares problem without the third term for penalizing of big leveling function values, they nearly stay completely in the valid range when the third term is used.

3.2.4 Filtering

In [13], Gal et al. propose the enlargement of every patch in the leveled photos by one pixel for better filtering results when the underlying model is visualized. In our tests, we observed visible artifacts at the seams when just a one pixel wide border was calculated around each patch. This is illustrated in Figure 3.18. The used model is the ground of a corridor in the Domitilla catacombs. In Figure 3.18a, the unleveled version of the model is shown. The result when the patches are not enlarged, but only the used regions of the photos are leveled, is shown in Figure 3.18b. Obviously, unleveled regions are filtered into the currently used mipmap level used for visualization of the model. When a one pixel wide border is calculated around each patch, the filtering result is not significantly better than without the calculated borders. This is shown in Figure 3.18c. Therefore, we introduce a new method for the calculation of borders for the patches to ensure better filtering results.

Outline normals Our approach for the calculation of sufficiently wide borders around the patches in the photos is based on the computation of outline normals. For this, the set of projected edges of the mesh is found for every patch that forms the outline of the particular patch. Then, an iteration over the set of outline edges is carried out for every patch. For each outline vertex, the 2D normal vector inside the photo is calculated. This normal vector is then scaled to a user-defined length. All the endpoints of the normal vectors of a particular patch connected together define an area around this patch. This area is then leveled additionally to the area of the patch. The values of the leveling function at the endpoints of the computed normal vectors are simply evaluated by extrapolation. In Figure 3.19, the calculation of outline normals vectors is shown for three patches in a photo.

Discussion Using our introduced outline normal vectors for the user-defined en-


(a)

(b) (c) (d)

Figure 3.18: (a) Original unleveled model. (b) No enlargement of patches. (c) 1 pixel wide boundary, as proposed in [13]. (d) 20 pixel wide boundary using our outline normals, yielding much better filtering results

largement of patches delivers significantly better filtering results, as it is shown in Figure 3.18d. A remaining problem are intersections of regions of the calculated border of just one particular patch as well as the intersection of calculated borders of two or more different patches. This can lead to interferences of the leveling function. In future releases of our leveling application, we will introduce some kind of collision detection to prevent the interference of two differently leveled regions. Further, we want to improve our scaling approach to be similar to the approach shown in [33]. Similiar to [33], we think of not just shifting the vertexes of the border along the outline normals, but to produce three new vertexes for every outline vertex. These vertexes would then ensure that the shifted edges are parallel to the outline edges.

3.2.5 Issues

The leveling method that was presented by Lempitsky and Ivanov in [18] can deliver satisfactory results, but only if the input data is of high quality. A big problem are camera registration errors. These camera registration errors lead to misalignments of adjoining areas that are textured by different photos. Such misalignments are shown in Figure 3.20a. The used model is the Domitilla cubiculum. For copyright reasons, we can not show the entire model, but only this small section where we observed problems

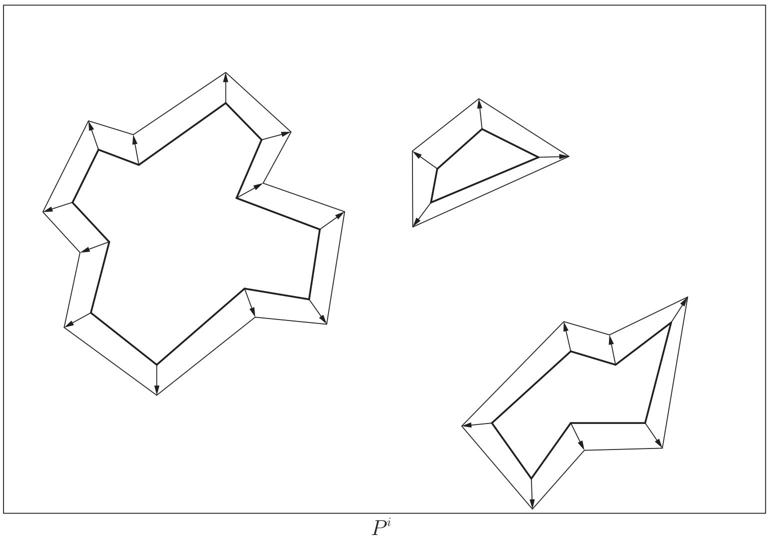
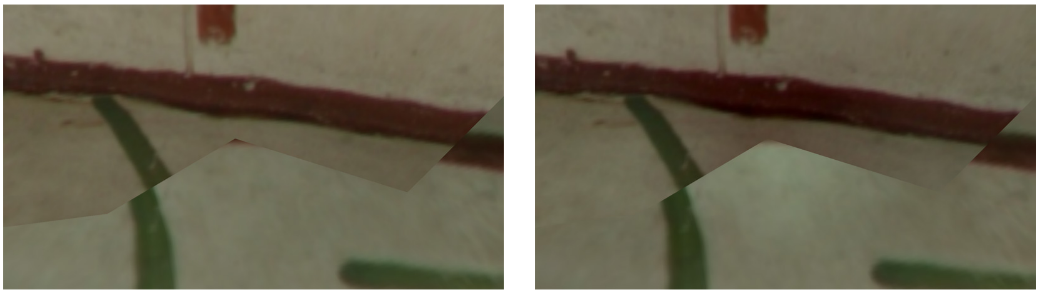


Figure 3.19: Calculation of outline normals and the resulting additionally leveled border regions for three patches in a photo P^i

with camera registration errors. In the figure, the upper part of the model is textured by a different photo than the lower part. Obviously, the image regions that belong together do not adjoin, but are shifted apart. Certainly, this is already unacceptable in the labeled model. Unfortunately, the issues become even worse when leveling comes into consideration. In the leveling approach proposed in [18], adjoining mesh regions that are textured by different photos are leveled together at the vertexes of the mesh. A big problem arises when a vertex receives significantly different colors when it is projected into the photos belonging to the adjoining mesh regions. As it is shown in Figure 3.20a, there is a vertex in the middle of the image that receives a red color for the lower photo, but a significantly different color for the upper photo. When the photos are leveled together at this vertex, the results are massive color shifts. This is shown in Figure 3.20b. Obviously, the color the vertex received for the upper photo was a green tone. The upper photo is shifted towards a red color at the vertex, while the lower photo is shifted into a green color at the vertex. This extreme color shift is then linearly interpolated along the triangles the vertex belongs to.

Another issue that is visible in Figure 3.20 is the problem that leveling function values are only calculated at the vertexes of the mesh, and are then linearly interpolated along the triangles. Assuming that we have a mesh consisting of very big triangles relative to the image information, there may be color differences in the photos used for texturing of adjoining mesh regions, that are not leveled together. This can also be the case when the camera-to-geometry registration is absolutely perfect.



(a) (b)

Figure 3.20: Massive color shifts after leveling a model with huge misalignments of photos caused by camera registration errors. (a) Labeled model. (b) Labeled + leveled model.

3.2.6 Implementation

Our *PoissonLeveler* is implemented in C++. As it is also the case for our MosaicBuilder, we use OpenCV in the version 2.3 for all image operations. An overview of the leveling procedure carried out by our PoissonLeveler is shown in Figure 3.21.

First, the labeled 3D model and all the corresponding photos that are used for texturing of the model are loaded. Then, all the connected components C_i are evaluated. The connected components are sets of adjoining triangles so that each set is textured by the same photo. Each of these connected components corresponds to a patch when it is projected into the photo from which the connected component receives its color information.

In order to enable the user-defined scaling of the patches to ensure better filtering results, the outline normals are calculated for each patch in every photo. After that, the linear equation system is constructed. To enable the leveling of huge models, we use a sparse solver. In our implementation, we use the SparseLib++ library in conjunction with IML++ [11]. SparseLib++ is a C++ class library that enables the storage of very sparse matrices in different formats. IML++ is a C++ templated library that delivers iterative methods for solving of linear equation systems.

The mentioned libraries are then used to solve the linear equation system. As it was shown in Section 3.2.2, for every vertex of the mesh there have to be as many variables g_i^j in the linear equation system as there are connected components the vertex belongs to.

