FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Implementation and Evaluation of a Fish-Eye Lens for Interactive Visualization of Features in Volumetric Datasets

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Klara Brandstätter
Matrikelnummer 1326465

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Univ.-Doz. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Dipl.-Ing. (FH) Dr. Christoph Heinzl
Dipl.-Ing. Johannes Weissenböck, BSc

Wien, 23. August 2017

Klara Brandstätter

Eduard Gröller

Implementation and Evaluation of a Fish-Eye Lens for Interactive Visualization of Features in Volumetric Datasets

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Klara Brandstätter
Registration Number 1326465

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Univ.-Doz. Dipl.-Ing. Dr.techn. Eduard Gröller
Assistance: Dipl.-Ing. (FH) Dr. Christoph Heinzl
Dipl.-Ing. Johannes Weissenböck, BSc

Vienna, 23rd August, 2017

Klara Brandstätter Eduard Gröller

Erklärung zur Verfassung der Arbeit

Klara Brandstätter
Brunnenweg 12
4921 Hohenzell

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

Wien, 23. August 2017

Klara Brandstätter

Kurzfassung

Die Exploration von Datenvisualisierungen ist eine wichtige Tätigkeit, die längst nicht mehr nur in der Wissenschaft Verwendung findet. Besonders in den Materialwissenschaften und den damit verbundenen Industriebereichen ist die Datenanalyse eine nicht mehr wegzudenkende Methode um mögliche Fehlerquellen oder Schwachstellen aufzudecken, die während des Fertigungsprozesses entstanden sind, und um die notwendigen Qualitätskriterien zu erfüllen. *open_iA* ist eine Open Source Software, die solche Analysen für computertomographische Daten von Materialien durchführen kann.

Diese Bachelorarbeit thematisiert das Thema der Datenerforschung und widmet sich der Implementierung und Integration einer 2D Fish-eye lens in *open_iA*. Die Fish-eye lens wurde wegen ihrer Vergrößerungseigenschaften ausgewählt, und weil sie gleichzeitig Focus und Kontext zur Verfügung stellen kann. Die charakteristische Verzerrung der Fish-eye lens wurde durch eine Thin-Plate Splinetransformation (TPS_2) ermöglicht, die direkt auf den Datensatz angewendet wird.

Neben der Fish-eye lens existieren noch viele weitere so genannte Magic lenses für die unterschiedlichsten Arten von Datensätzen und für verschiedenste Anwendungen. Diese Arbeit gibt einen kurzen Überblick über die gängigsten magischen Linsen. Es folgt eine Definition der Hard- und Softwarevoraussetzungen. Außerdem werden alte und aktuelle Designkonzepte der Fish-eye lens diskutiert. Des Weiteren wird der Algorithmus der Thin-Plate Splinetransformation anhand von Beispieldatensätzen exemplarisch durchgerechnet und erklärt. Abschließend werden Evaluierungsergebnisse der Fish-eye lens anhand von verschiedenen Materialdatensätzen gezeigt, um ihre Funktionalität und Verwendbarkeit zu demonstrieren.

Abstract

The exploration of data visualisations has become an important task and already reaches beyond scientific purposes. Especially, in material sciences and related industries, data analysis is crucial for the detection of possible error sources or weak spots, that occurred during fabrication, to meet the required quality criteria. *open_iA* is an open source software that offers such data analysis of computed tomography material data.

This thesis addresses the topic of data exploration by implementing and integrating a 2D fish-eye lens into *open_iA*. The fish-eye lens has been chosen due to its magnification characteristics, that provide focus and context at the same time. The distinct distortion of the fish-eye lens was achieved by applying a thin-plate spline (TPS_2) transformation to the dataset.

Beside the fish-eye lens, there exist many more magic lenses for different kinds of datasets and purposes, which will be introduced in a few words. Furthermore, the hard- and software requirements for the fish-eye lens as well as general design concepts are defined. Then, a detailed explanation of the thin-plate spline transformation and its algorithm is given and illustrated with example datasets and step-by-step calculations. Finally, the fish-eye lens is tested for its functionality and usability with material datasets.

Contents

Kurzfassung **vii**

Abstract **ix**

Contents **xi**

1 Introduction **1**

2 State of the Art **3**

3 Prerequisites and First Overview **9**
- 3.1 Hard- and Software Components 9
- 3.2 Integration of the Fish-Eye Lens into open_iA 10
- 3.3 Concepts 14

4 Implementation **15**
- 4.1 Used Classes and Functions 15
- 4.2 Exemplary Calculation of the Thin-Plate Spline Transformation 16
- 4.3 Functionality of the Fish-Eye Lens 31

5 Evaluation and Results **35**
- 5.1 Rock Crystal Dataset 35
- 5.2 Pore Dataset 38
- 5.3 Fibre Dataset 40

6 Conclusion **45**

List of Figures **47**

List of Algorithms **51**

Bibliography **53**

CHAPTER 1

Introduction

Nowadays, with the continuous growth of big data, it is getting more and more important to find adequate visualisation techniques [WGK10]. With the possibility to visualize such multidimensional and multivariate data, there also comes the problem of visual representations that are completely overcrowded or cluttered with too much information at once.

Beside zooming and filtering techniques, there exist several concepts to make very complex visualisations clearer and easier to understand. The concept of „Overview first, zoom and filter, then details-on-demand“by Shneiderman [Shn96] is well known in visualization. Since there are often difficulties with insufficient space for showing overview and detail at once, the concept of ‚focus and context‘ tries to resolve this by focusing on small parts of currently relevant information while keeping the context at a less detailed view[HS17, CKB08, LA94, Hau06]. If the datasets are huge, it is also possible to provide multiple coordinated views [Rob07], so the information can be observed from different perspectives. A concept called ‚Linking and Brushing‘ offers linked representations of different aspects of a dataset. Whenever features in one representation are selected (brushed) or edited, all the other representations are updated as well. The concept of ‚Interactive Steering‘ allows the adjustment of parameters during processes of data acquisition or running simulations to change the outcome or to direct it into preferred regions of interest [HS17].

Another important concept, which this thesis will describe, are interactive or so called magic lenses. Such lenses offer different visual representations of the underlying data.

Typically, they are circularly shaped, and it has to be possible to alter position and size of the lens to guarantee a flexible data exploration. The orientation of a magic lens is another important factor, but concerns mostly 3D lenses. For 2D circular lenses orientation would not be meaningful.

The decisive part of an interactive lens is its lens function. This lens function is responsible for the intended effect the lens should have on the data. It consists of calculations that are applied to the data in order to achieve the intended visual results. A lens function can intervene in any stage of the visualisation pipeline. It can either be used to manipulate pixels in the view stage or to process the values of the data source directly.

Furthermore, what has to be considered is, which parts of the data are affected by the lens. In general, this concerns data underneath the lens, and often just a subset of the data to make calculations that the lens performs faster.

In the end, the data underneath the lens and the base visualisation tool have to be joined to provide a convincing feedback, in order to allow the user to understand how the view of the lens relates to the original data [BMe+14].

Beside simple magnification lenses, there also exist complex lenses which modify the data underneath only slightly so that certain requested details can be emphasized for exploration. Modifications can reach from filtering particular information from the data to making hidden features visible by moving occluding information aside. Although a magic lens modifies the underlying information, it should never falsify it. Therefore, the choice which kind of lens is best suited for specific data has to be considered carefully.

In this thesis, the implementation of a 2D fish-eye lens into the open source visualisation and analysis software open_iA[1] is presented. The algorithm of the fish-eye lens is explained in detail, and results with different datasets are depicted and discussed. The advantage of a fish-eye lens is, that it magnifies the data around a selected point without losing the context of the whole data. This is done by softening the distortion, which is responsible for the magnification, towards the boarders of the lens, so that a smooth transition between the data inside and outside the lens can be achieved.

In Chapter 2, an overview of state of the art magic lenses is given, Chapter 3 introduces the used hardware and software components as well as concepts that lead to the final version of the fish-eye lens. The implementation of the lens, which is based on a thin-plate spline transform, is shown using a systematic step-by-step algorithm run in Chapter 4. Chapter 5 presents the results that have been realized by applying the fish-eye lens to different computed tomography (CT) datasets of carbon and glass fibre reinforced polymers (CFRP, GFRP) and a CT dataset of a rock crystal. Finally, Chapter 6 concludes the thesis with a discussion on the findings and some remarks on future work in the field.

---

[1] `http://www.3dct.at/cms2/index.php/en/software-en/open-ia-en`

CHAPTER 2

State of the Art

From 2D to 3D there is a huge variety of magic lenses, with different possibilities of user interaction and data representation. In this chapter, selected types of existing lenses an their field of application are presented.

When analysing time series, it is often necessary to eliminate seasonal effects, in order to normalize the data or to do general temporal transformations. Time series are important in every domain, from finance over engineering to scientific disciplines. Temporal data does not depend on other variables. Is uniform and absolute. Therefore, it should be treated differently than other data. So called *ChronoLenses* are meant to support users in their tasks by offering on-the-fly transformations of data points in the focus area and integration of visual analysis, e.g. of quantitative or derived data [ZCPB11]. In Figure 2.1 a *ChronoLens* is visible.

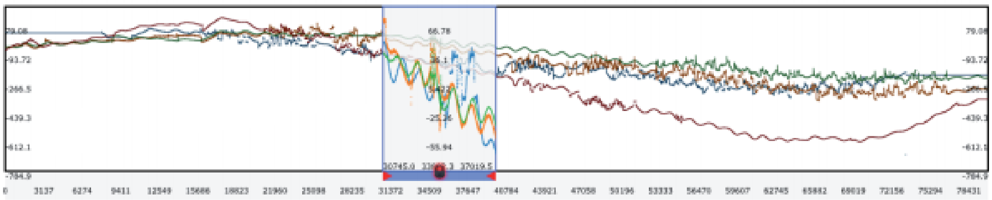


Figure 2.1: The *ChronoLens* is transforming the underlying content for better exploration.

Maps are an established technique to visualize geo-spatial data. However, following driving directions on maps is often a difficult task since such maps are densely packed with information. Karnick et al. [KCJ+10] present *Detail Lenses*. They encircle points of interest (POI) along a route on the map (see Figure 2.2). Those POIs are shown in detail through the lenses, whereas the rest of the map is shown as an overview. It is important that those lenses do not occlude each other or the route. The correct layout of the lenses on the map is therefore a crucial aspect.

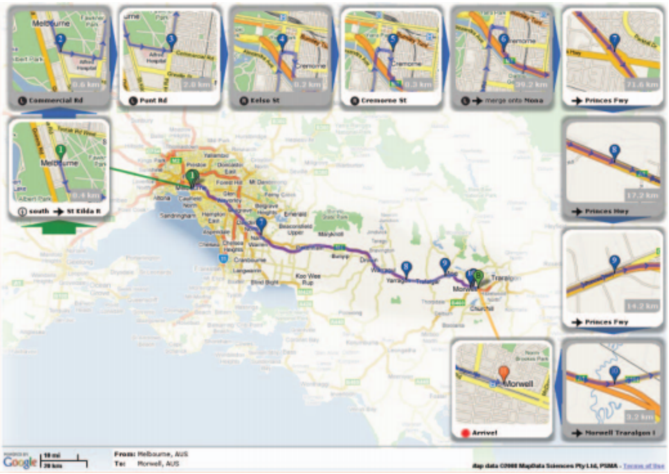


Figure 2.2: Several *Detail Lenses* are arranged around the centre of the map and magnify important POIs along the route.