When the leveling function has been calculated, it is added to the original texture. Therefore, each connected component is projected into its corresponding photo. Each vertex is projected onto a particular pixel of the photo. The color of this pixel is then altered according to the leveling function value that was calculated for the projected

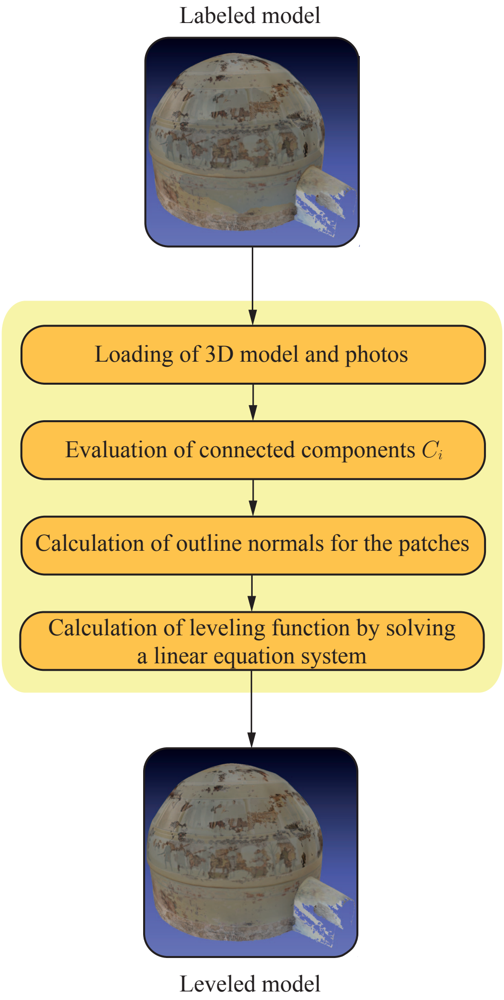

Labeled model
Loading of 3D model and photos
Evaluation of connected components C_i
Calculation of outline normals for the patches
Calculation of leveling function by solving a linear equation system
Leveled model

Figure 3.21: Overview of the leveleing procedure as it is performed by our Poisson-Leveler.

vertex and this connected component. In case of RGB photos, three leveling function values have been calculated that are used to alter the the three color channels of the pixel. The pixels along each projected edge are altered by the leveling function values that are calculated by linear interpolation of the leveling function values that were calculated for the two vertexes that define the projected edge. All the pixels inside the projected triangles are then also altered by the linearly interpolated leveling function. Our PoissonLevel does not overwrite the original photos, but generates the set of leveled photos at a user-defined destination.

CHAPTER 4

Manual workflow

In this chapter, we will show the manual workflow of the graphic artist that remains after the introduction of the applications introduced in Chapter 3. Although these applications can significantly reduce the work of the graphic artist, in practice they do not always deliver a perfect result. One reason for this are significantly different lighting situations during the exposure of the photos. When photos of an outdoor monument are taken at different times of day, the lighting situation naturally changes because of the different positions of the sun. This leads to different intensities in the photos. Also a flash light can not solve the problem, since it can not provide consistent lighting conditions. Furthermore, a flash light often leads to highlights in the photos. As already mentioned, visual artifacts like highlights resulting from using a flash light are not handled at all by our applications. These and other remaining visual artifacts have still to be removed manually in an image editing application. For this, we implemented applications in order to simplify the manual workflow of the graphic artist.

In Section 4.1, we will introduce the *MaskDrawer*, which is our application used for the generation of alpha masks. These masks are used for indication of areas in the photos that are used for texturing of the model. We will first talk about the method how the masks are generated. Then, we will give some insights into the implementation.

In Section 4.2, we will introduce the *VT-Tools*, which is our application used for the fast generation of the needed data structures for virtual texturing. Since there already exist some scripts for this task, we will compare our application and the scripts throughout the section.

4.1 MaskDrawer

4.1.1 Overview

Our first application that we implemented for the improvement of the manual workflow of the graphic artist is a simple application for mask generation. A mask in this context is an image with the same resolution as the corresponding photo used for texturing of the model. Every mask consists only of black and white pixels, it is therefore sufficient to store them as binary images. A black pixel corresponds to a pixel in the dedicated photo that is not used for texturing of the model. In contrast, a white pixel corresponds to a used pixel in the corresponding photo. The masks can be used in any image editing application as the alpha channel for the corresponding photos.

The essential information of a particular 3D model to generate the masks are the face indices into the texture coordinates list, the texture coordinates themselves, as well as the material information to know for every face the photo that is used to texture it. The indices into the texture coordinates list are important to know which of the texture coordinates belong together to form a primitive, usually a triangle. Texture coordinates are normally in the range $[0, 1)$ for both u and v [2] to address the whole area of a single texture. This enables swapping of textures with different resolutions without the need to change the texture coordinate values [2]. When generating the masks, it is therefore important to use a virtual camera that images exactly the region of texture coordinates that correspond to the whole texture area. The virtual camera that fulfills these requirements is an orthographic camera positioned at $\mathbf{p} = (\ 0.5,\ 0.5,\ 0.0\ )$. The width and height of the view frustum both have to be 1.0. Since the most intuitive way to render 2D content is to use the XY-plane, the Z-value for the near clipping plane of the view frustum has to be positive, whereas the Z-value for the far clipping plane must be negative. A view frustum with exactly this configuration can be seen in Figure 4.1.

For performance reasons, we decided to use the GPU for the generation of the masks. Therefore, for each photo for which a mask is generated, a buffer with exactly the same size as the photo is created. After that, the buffer is initialized with the color that corresponds to invisible pixels in the corresponding photo, namely black. Then, all the triangles of the model that are textured by the current photo are rendered into the buffer with white color using the texture coordinates as if they were vertex positions. Because texture coordinates consist only of two elements (u and v), the Z-value is set to 0.0 to render the primitives into the XY-plane.

4.1.2 Implementation

Our MaskDrawer is implemented in C++ using OpenGL 3.1 as the graphics API. Currently, it is only available for Microsoft Windows. In oder to make it small and simple, we omitted a graphical user interface. Per default, a shortcut to the application is copied

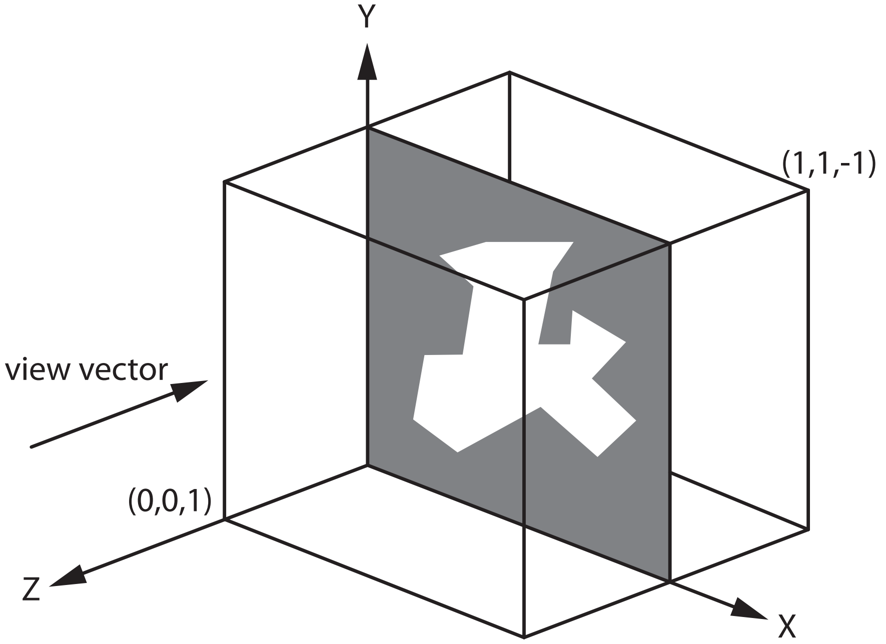


Figure 4.1: View frustum of the orthographic camera used for rendering of the masks. The masks are rendered into the XY-plane.

to the “SendTo” directory. Therefore, it can be launched via right click onto the model file followed by a “Send To” to our MaskDrawer. Then, the masks are generated in a subdirectory called masks. To avoid any error message, the photos used for texturing of the model have to be at the destination referenced in the model file.

4.2 VT-Tools

An essential part of the manual workflow of the graphic artist is the visualization of the 3D model. Without visualization, it is not possible to see the remaining artifacts in action. It is important that the graphic artist can see the artifacts in the rendered model to exactly know the needed editing steps in the corresponding photos. In case of the digitized archaeological items, the high-resolution photos used for texturing the model do not fit into the memory of a conventional graphics card. This leads to slow frame rates and jerky movement through 3D space, which makes a fast positioning of the virtual camera difficult. Therefore, we decided to introduce virtual texturing into the workflow of the graphic artist. With virtual texturing, only the needed texture parts used for rendering of the current frame are streamed to the graphics card. There is already a library for virtual texturing that was implemented by J. Mayer, the author of [23]. This library is called LibVT. The LibVT has been integrated into a visualization application developed at the Vienna University of Technology. This application is called Scanopy, and it is primarily used for visualization of point-based datasets. The idea was that the graphic artist shall use Scanopy in conjunction with the LibVT for better visualization performance of the 3D models. In addition to the LibVT, J. Mayer also implemented some scripts for the generation of the needed atlas and tile store. However, these scripts, implemented in Python, are so slow that we decided to re-implement this functionality to reach a better performance. The whole functionality was implemented in a library called *VT-Tools*.

4.2.1 Atlas generation

The script implemented by Mayer for the atlas generation expects the single names of all the images that will be part of the atlas. Further, it waits for values for the number of parts the atlas is stored into and the side length in pixels of the whole atlas. Since the user has to do some unnecessary calculations to get the values fulfilling the requirements (e.g. finding the atlas side length that is suitable for the chosen images, finding the number of parts that corresponds to a handy side length of produced parts), we decided to simplify the required input parameters for the *VTTAtlas*, the object inside the VT-Tools that corresponds to an atlas used for virtual texturing. For the atlas generation, our library only expects the path of the directory containing all the images that will be part of the atlas. Further, only the maximum side length of a single atlas part is expected. The side length of the atlas that is suitable for all images in the provided directory will be calculated automatically. In contrast to the python script that only produces bitmap files, our library allows for different file formats (PNG and JPEG).