Volume datasets can pose some obstacles in visualisation as well. Apart from the difficulties of spatial selection and occlusion, also the datasets in science and medicine have been growing in size enormously, while the screen resolution cannot keep up. Interactively configurable *Magic Volume Lenses*, embedded into the volume rendering, can offer magnification lens rendering techniques in a focus + context framework. Features of interest are emphasized and the rest of the volume data is compressed but not clipped entirely, so the user does not lose the context. In Figure 2.3 a *Magic Volume Lens* is applied to a direct volume rendering (DVR) of an engine [WZMK05].

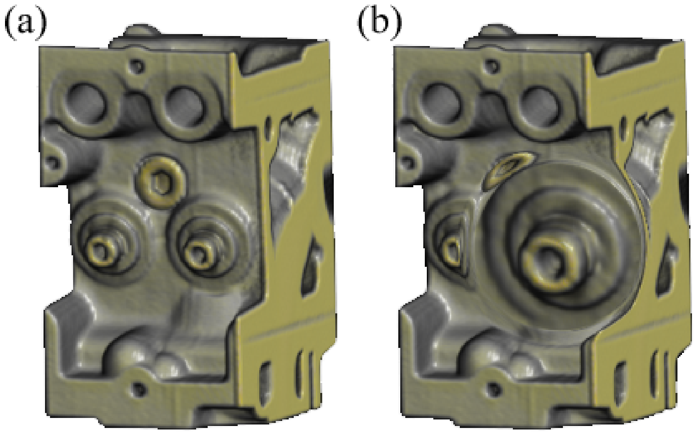


Figure 2.3: DVR of an engine without (a) and with (b) magnification by the embedded *Magic Volume Lens*.

Data, which employs special lenses is 2D and 3D flow data. Beside the vectors, several additional aspects, such as derived scalar attributes, vortices or flow topology, have to be considered when developing appropriate lenses. Gasteiger et al. [GNBP11] propose the *FLOWLENS*, a lens designed especially for blood flow and derived data, to explore cerebral aneurysms and ruptures (see Figure 2.4). This lens has to consider hemodynamics such as the wall shear stress (WSS) and the inflow jet, and of course, it needs to offer a visualisation preferably free from unwanted distortions, occlusions or clutter.

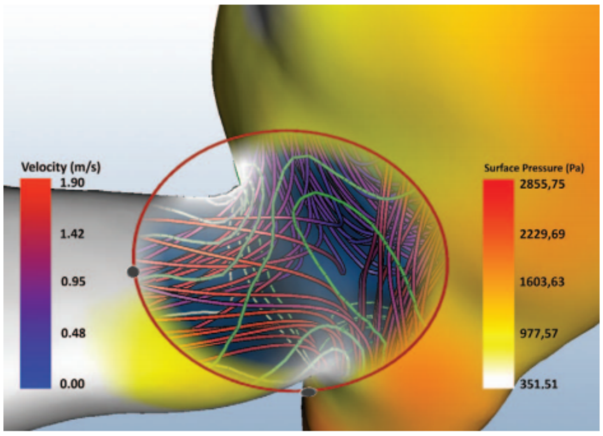


Figure 2.4: The *FLOWLENS* visualizes the blood flow inside an aneurysm with the help of streamlines. The flow pressure is depicted by the green saturation-coded contour lines.

The *Sampling Lens* by Ellis et al. [EBD05] is particularly useful for multidimensional and multivariate data. This circular lens performs random sampling on scatter plots and parallel coordinate systems, giving the user the possibility to discover hidden trends or patterns without losing the data context. In Figure 2.5 the *Sampling Lens* is applied to a parallel coordinate system.

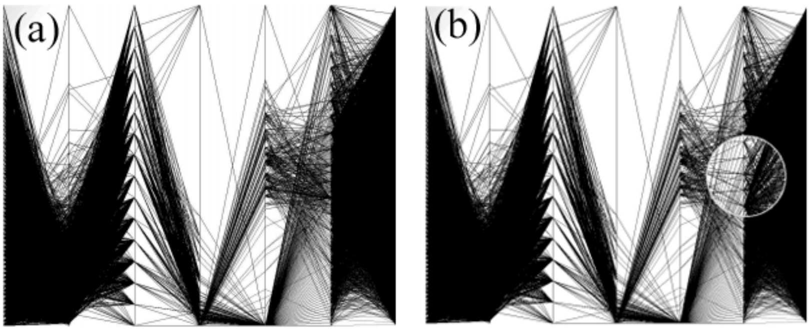


Figure 2.5: The strongly saturated parallel coordinate system makes it difficult to perceive the lines on the left and right side (a). The *Sampling Lens* offers a clearer view of the dense areas to perceive any hidden trends (b).

Texts and documents often consist of a mass of data. Querying and the focus on specific topics (e.g. in a word cloud) are popular ways of representing such data. However, these approaches do not consider the meanings of words, although many of the words could be connected to each other by meronyms (part-of relations). Meronyms show a spatial relationship since they are a part of a whole entity. Chang and Collins [CC13] propose 3D models that contain meronyms of the model on their respective position. On such a model a lens can be applied. The 3D model's meronyms can be emphasized with the lens. Around the lens additional corresponding keywords are depicted as heatmap charts and the respective text documents, including the words (meronyms) imaged inside the lens, can be displayed as well. Figure 2.6 shows such a lens applied to a 3D model of a car.

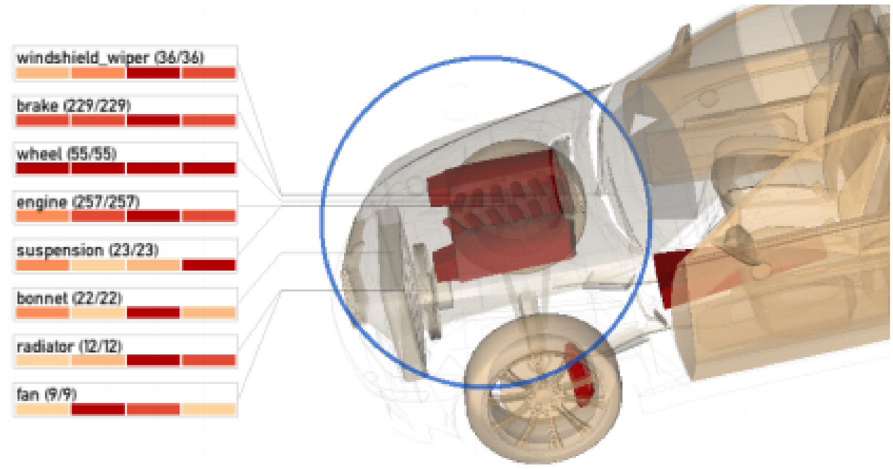


Figure 2.6: The car parts inside the lens are listed next to it. The heatmap chart underneath every list entry describes the frequency (red = very frequent, beige = less frequent) in which the names of the car parts appeared in certain documents in an interval of 4 months.

A common problem with graph data is its ambiguity and edge congestion. The *EdgeLens* [WCG03] address edge congestion among other things. This lens reduces the number of edges inside the lens by curving edges around the lens (see Figure 2.7). This is done for all edges which do not have their nodes inside the lens. With this technique it is possible to preserve the node layout but still gain the necessary hidden information.

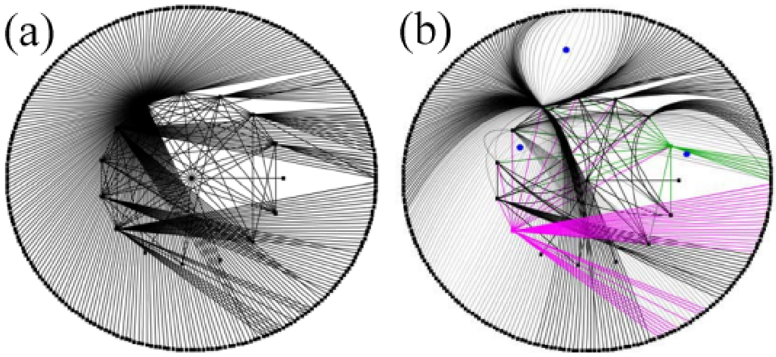


Figure 2.7: Graph data without (a) and with the application of three *EdgeLenses* (b). The blue points mark the centre of the lenses. The green and pink coloured edges are ignored by the *EdgeLenses* and are not curved around the lenses.

Apart from developing lenses for specific datasets, lenses are also designed to fulfil particular tasks. They can be used to select POI in complex datasets, or reconfigure data underneath the lens so the user gets a view at a new layout and gains insight from a different perspective. Reconfiguration offers e.g. the *Layout Lens* by Tominski et al. [TAS09], which reallocates graph nodes and creates overviews of the local neighbourhood for the analysis of node connectivity (see Figure 2.8). Furthermore, there are lenses for filtering, to only show desired information, or lenses that offer more or less detail in a selected region.

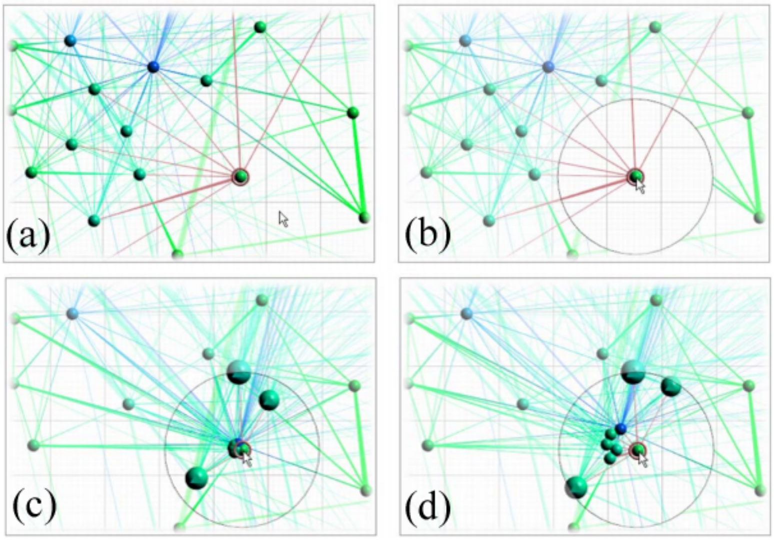


Figure 2.8: The focus lies on the red-rimmed node (a). With a local edge lens, edges that do not belong to the focused node are removed (b). Then, the *Layout Lens* gathers all nodes adjacent to the focused node (c). Accumulated nodes in the centre are spread by a fish-eye lens (d).

Fish-eye lenses are very well suited for showing more detail through magnification. Their main fields of application are found in the exploration of geo-spatial data and graph data [BMe+14]. In the early 90's, Sakar and Brown already used fish-eye views for graphs, because of the mentioned advantages of local detail and global context within one view [SB92, SB94]. In Figure 2.9 a fish-eye lens is applied to a map.[1]
As shown in the subsequent sections of this thesis, a fish-eye lens can also be used for 2D data as a context-preserving magnification lens.