4.2.2 Tile store generation

The existing python script for the tile store generation can only be called to generate the tiles for a single part of the atlas that was produced with the python script used for atlas generation. Therefore, the script has to be called for every single atlas part. Then, the tile store is still not ready for use. All the produced sub tile stores then have to be merged together. This leads to new problems. Assuming that there are four atlas parts, there is one missing tile store level that has to be generated after merging the four corresponding sub tile stores. With 16 atlas parts, there are two missing tile store levels and so on. When merging the sub tile stores, the existing levels have to be copied together and the single tile files renamed to include the coordinate in the merged tile store. As if that was not enough, the borders of some tiles have to be fixed. This can be the case when bilinear filtering is used and the tiles are generated with a border of one or more pixels.

In contrast to this unnecessary indirection, the VT-Tools generate the whole tile store in a single run including all the generated atlas parts.

4.2.3 Update of atlas and tile store

For a smooth manual workflow, an update of the atlas and the tile store has to be enabled. Although Mayer states in his thesis that "runtime modification of a virtual texture is an expensive operation that should be avoided" [23], in our case it is unavoidable. In the manual workflow, the graphic artist is using atlas and tile store based on a set of photos for high-performance rendering of the corresponding model. Virtual texturing enables higher frame rates to position the camera to see the mentioned artifacts in action. After estimation of the needed editing steps, the concerned photos are edited in an image editing application like Adobe Photoshop. When the editing step has been done, the graphic artist needs to see the changes in the rendered model immediately. With virtual texturing, atlas and tile store have to be updated to make this possible.

To make an update as fast as possible, we avoid a full rebuild of atlas and tile store but only touch the changed parts. Therefore, it is essential that the exact position of every image inside the atlas is known. Further, the time of last change of every image has to be stored to recognize an image change. This is accomplished by the generation of a small text file during the generation of the atlas. Into this text file, the exact position and time stamp of last change of every image that is part of the atlas is stored. The time stamp of the image files can simply be queried from the operating system. Assuming that an image was changed and the update procedure is run, the current time stamp of the image file and the corresponding time stamp in the text file would differ, indicating an image change. Because the position of the image inside the atlas can be read out of the text file, it is easy to find the atlas parts that are concerned by the update. These atlas parts have then to be read in, the parts corresponding to the image have to be replaced by the new version of the image, and then the atlas parts have to be saved again. After the update of the atlas, every level of the tile store has to be updated. For level 0 of the tile store, the atlas parts themselves deliver the needed image data for the update. For higher levels, the atlas parts have to be scaled down to the half again and again. Because the atlas can be stored in 4, 16 or more files, there may be atlas parts that are not concerned by the update procedure. Therefore, scaling down the unconcerned atlas parts for every update would be redundant work. Therefore, in addition to the atlas parts, also all needed mipmap levels of the atlas are stored on the hard disc. Certainly, also these scaled atlas parts have to be updated when they contain parts of the changed image.

Tile cache and VRAM. When atlas and tile store are updated, it is also important that the dedicated memory region on the graphics card (*VRAM*) that is used to store the currently used tiles is updated. The LibVT also holds a finite number of tiles in main memory to prevent another time consuming streaming from hard disc. Another streaming of a particular tile can be necessary when it has been overwritten inside the VRAM by another tile because of a certain time period without usage. The dedicated region in main memory reserved for a finite number of tiles is also referred to as the *tile* *cache*. A problem that arises when the tile cache and the VRAM are not updated is the simultaneous usage of tiles that correspond to the old and the new version of the changed image. This is because of the (wise) lazy practice of the LibVT. If a tile that is needed to render the next frame is already available in the VRAM, it is used immediately, no matter if there is an updated version of it on hard disc. This also holds for the tile cache. If the graphics card requests a particular tile that is available in the tile cache, the version stored in the tile cache is streamed to the graphics card. In Figure 4.2, the artifacts that arise when tile cache and VRAM are not updated after an update of atlas and tile store are shown.

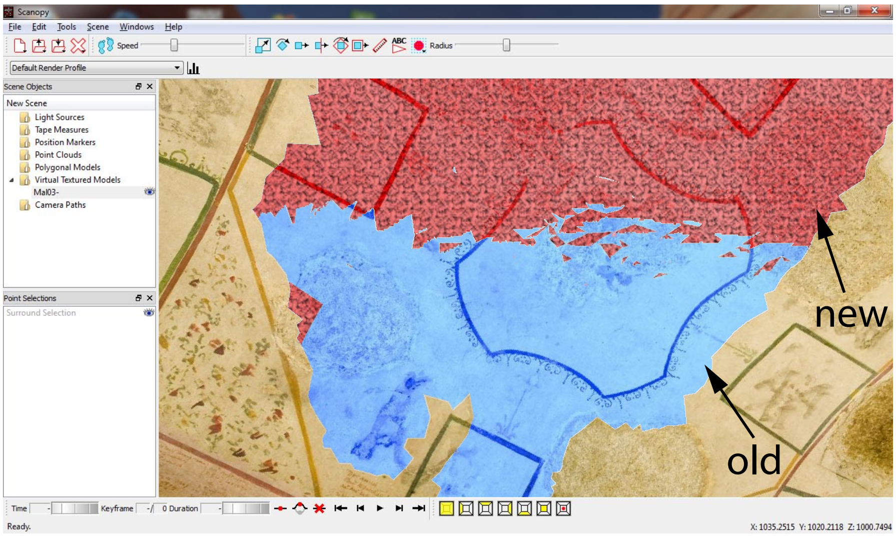


Figure 4.2: Visible artifacts in the model when atlas and tile store have been updated, but not tile cache and VRAM. The artifacts arise because of the simultaneous usage of old and new (patterned) version of tiles corresponding to a particular photo.

Unfortunately, the LibVT was originally not designed to support a changing atlas and tile store. After the initialization step, a modification of atlas and tile store is not allowed, otherwise the shown artifacts arise. Fortunately, J. Mayer was so friendly to implement two further functions into the LibVT. One function is used to delete a particular tile from tile cache, one function is used to delete it from VRAM. With the new version of the LibVT, it is now possible to delete an old tile from tile cache and VRAM so that it is loaded again from hard disc.

We integrated the new version of the LibVT into Scanopy. The update procedure now works in a semiautomatic way. When a model is visualized using virtual texturing, it is possible to change one or more photos which are part of the atlas. Then, the update procedure is called inside Scanopy via keystroke. During the update, the atlas and tile store are updated on hard disc as it was described before. After that, the new functions of the LibVT are used to delete the old versions of the tiles from tile cache and VRAM. This ensures that the changed tiles are loaded again from hard disc and streamed to the GPU. After the update procedure, the changes of the photos are visible in the rendered model immediately.

4.2.4 Implementation

The VT-Tools are entirely implemented in C++. We use the libjpeg-turbo for loading and writing of JPEG files and the libpng for loading and writing of PNG files. The libjpeg-turbo has been chosen for the JPEG files because it produced the best results regarding loading time in the performance test shown in [23].

The VT-Tools have been implemented in a way so that it can be called similar to the MaskDrawer via “SendTo”. The only file that it expects is a small text file with all the needed parameters. The most important of these parameters are the maximum side length of the atlas parts, the paths where the atlas and the tile store will be stored, the side length of the tiles, as well as the output formats of the atlas and the tile store. The configuration file has to be placed into the directory where all the images are stored that will be part of the atlas. When the VT-Tools are called by “SendTo” using such a parameter file, all the images files that are in the base directory of the parameter file are taken into account. If all the parameters are valid, the VT-Tools generate the atlas, split into as many parts so that the maximum side length is not exceeded, and the tile store, consisting of tiles of the desired length. The number of levels of the tile store results from the size of the atlas and the side length of the tiles.

CHAPTER 5

Results

5.1 Platform

We use two different systems for testing of our applications. The first system is a personal computer with an Intel i7 2600K CPU with 3.4 GHz, 8 GB RAM (Corsair XMS3 PC3-10667U CL9-9-9-24), 128 GB solid state drive (Crucial RealSSD C300), 2000 GB hard disc drive (Western Digital Caviar Green, 64MB Cache) and a nVidia GeForce GTX 570 graphics card. If it is not mentioned explicitly, this system is used as platform for testing and the hard disc drive is used as the origin of the data for the tests. The operating system used is Microsoft Windows 7 Professional in the 64 Bit edition.

The second system is a Hewlett Packard Pavilion dv6599eg notebook with an Intel Core2Duo T7300 processor with 2.0 GHz, 2 GB RAM and an nVidia GeForce 8400M GS graphics card. The operating system used is Microsoft Windows 7 Professional in the 32 Bit edition.

5.2 Performance Tests

In this section, we will evaluate the performance of our applications. All performance values are snapshots, which means that we do not average over several iterations. In order to present reasonable numbers, the times needed to load the models and the photos from hard disc were subtracted from the overall runtimes.

5.2.1 MosaicBuilder

The runtime of our *MosaicBuilder*, the application for the automatic labeling of a triangulated 3D model, depends on several parameters. Some of them, like the smoothness cost weight λ, can be chosen by the user. Other parameters, like the number of triangles, are naturally given by the 3D model that is the input for the labeling process. We want to show the effect of these parameters on the runtime in the following.