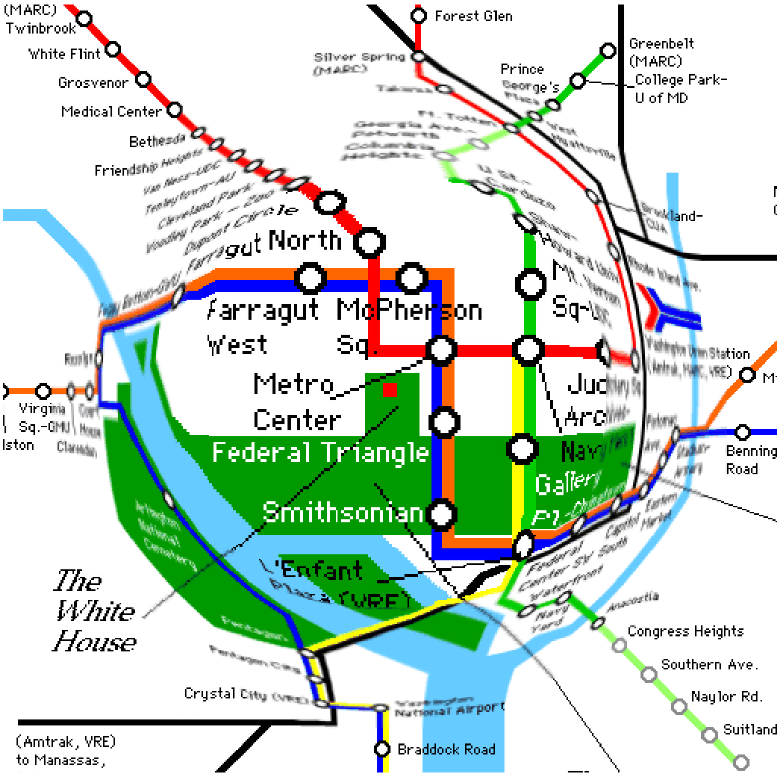


Figure 2.9: Cutout of the subway map of Washington D.C. The applied fish-eye lens magnifies the focused area in the middle. On the border of the focused area, the map is strongly distorted by the lens to permit a smooth transition between the focused area and the context.

---

[1] `http://www.cs.umd.edu/class/fall2002/cmsc838s/tichi/fisheye.html`, Accessed: 20.08.2017

CHAPTER 3

Prerequisites and First Overview

3.1 Hard- and Software Components

The code for the fish-eye lens was implemented in C++ on an Intel® Core™ i7-4720HQ 2.60 GHz processor with a Windows 10 Home© 64-bit operating system. Microsoft Visual Studio[1] Community 2015 (Version 14.0.25425.01 Update 3) was used as integrated development environment (IDE).

3.1.1 open_iA

The fish-eye lens is an additional feature which is integrated into the framework of the open source software open_iA[2]. open_iA supports users in exploring and analysing computed tomography (CT) datasets by offering visual analysis and data processing tools, mainly focused on volumetric and polygonal datasets. The core of open_iA consists of functionalities such as loading of volumetric datasets as well as displaying them in different views and applying transfer functions. It is possible to load datasets using a variety of file formats - Raw images (*.raw, *.rec, *.vol), MetaImage (*.mhd, *.mha), STL files or VGI files, just to name a few. During the implementation of the fish-eye lens, MetaImage data from a rock crystal and from pores of fibre-reinforced polymers (FRPs) have been used for testing.
When a dataset is loaded into open_iA the core provides a 3D rendering view and three 2D axis-aligned slice views of the volumetric data. Also a histogram view can be used to set transfer functions which are then assigned to the data in the 3D renderer and the three 2D slicer views. To examine the various intensities of a dataset, a profile plot is used that depicts the intensities along a line through the dataset. Furthermore, it is possible to load more than one volumetric dataset into the 3D renderer.

---

[1] `https://www.visualstudio.com/`, Accessed: 20.08.2017

[2] `https://github.com/3dct/open_iA`, Accessed: 20.08.2017

Beside the core functionalities, open_iA integrates several additional image processing filters as well as analysis and visualization tools[3] for different scenarios. These tools, for example, allow the exploration of multi-modal and multi-scalar data, or facilitate the analysis of data from FRPs and help determining their porosity. Furthermore, they facilitate the detection and classification of defects in glass fibre-reinforced polymers (GFRPs), and the parameter space of multi-channel segmentation algorithms can be explored visually as well [FMH16].

3.1.2 Libraries

open_iA is based on the open-source application development framework *Visualisation Toolkit*[4] (VTK) Version 7.0.0 from Kitware collection [SML06]. VTK offers a C++ library with a diversity of algorithms for 3D computer graphics, image processing and visualization.

The `vtkThinPlateSplineTransform`[5] class of the VTK library is primarily responsible for the characteristic distortion of the implemented fish-eye lens. Detailed information about this class and other employed VTK classes are discussed later.

The *Insight Segmentation and Registration Toolkit*[6] (ITK) Version 4.10.0 offers segmentation and registration algorithms for multidimensional data [JMI15]. Its libraries include many tools for image analysis which are used or adapted by open_iA. Since ITK is not needed for the implementation of the fish-eye lens, it will not be explained any further.

The cross-platform GUI-Toolkit *Qt*[7] Version 5.6.0 supports the open_iA framework with an easily usable graphical user interface. *Qt* is needed for the fish-eye lens for general user interactions such as the keyboard inputs and mouse movements. Furthermore, it provides widgets to seamlessly integrate VTK.