Unfortunately, the labeling with shift vectors as it was presented by Gal et al. in [13] takes much time. This approach was also implemented into our *MosaicBuilder*, and we tested the performance for the Centcelles cupola model which has about four million triangles and 70 input photos. The labeling procedure took about 5 days with deactivated occlusion detection for higher performance, but the result was still not satisfactory. The reason for this was an inappropriately chosen value of 6 for the maximum number of levels for the image pyramids. Also a better value of 3 would have led to a runtime of about 2 days. Because the approach of [13] is therefore only suited for very small models with a small number of input photos, we only tested the standard labeling approach as it was presented by Lempitsky and Ivanov in [18] with our introduced octree for occlusion detection thoroughly.

5.2.1.1 Number of triangles

The number of triangles is naturally given by the 3D model that is used as input for the labeling procedure. In order to have some models with a different number of triangles so that the number changes in a linear way when they are ordered by their number of triangles, we generated differently detailed versions of one particular model in Geomagic. Our input is the Centcelles cupola model. We generated five versions with a number of triangles from $300{,}000$ to $100{,}000$.

In Figure 5.1, the impact of the number of triangles of the model on the labeling time is illustrated. To be comparable, all parameters apart from the number of triangles were the same for each iteration. We used all available photos (70) as input labels. The number of iterations for the α-expansion Graph Cuts algorithm in the form of the gco-v3.0 library was set to 2, the smoothness cost weight λ was set to 25. The upper threshold for the octree cell side length was set to 0.1.

The red line in the graph shows the times with occlusion detection, the blue line without occlusion detection. Obviously, the occlusion detection has a significant impact on the labeling time. In every case, the labeling time with occlusion detection at least doubles the labeling time without occlusion detection. In our case, the occlusion detection was not really needed because the model, the Centcelles cupola, has naturally no occlusions because of its convex shape. Nonetheless, for very complex models that are far away from convexity, the longer labeling time using occlusion detection has to be accepted in order to get a good labeling result.

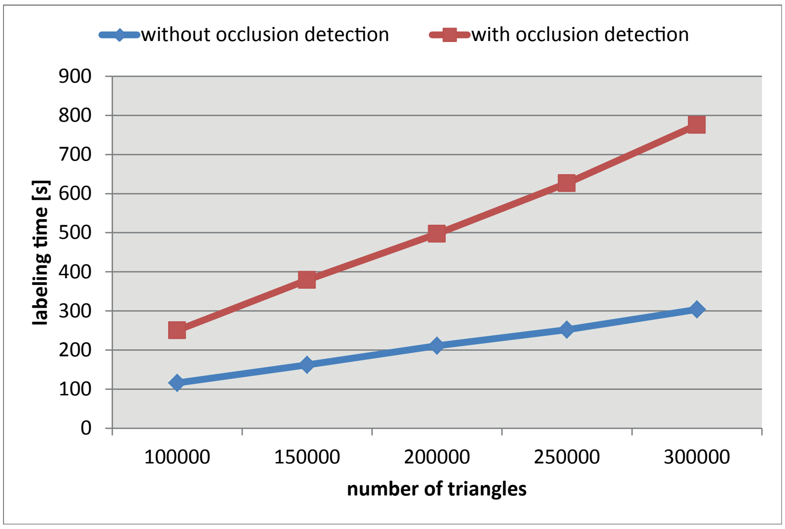


Figure 5.1: Impact of the number of triangles on the time needed to label the model

5.2.1.2 Number of labels

The number of labels is given by the number of registered photos that are used for texturing of the model. In our performance tests concerning the impact of the number of labels, we used the version of the Centcelles cupola model with $150{,}000$ triangles. The occlusion detection was deactivated, the number of iterations for the α-expansion Graph Cuts algorithm in the form of the gco-v3.0 library was set to 2, and the smoothness cost weight λ was set to 25.

In Figure 5.2, the impact of the number of labels on the labeling time is illustrated for the runtimes with and without occlusion detection. For the labeling procedure without occlusion detection, the time needed to label the model, linearly depends on the number of input photos. When occlusion detection is activated, the time needed to label the model increases more intensive the more labels are considered.

5.2.1.3 Upper threshold for octree cell side length

As already mentioned in Section 3.1.4, we allow the user to define an upper threshold for the side length of the octree cells. Only when all three side lengths of a particular cell are bigger than this threshold, the cell is further divided into its octants. Otherwise, the cell is a leaf node of the octree and the triangle is added immediately, provided that the triangle intersects the cell.

In Figure 5.3, the impact of the upper threshold for the side lengths of the octree cells on the labeling time is illustrated. In the figure, the labeling times for a threshold of 0.075 and 0.1 are equal, because these values lead to the very same number of octree

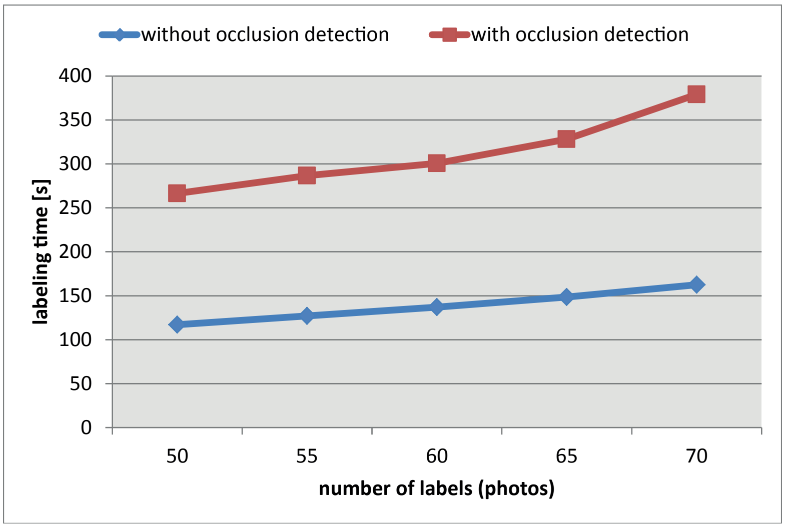


Figure 5.2: Impact of the number of labels (photos) on the time needed to label the model

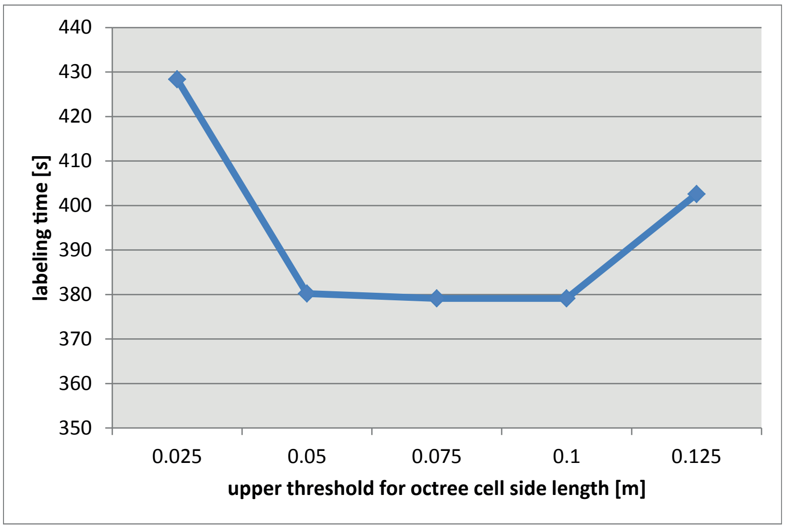


Figure 5.3: Impact of the upper threshold for the octree cell side length on the time needed to label the model

layers. These values correspond to the average diameter of a triangle of the mesh. As can be seen, the labeling times are minimum for these values. A bigger threshold leads to a higher labeling time because of the resulting fewer number of octree layers. When the octree has fewer layers, the cells corresponding to the leaf nodes of the octree are bigger and therefore each cell has to store more triangle indexes. Then, during a particular occlusion test, more real triangle-triangle intersection tests have to be carried out than with an octree with more layers. The cells corresponding to the leaf nodes of an octree with more layers are smaller so that many triangles that come into consideration for an occlusion can be filtered out by a more efficient intersection test with some axis aligned boxes.

As it can be further seen in the figure, also a too small upper threshold leads to a higher labeling time. This is because of the fact that when using very small cells corresponding to the leaf nodes of the octree, one particular triangle of the model is stored in many leaf nodes. When testing a triangle sight frustum and the octree for intersection, it is possible that one particular triangle that is stored in the octree comes many times into consideration for an intersection and is then also tested every time. Certainly, it is possible to use a data structure to remember all the triangles that heave already been tested for intersection. However, we observed an even worse performance caused by the overhead of such a data structure.