First implementation attempts of the fish-eye lens aimed at using Qt-based approaches for creating the lens. The difficulties and the undesirable outcomes, such as flickering of the lens and poor performance, that came with the `QtWidget`[8] and `QVTKWidget2`[9] classes (for the shape of the lens) lead to discarding this idea.

3.2 Integration of the Fish-Eye Lens into open_iA

The fish-eye lens is a feature that extends the core functionality of open_iA. It is a 2D circular lens that can be activated separately in all of the three 2D slice views as soon as a volumetric dataset has been loaded into open_iA.

By clicking with the `left mouse button` inside the XY-, the YZ- or the XZ-slice view

---

[3] `https://github.com/3dct/open_iA/wiki/Tools`, Accessed: 20.08.2017

[4] `http://www.vtk.org/`, Accessed: 20.08.2017

[5] `http://www.vtk.org/doc/nightly/html/classvtkThinPlateSplineTransform.html`, Accessed: 20.08.2017

[6] `https://itk.org/`, Accessed: 20.08.2017

[7] `https://www.qt.io/`, Accessed: 20.08.2017

[8] `http://doc.qt.io/qt-5/qwidget.html`, Accessed: 20.08.2017

[9] `http://www.vtk.org/doc/nightly/html/classQVTKWidget2.html`, Accessed: 20.08.2017

and by pressing the `O-key` on the keyboard the fish-eye lens is activated. While the lens stays enabled, it follows the movements of the mouse cursor. Pressing the `O-Key` again, the fish-eye lens is disabled and vanishes. Figure 3.1 shows a 2D slice of the pore dataset (pores.mhd), before and after activating the fish-eye lens. The centre of the lens is given by the position of the mouse cursor. The fish-eye lens distortion which is applied to the image data in Figure 3.1(b) can be seen clearly compared to the non-distorted data in Figure 3.1(a).

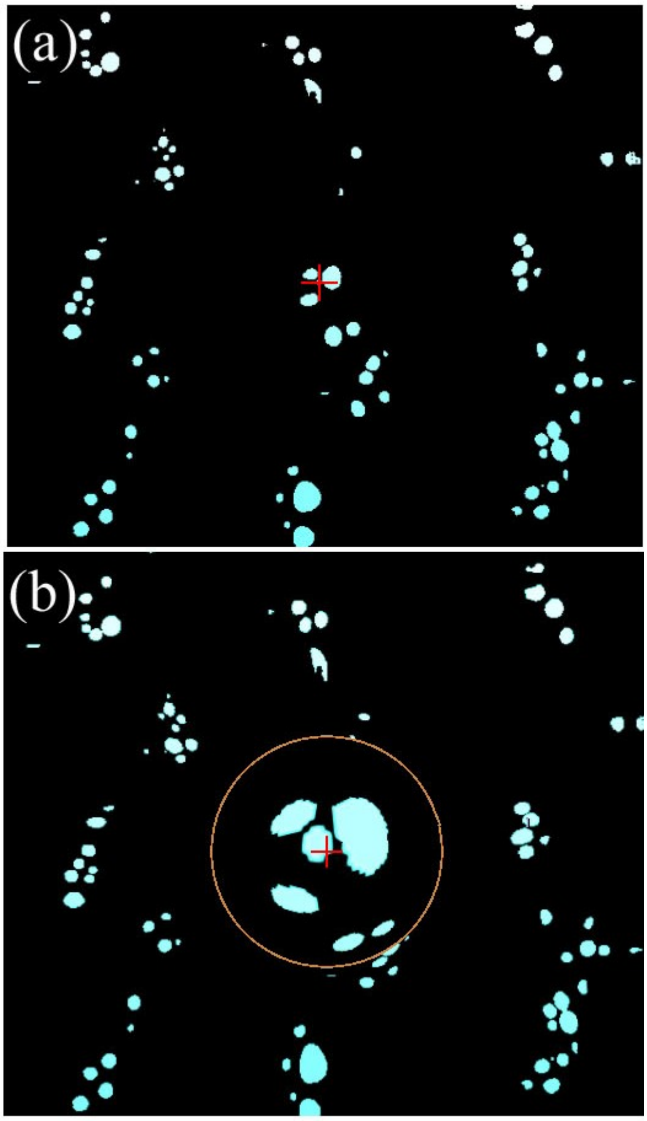


Figure 3.1: XZ-slice view of the pore dataset (pores.mhd) before (a) and after (b) activating the fish-eye lens (a 'cyan' colour transfer function has been applied to the data for better visibility). The orange circle represents the fish-eye lens.

Note: The context of the image data does not get lost, since the fish-eye lens provides a smooth transition from the magnified centre of the lens to the normal unmagnified data outside the circle.

The size of the fish-eye lens radius and also the degree of distortion can be adjusted by means of keyboard input. Pressing the `plus-key` increases the radius of the fish-eye lens and pressing the `minus-key` decreases the radius respectively. Changing the degree of the lens distortion works similar. The key combination `STRG + plus` magnifies the distortion, whereas `STRG + minus` reduces it. The larger the lens radius gets, the less the influence of the distortion becomes, which means that the distortion might need to be adjusted again. This is due to the increasing radius. A large lens, however, can be distorted much more, than a smaller lens, because small lenses, cannot offer so many magnifying distortion degrees, without corrupting the image data.

The default radius of the lens is 80.0 pixels. It can be scaled down to a minimal radius of 2.0 pixels (for very small datasets) and scaled up to a maximal radius of 220.0 pixels. The distortion degrees go from no distortion at all to a strong distortion that only just delivers satisfying results without confusing the user too much. The degrees of the distortion are actually based on a resizeable radius too, that is simply invisible for the users. The principles on how the distortion works exactly are given later during the implementation details.