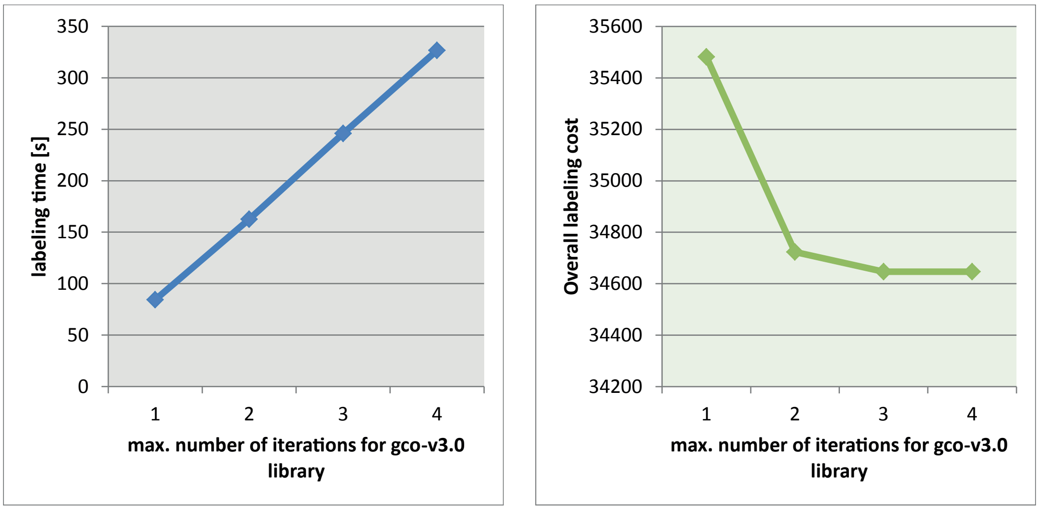
5.2.1.4 Max. number of iterations

The gco-v3.0 library that implements the α-expansion Graph Cuts algorithm, which we explained in Section 3.1.3, allows the user to set the number of maximum iterations. This maximum number also has a significant impact on the labeling time. This is shown in Figure 5.7a. As can be seen, the time needed to label the model linearly depends on the number of iterations. However, a bigger number of iterations not always implies a smaller overall labeling cost and therefore a better labeling result. This is illustrated in Figure 5.7b. From the first to the second iteration, there is a significant reduction of labeling cost. From the second to the third iteration, the improvement is not that big anymore. Finally, from the third to the fourth iteration, there is no noticeable improvement. In our labeling experiment, we mainly did two iterations, which seams to be a good trade-off between labeling time and cost.

5.2.2 PoissonLeveler

The runtime of our *PoissonLeveler*, the application for the automatic leveling of the overall texture of a 3D model, depends on the size of the model and the number of connected components. Each connected component is textured by the same photo. The parameter λ that controls the degree of penalizing of color differences between adjoining connected components C_i does not have any influence on the runtime.

In Figure 5.5, the impact of the number of triangles on the time needed to level the overall texture of a 3D model is shown. The used models are the differently detailed versions of the Centcelles cupola. Each of these versions is textured by about 30 photos. Therefore, the number of connected components is approximately the same for all instances. As it can be seen in the figure, for the first four versions the leveling time



(a) (b)

Figure 5.4: Impact of the maximum number of iterations for the gco-v3.0 library on the time needed to label the model (a) and the overall labeling cost (b).

linearly depends on the number of triangles. Only for the model with $300{,}000$ triangles, the resulting runtime is an outlier of the observed linear increase.

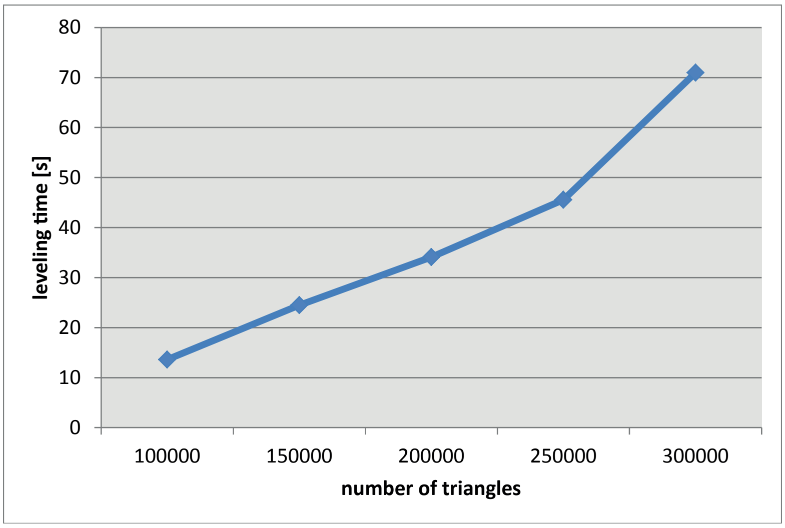


Figure 5.5: Impact of the number of triangles on the time needed to level the overall texture of a 3D model.

5.2.3 MaskDrawer

A performance test for our *MaskDrawer*, the application for the generation of black-and-white masks for all photos that are used for texturing of a 3D model, is not very spectacular. Since it only reads a model file and all the corresponding photos from hard disc, generates the masks on the GPU, and writes the masks back to hard disc, its runtime naturally depends on the number of photos. Certainly, a big model file needs more time to be loaded, but in practice this is insignificant. Concerning the input photos, we observed a performance of about 14.3 megapixel per second. This number includes the times for loading and writing of the masks.

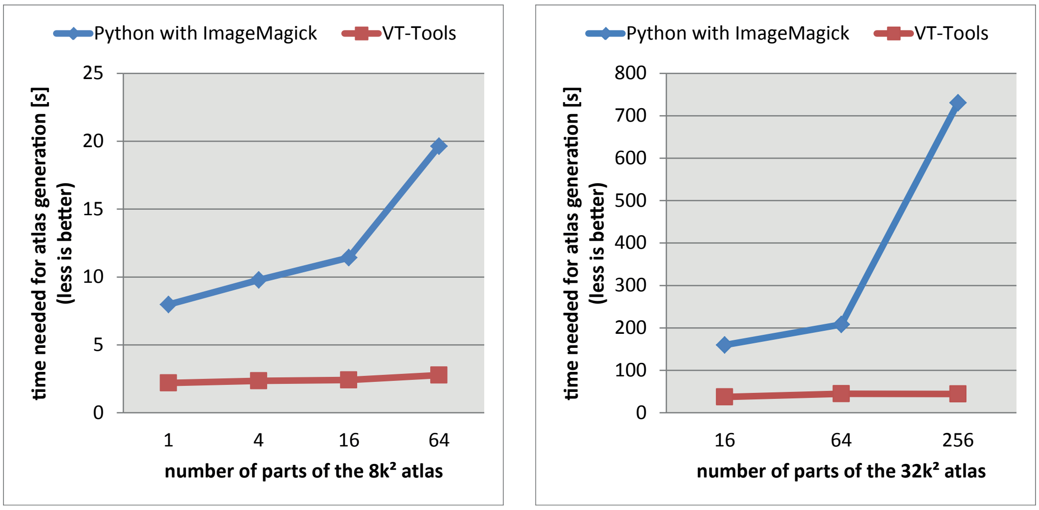
5.2.4 VT-Tools

The VT-Tools, our application for the generation of the data structures that are needed for virtual texturing, has been compared to the existing scripts implemented by Mayer, the author of [23]. The scripts are implemented in Python using the Python Imaging Library and ImageMagick for image operations. For consistency reasons, we altered the scripts for our tests so that only ImageMagick is used. This is no violation, since it was planned by the author of the scripts according to some code comments anyway. The VT-Tools have been tested on our second testing platform, which is the Hewlett Packard notebook with Intel Core2Duo T7300 processor.

We split this section into three parts. The first part covers the performance tests concerning the atlas generation. In the second part, we will take a look onto the needed times for the tile store generation. Finally, we will evaluate the performance of the implemented update procedure.

5.2.4.1 Atlas generation

In Figure 5.6, the performance of our VT-Tools concerning atlas generation compared to the existing Python implementation is shown. We tested the performance for two different atlas sizes ($8k^2$ and $32k^2$). In both cases, our application is significantly faster than the existing Python script. While our application needs approximately the same time to generate an atlas with a particular size, regardless of the number of files the atlas is stored into, the Python script needs the longer the more parts are produced. This behavior can be explained by the nature of the script. The script calls ImageMagick for every single atlas part that is generated. So when the atlas is stored in 64 files on hard disk, ImageMagick has to be called 64 times. One particular photo has therefore to be loaded from hard disk again for each part the photo belongs to. In contrast to this behavior, our application keeps as many photos as possible (up to a user-defined upper limit for the memory consumption) in memory.



(a) (b)

Figure 5.6: Performance of the VT-Tools concerning atlas generation compared to the existing Python implementation. (a) Generation of an 8k^2 atlas. (b) Generation of a 32k^2 atlas.

5.2.4.2 Tile store generation

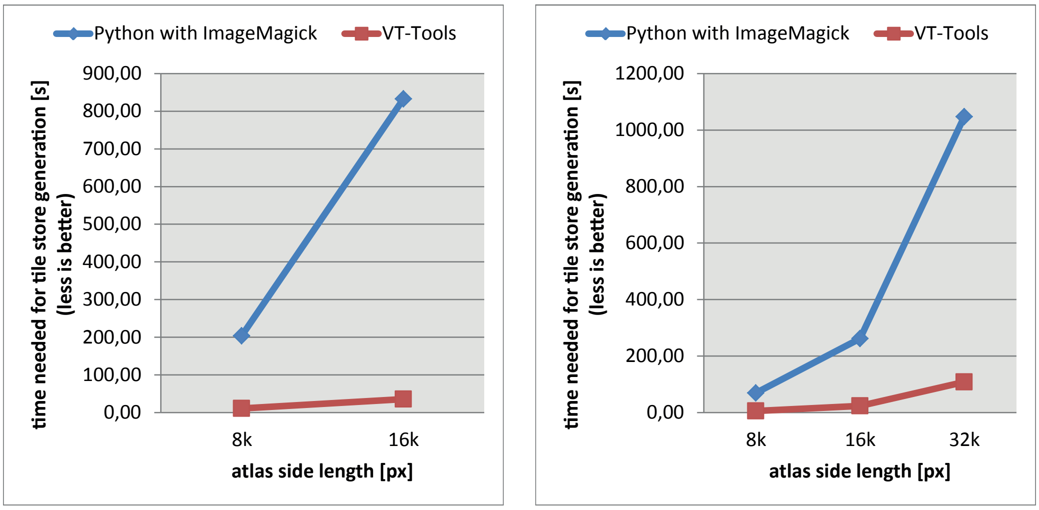
Also the generation of the tile store is significantly faster with our VT-Tools than with the existing Python implementation. This is shown in Figure 5.7. We tested the generation of tile stores with sizes for the tiles of 128^2 and 256^2. Because of the bad performance of the Python script, we omitted the generation of the tile store consisting of tiles with a size of 128^2 for the 32k^2 atlas.

The scripts need significantly more time to generate the tile store the more tiles are produced. The reason for this is the same as for the atlas generation. Because ImageMagick is called for every single tile that is produced, at least one atlas part has to be read from hard disc every time.

We also tested the performance of our VT-Tools when generating an atlas with a size of 128k^2 with its corresponding tile store with a tile size of 128^2. The atlas was generated in 9min 11s, the tile store (11 levels with 1,398,101 tiles) in 2h 58min 21s. In contrast to all the other tests of the VT-Tools, this big atlas with its tile store has been generated on the Intel i7.

5.2.4.3 Update procedure

The easiest way to update atlas and tile store is a rebuild. However, when just a single photo has changed, there are many parts that are unnecessarily regenerated. Therefore,



(a) (b)

Figure 5.7: Performance of the VT-Tools concerning tile store generation compared to the existing Python implementation. (a) Tile size of 128^2. (b) Tile size of 256^2.

we implemented an update function that only touches the changed parts. The performance boost of our optimized update function in comparison to a rebuild of atlas and tile store is shown in Figure 5.8. We tested our update function for two different atlas side lengths (8k and 32k). The tile stores consist of tiles with a side length of 128 pixels. We only changed a single photo before we started the update. As can be seen in the figure, the optimized update is significantly faster than a full rebuild.

5.3 Our applications in practice

5.3.1 Scanopy integration

Our applications were compiled as libraries and integrated into Scanopy, an application developed at the Vienna University of Technology and at the Imagination Computer Services GmbH. Scanopy is mainly used for the visualization of huge point clouds, but it can also be used for polygonal models. The already mentioned LibVT that enables virtual texturing functionality has already been integrated into Scanopy.

All the functionalities of our applications can be controlled by the user by means of a graphical user interface. Therefore, we implemented some dialogs into Scanopy.

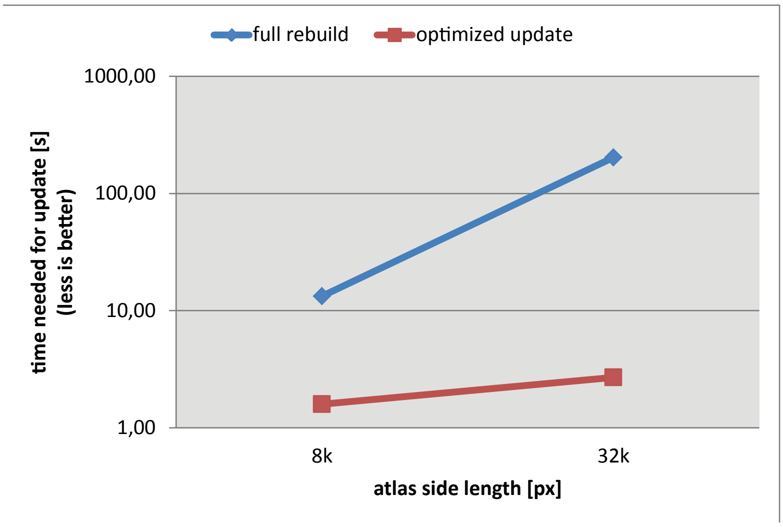


Figure 5.8: Performance of the optimized update function of the VT-Tools compared to a full rebuild of atlas and tile store.

5.3.2 Overall workflow example

In this section, we want to show how the functionalities of our applications will ease the work of the graphic artist in future by means of an example. To this, we will show how the model of the Centcelles cupola first walks through the automatic workflow for labeling and leveling. Then, we will show how the masks are used to ease the manual editing. Finally, we will show how the model is visualized in Scanopy using virtual texturing.

5.3.2.1 Automatic Workflow

In the automatic workflow, the 3D model of interest is labeled and leveled. The parameters for these procedures can be set in the mentioned dialogs we implemented into Scanopy. In future, the graphic artist can simply choose a triangulated 3D model and a set of registered photos via file dialog. The registration information for the photos either comes from a RiScan Pro project file or from the files produced by the 3DM Analyst from Adam Technology. When the model file and the photos are chosen, the graphic artist can set values for the parameters that are needed for the labeling process. The most important parameter is the weight for the smoothness cost term that penalizes color differences between adjoining triangles. A simple click on a button then starts the labeling process. The labeled model file and all the undistorted photos are then stored at a predefined location. The undistortion of the photos is performed as it was described in Section 3.1.2. For our example, we chose 25 as weight for the smoothness cost term.

The input mesh is shown in Figure 5.9a. The result of the labeling process is shown in Figure 5.9b.

The input for the leveling procedure is the labeled model in conjunction with the undistorted photos. The model file can again be chosen via file dialog. The most important parameter for the leveling process is the weight for the term that penalizes color differences between adjoining connected components. For our example, we chose 100 as weight for this term. The result of the leveling process is shown in Figure 5.9c.

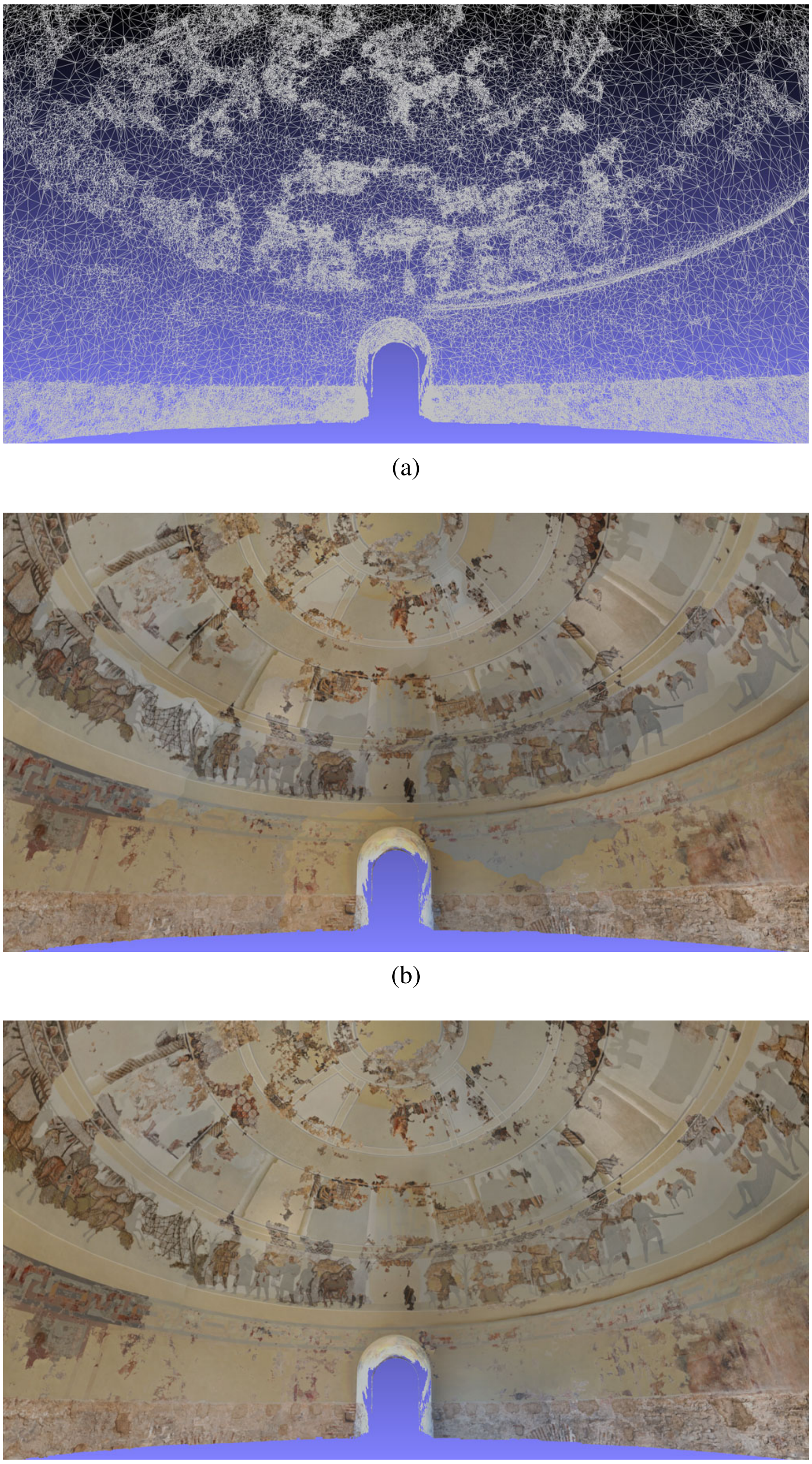
5.3.2.2 Manual Workflow

In the manual workflow, the remaining visual artifacts are handled in an image editing application like Adobe Photoshop. For this, the masks produced by our MaskDrawer are used. The masks show the regions of the corresponding photos that are used for texturing of the model and support the graphic artist during the manual editing steps so that he knows where the main focus of the editing has to be.