The lens radius and the distortion degree settings are stored between disabling and enabling of the fish-eye lens, as long as the open_iA core is running. Figure 3.2(a) and Figure 3.2(b) show examples of a fish-eye lens with a radius of 150.0 pixels and different distortion or magnification degrees.

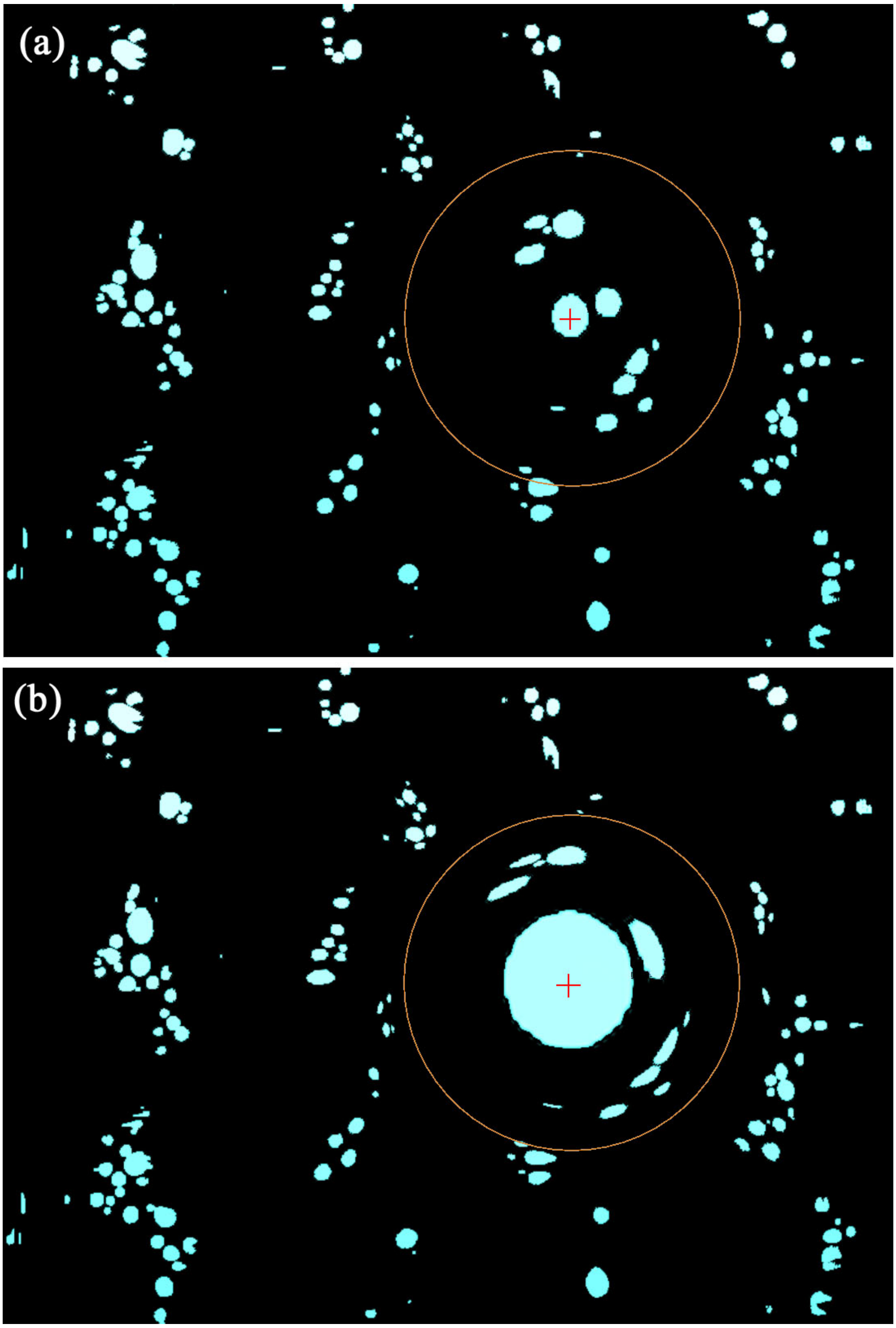


Figure 3.2: XZ-slice view of the pore dataset (pores.mhd) with a fish-eye lens radius of 150.0 pixels and very little distortion (a) and strong distortion (b). The orange circle represents the fish-eye lens.

3.3 Concepts

The open_iA core features a rectangular *magic lens* that can be enabled in the three slice views. When two datasets have been loaded into open_iA, this *magic lens* makes it possible show one of the two datasets blended in on top of the other dataset. The *magic lens* basically consists of an `iAFramedQVTKWidget2`, which is a subclass of the `QVTKWidget2`. With the `QVTKWidget2` a *VTK* render window can be displayed inside a *Qt* window. It inherits the `QGLWidget`[10] class from the *Qt* library. The `QGLWidget` can be used like any other `QWidget`, but offers additional commands for *OpenGL* rendering.

The initial approach for the fish-eye lens was based on the classes used for the rectangular *magic lens* implementation. The idea was, to use the `iAFramedQVTKWidget2` as the lens and let the transformation (distortion) only happen within this widget. The widget would be then overlaid onto the original data. However, this attempt was very error-prone, and despite correct calculations (in external test slicer views), the transform never worked properly inside the widget for the lens.

Another problem was the rectangular shape of the widget. The typical shape of a lens is circular and not rectangular, but since *Qt* does not offer circular widgets, it would have led to additional effort to find a solution for programming a seemingly circular widget, that would not cause too much performance loss.