In order to generate the masks, we also integrated the MaskDrawer into Scanopy. Because the masks are defined as soon as the model is labeled, they are generated directly after the labeling procedure.

In Figure 5.10, one photo that is used for texturing of the Centcelles cupola model and the corresponding mask is shown. This mask can be loaded into an image editing application and assist the graphic artist by showing the regions that are used for texturing.

In Figure 5.11, a screenshot of the Scanopy application while rendering the final Centcelles cupola model using virtual texturing is shown. As already mentioned, we implemented an update function into the VT-Tools. When a model is now visualized in Scanopy using virtual texturing, it is possible to edit one or more photos that are part of the atlas. Then, a single keypress starts the update procedure. The atlas and the tile store are updated so that they contain the new versions of the changed photos. Then, the old versions of tiles that correspond to changed image regions are deleted from the tile cache and from the VRAM. The new versions of the tiles are then loaded so that the manual changes of the photos are visibile inside the 3D model immediately.

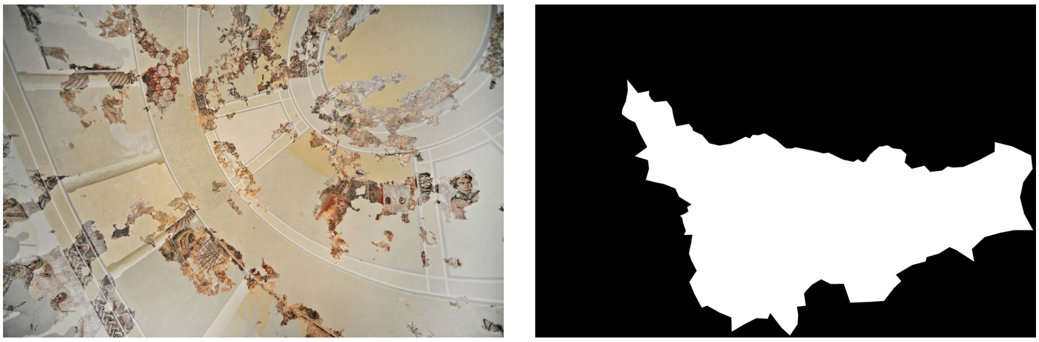


(a)

(b)

(c)

Figure 5.9: Result of the automatic workflow. (a) Triangulated 3D input model. (b) Labeled model. (c) Labeled and leveled model.



(a) (b)

Figure 5.10: (a) Photo used for texturing of the final model. (b) Corresponding black-and-white mask that is used as an alpha mask

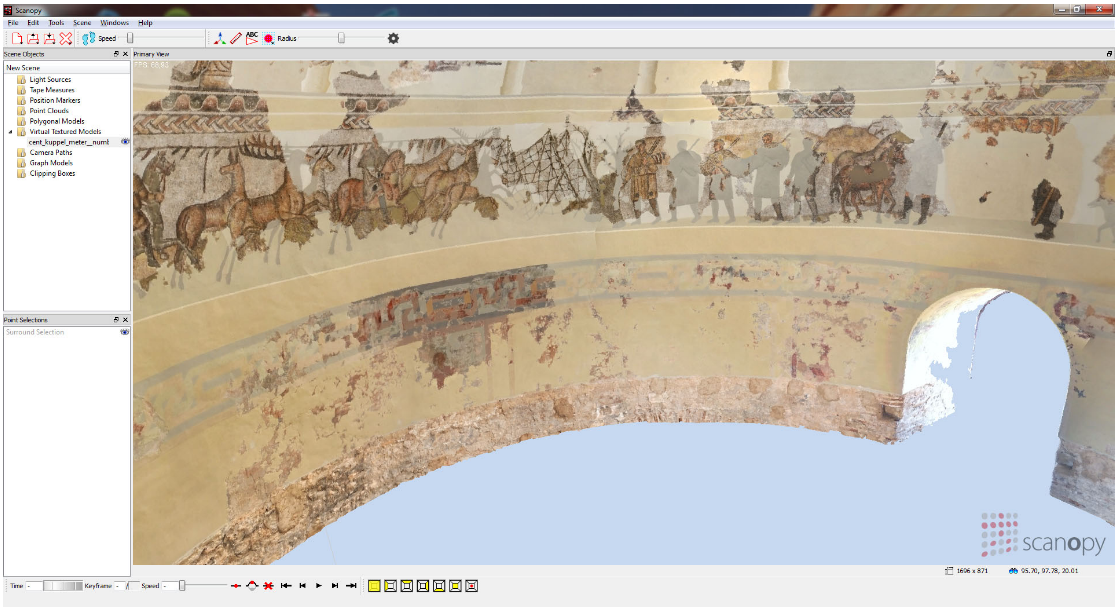
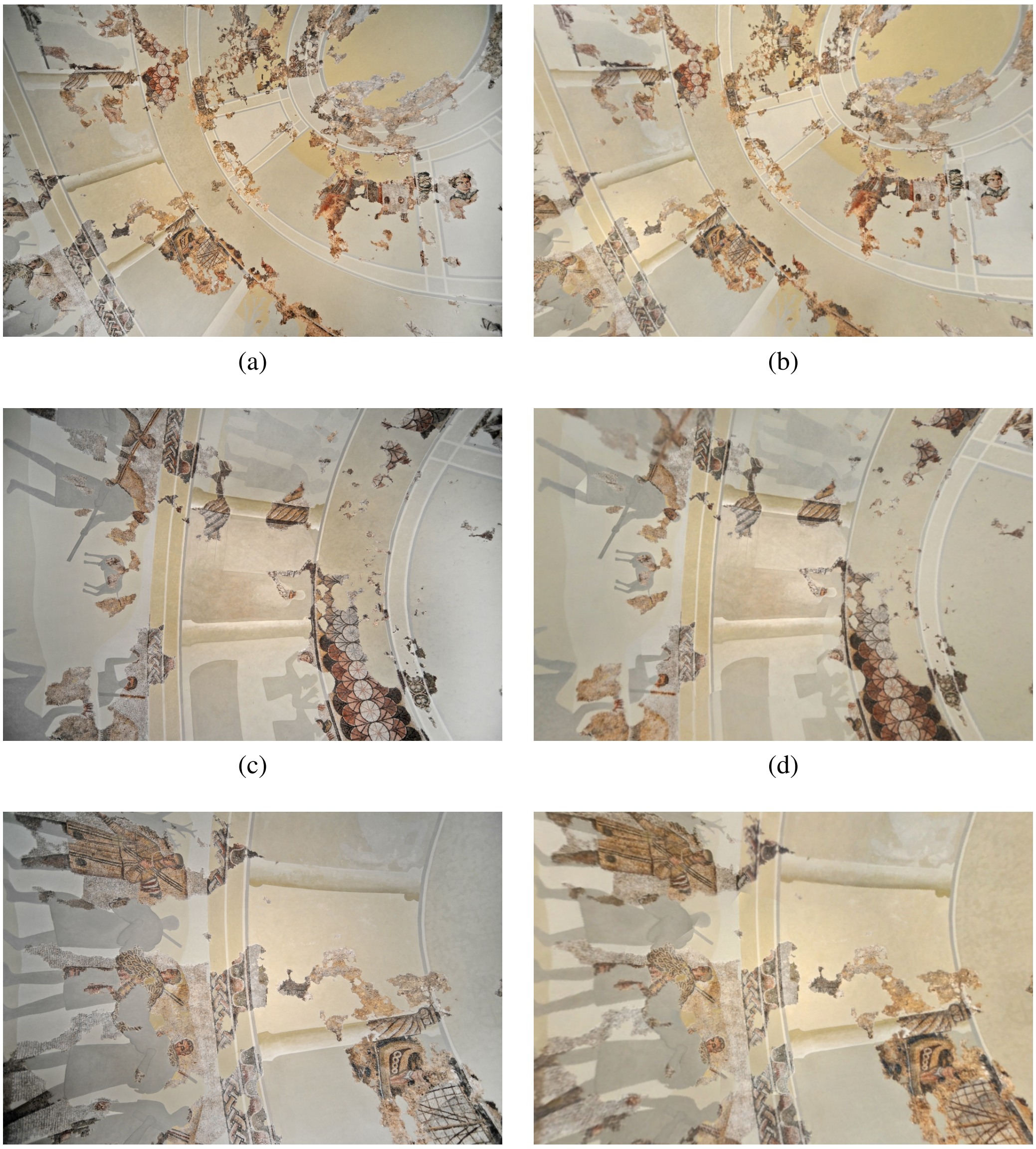


Figure 5.11: Screenshot of the Scanopy application while rendering the final Centcelles cupola model using virtual texturing.

5.4 Ground Truth Test

A remaining question is how the quality of a final labeled and leveled model is in comparison to the original input photos that are used for texturing. This comparison can also be considered as a ground truth test, since the photos show the archaeological monument of interest how it looked like during the time of exposure.

For the ground truth test, we take some original photos of an archaeological monument and render some images of the labeled and leveled 3D model of the monument. We set the internal and external parameters of the virtual camera to be the same as of the real world camera. Ideally, the input photos and the rendered images using the virtual camera look the same. In Figure 5.12, we show the ground truth test for the labeled and leveled model of the Centcelles cupola. In the left column, some original input photos are shown. In the right column, there are the corresponding views of the final labeled and leveled 3D model. The photos and the corresponding views look the same at first sight, which shows that our labeling application correctly maps the photos onto the model. However, the original photos provide a higher contrast than their corresponding views of the 3D model. This can be explained by the fact that a particular region of one original photo is not necessarily the one that is used for texturing of the corresponding surface area of the model, since it can also be part of another photo that shows this surface area from a worse viewpoint. Further, the quality of the views of the 3D model is negatively influenced by visible artifacts that are caused by camera registration errors. These registration errors lead to misalignments of the photos as it is visible on the right side of Figure 5.12b. When a labeled model that contains misalignments of photos is leveled, color shifts as they are visible at the top left of Figure 5.12d can arise.



(a) (b)

(c) (d)

(e) (f)

Figure 5.12: Comparison of some original photos (a, c, e) of the Centcelles cupola and their corresponding views of the labeled and leveled 3D model (b, d, f).

CHAPTER 6

Conclusion and Future Work

We presented a set of applications for the post-processing of a digitized archaeological item. The input of our approach is a triangle mesh of the item and a set of registered photos. We showed how we could further improve the results of existing labeling methods. For this, we introduced an octree for occlusion detection to prevent texturing of surface areas with image material that does not contain the corresponding image information, but the colors of an occluder. Further, we presented different methods for photo undistortion, which is an essential part in the labeling process.

For the automatic adjustment in terms of colors of the photos used for texturing of the model, we showed how we could improve existing methods used for leveling. We introduced a new term into a proposed least squares problem in order to prevent the leveled colors to leave the valid range. Because the proposed approach of adding just a 1-pixel wide border to the patches in the photos is insufficient in our opinion, we presented the user-defined scaling of the patches by introduction of outline normals. The scaling of the patches then ensures better filtering results, so that unleveled regions are not filtered into areas of the photos that are used for texturing of the model.

Because our two proposed applications for automatic labeling and leveling can not guarantee a perfect result, a manual editing of the photos used for texturing is still needed. Therefore, we presented an application for mask generation. These masks are black-and-white images and are used as indicator for used areas in the photos that are used for texturing of the model. The graphic artist who is editing the photos can use these masks in order to know where the focus of his work has to be.

Last, we presented an application for the generation of the data structures atlas and tile store that are needed for virtual texturing. We could show that our application is significantly faster than existing scripts for this task. In contrast to the scripts, our application further allows the update of atlas and tile store when a consisting photo was changed.

Future Work As we could show in this thesis, our applications for the automatic labeling and leveling can deliver satisfactory results. However, the quality of the resulting textured models highly depends on the quality of the input data. Especially camera registration errors can lead to visible artifacts such as misalignments after the labeling, and massive color shifts after the leveling procedure. In our opinion, also the proposed approach by Gal et al. [13] that introduces shift vectors into the labeling process in order to get rid of the camera registration errors is not the perfect solution. Their approach is only suitable for very small models and a few photos. Still then, it can not guarantee a perfect result, since camera registration errors are not always translational. Therefore, we think that there is still much room for improvement concerning labeling methods.

An interesting way to think about labeling was presented by Dellepiane et al. in [15], who calculate the optical flow for surface regions where projected photos overlap in order to warp these photos together. A global adaption of the camera registrations, so that no visible misalignments in the model are the result, is either very difficult to calculate or even impossible. Therefore, we think that research in the fields of local photo adaption as it is done in the approach proposed in [15] in order to compensate for camera registration errors is the right way to go.

Also the leveling method proposed by Lempitsky and Ivanov in [18] is not without fail. Since the leveling function is only calculated at the vertexes of the model and linearly interpolated in between, this can lead to significantly color differences of adjoining triangles that receive its color information from different photos, also when the vertexes are perfectly leveled. A solution would be an adaptive refinement of the mesh at the borders of regions that are textured by different photos.

Bibliography

[1] Ahmed Abdelhafiz. *Integrating Digital Photogrammetry and Terrestrial Laser Scanning*. PhD thesis, Technical University Braunschweig, 2009.

[2] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[3] Michael Birsak. Workflow optimization for a graphic artist working on large texture sets using virtual texturing. In *Proceedings of the 15th Central European Seminar on Computer Graphics*, pages 35–41, 2011.

[4] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(9):1124 –1137, sept. 2004.

[5] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 23(11):1222 –1239, nov 2001.

[6] Marco Callieri, Paolo Cignoni, Massimiliano Corsini, and Roberto Scopigno. Masked photo blending: Mapping dense photographic data set on high-resolution sampled 3d models. *Computers & Graphics*, 32(3):464–473, 2008.

[7] Edwin Earl Catmull. *A subdivision algorithm for computer display of curved surfaces*. PhD thesis, 1974.

[8] Paolo Cignoni, Massimiliano Corsini, and Guido Ranzuglia. *MeshLab: an open-source 3D mesh processing system*, April 2008.

[9] J. Davis. Mosaics of scenes with moving objects. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, CVPR ’98, pages 354–, Washington, DC, USA, 1998. IEEE Computer Society.

[10] Matteo Dellepiane, Ricardo Marroquim, Marco Callieri, Paolo Cignoni, and Roberto Scopigno. Flow-based local optimization for image-to-geometry projection. *IEEE Transactions on Visualization and Computer Graphics*, 18:463–474, 2012.

[11] Jack Dongarra, Andrew Lumsdaine, Xinhui Niu, Roldan Pozo, and Karin Remington. Sparse matrix libraries in c++ for high performance architectures, 1994.

[12] Alexei A. Efros and William T. Freeman. Image Quilting for Texture Synthesis and Transfer. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 341–346. ACM Press / ACM SIGGRAPH, 2001.

[13] Ran Gal, Yonatan Wexler, Eyal Ofek, Hugues Hoppe, and Daniel Cohen-Or. Seamless montage for texturing models. *Comput. Graph. Forum*, pages 479–486, 2010.

[14] RIEGL Laser Measurement Systems GmbH. http://www.riegl.com, 2011.

[15] Enrico Gobbetti and Fabio Marton. Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Trans. Graph.*

[16] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2003.

[17] V. Kolmogorov and R. Zabin. What energy functions can be minimized via graph cuts? *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(2):147 –159, feb. 2004.

[18] V. Lempitsky and D. Ivanov. Seamless mosaicing of image-based texture maps. In *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, pages 1–6, june 2007.

[19] Anat Levin, Assaf Zomet, Shmuel Peleg, and Yair Weiss. Seamless image stitching in the gradient domain. In *ECCV (4)*, pages 377–389, 2004.

[20] S. Z. Li. *Markov random field modeling in computer vision*. Springer-Verlag, London, UK, 1995.

[21] Lighthouse3d.com. http://www.lighthouse3d.com/tutorials/view-frustum-culling/geometric-approach-testing-boxes-ii/, 2012.

[22] K. Madsen, H.B. Nielsen, and Ole Tingleff. Methods for non-linear least squares problems, 2nd edition, 2004.

[23] Albert Julian Mayer. Virtual texturing. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, October 2010.

[24] David L. Milgram. Computer methods for creating photomosaics. *IEEE Trans. Computers*, 24(11):1113–1119, 1975.

[25] Martin Mittring and Crytek GmbH. Advanced virtual texture topics. In *ACM SIGGRAPH 2008 classes*, SIGGRAPH '08, pages 23–51, New York, NY, USA, 2008. ACM.

[26] Tomas Möller. A fast triangle-triangle intersection test. *journal of graphics, gpu, and game tools*, 2(2):25–30, 1997.

[27] Przemyslaw Musialski, Christian Luksch, Michael Schwärzler, Matthias Buchetics, Stefan Maierhofer, and Werner Purgathofer. Interactive multi-view façade image editing. In *Vision, Modeling and Visualization Workshop 2010*, November 2010.

[28] Patrick Pérez, Michel Gangnet, and Andrew Blake. Poisson image editing. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 313–318, New York, NY, USA, 2003. ACM.

[29] Ruggero Pintus, Enrico Gobbetti, and Marco Callieri. A streaming framework for seamless detailed photo blending on massive point clouds. In *Proceedings of Eurographics Conference - Cultural Heritage Papers*, 32nd Eurographics Conference, Llandudno, Wales, UK. Eurographics, Wiley-Blackwell, April 2011.

[30] ADAM Technology. http://www.adamtech.com.au/, 2012.

[31] Matthew Uyttendaele, Ashley Eden, and Richard Szeliski. Eliminating ghosting and exposure artifacts in image mosaics. In *CVPR (2)*, pages 509–516. IEEE Computer Society, 2001.

[32] Lance Williams. Pyramidal parametrics. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '83, pages 1–11, New York, NY, USA, 1983. ACM.

[33] Peter Wonka, Michael Wimmer, Kaichi Zhou, Stefan Maierhofer, Gerd Hesina, and Alexander Reshetov. Guided visibility sampling. *ACM Transactions on Graphics*, 25(3):494–502, July 2006. Proceedings ACM SIGGRAPH 2006.