Due to these difficulties, it was decided to drop the initial approach and to come up with a new one.

It turned out, that not using any widget at all for the lens, worked well: The transform for the fish-eye lens is now applied on the whole 2D data available in the slice view. By setting the source and target landmarks for the transform accordingly across the slice image, the distortion can be narrowed down to a given radius. The positioning of the landmarks is a crucial step towards a convincing visual representation of the lens. On the one hand, with too few fixed landmarks, the distortion affects on its borders most of the image data which leads to a very turbulent distortion while moving the lens. On the other hand, too many fixed landmarks lead to long processing times and a juddering lens, even though the distortion can be controlled very well in a defined radius. A detailed image of the fish-eye lens with visible source and target landmarks is depicted in the next Chapter in Figure 4.2. Finally, for depicting the actual lens, a simple circle with that given radius is drawn around the centre of the distortion.

---

[10] `http://doc.qt.io/qt-4.8/qglwidget.html`, Accessed: 20.08.2017

CHAPTER 4

Implementation

4.1 Used Classes and Functions

This section gives a brief overview of the classes and functions that are used for the implementation of the fish-eye lens, including classes and functions from open_iA as well as from the *VTK* library.
The code for the fish-eye lens transform is entirely integrated into the **`iASlicerWidget`** class. The `iASlicerWidget` class is responsible for several interactions with the three 2D slice views of open_iA. As the fish-eye lens is an additional feature for the slice views, it has been decided to integrate the code there. Inside the `iASlicer Widget` class, two new functions are responsible for the initialisation and the functionality of the fish-eye lens:

- `initializeFisheyeLens()` is responsible for initializing the `vtkThinPlateSplineTransform`. The `vtkThinPlateSplineTransform` is responsible for calculating the distortion based on defined source and target landmarks. Detailed information about how the transform works is given in the following section. Furthermore, source and target landmarks and corresponding circles for visualizing them (for testing purposes) are initialized, as well as a circle variable representing the lens. It sets the respective mappers for the actors of the renderer of the slice views. The actors are needed for depicting the lens in the slice view and they were also helpful during implementation for showing the positions of the landmarks.

- `updateFisheyeTransform()` is doing the actual work and updates the transform whenever the mouse moves. It sets new source and target landmarks and passes them to the `vtkThinPlateSplineTransform` which is then applied to the data in the current slice view.

Two already existing functions have been extended to provide interaction with the fish-eye lens:

- `keyPressEvent()` handles keyboard input and processes the enabling and disabling of the fish-eye lens by pressing the `O-key`. It is also responsible for increasing and decreasing the lens radius and the distortion.
- `mouseMoveEvent()` updates the fish-eye transform whenever the mouse moves (as long as the lens is enabled).

The **`iASlicerData`** class stores information about the data in the slice views and offers multiple operations that can be executed on the data. A getter (`getSliceNumber()`) for the number of the current data slice that is visible in the slicer has been added. The number of the slice is important for calculating the correct landmarks in the different slice views in the `updateFisheyeTransform()`-function.

4.2 Exemplary Calculation of the Thin-Plate Spline Transformation

Thin-plate splines (TPS) are a special case of polyharmonic splines. They are used for data interpolation and smoothing [Duc77].
The `vtkThinPlateSplineTransform` is a non-linear warp transform. It requires a set of source and target landmarks that define the shape of the transform. Points on a mesh that are close to a source landmark are moved to a place near to a corresponding target landmark. Points that are not lying on the mesh are interpolated smoothly.
The data in the slice views is in 2D, meaning the XY-slices have constant Z-values, the YZ-slices have constant X-values and the XZ-slices have constant Y-values for every landmark on the slice. Therefore, an appropriate radial basis function (RBF) kernel has to be used for calculating a correct thin-plate spline warp. The default kernel for 2D is the *R2LogR* kernel (in VTK the respective function is `SetBasisToR2LogR()`). For 3D data the *R* kernel has to be applied (`SetBasisToR()`). Furthermore, it would be possible to specify a user-defined radial basis function, but this would also mean that the transform no longer was a true thin-plate spline transform.
The interpolation and also the notation in the source code of the `vtkThinPlateSplineTransform` has been inspired by Bookstein's Thin Plate Spline algorithm [Boo97, Boo89] and by the online work published by Tim Cootes, Professor of Computer Vision at the University of Manchester.
With the aid of the `vtkThinPlateSpline.cxx` file and the corresponding header file by Ken Martin, Will Schroeder and Bill Lorensen, the functionality of the thin-plate spline transform is explained. The naming of variables in this chapter is based on the naming from the source code and Bookstein's articles.

Apart from the header file, the `vtkThinPlateSpline.cxx`[1] additionally includes the `vtkMath.h`[2], the `vtkObjectFactory.h`[3] and the `vtkPoints.h`[4] header.
`vtkMath.h` is needed for common math operations such as vector and matrix operations (conversions from degrees to radians or the provision of constants). `vtkObjectFactory.h` creates vtk objects. `vtkPoints.h` is used for the representation of 3D points. Its data model is an array of x-y-z triplets and can be accessed by point or cell ID. The `vtkPoints.h` is essential for the definition of the source and target landmarks.
The `vtkThinPlateSpline` class is a subclass of `vtkWarpTransform`[5]. This superclass supports nonlinear geometric transformations. `vtkWarpTransform` is again a subclass of `vtkAbstractTransform`[6], which is the superclass for all geometric VTK transforms, such as warp transforms and also homogeneous (linear) transforms.
On the following pages, the algorithm of the thin-plate spline transform is explained in detail:
The radial basis function is defined as $U(r) = r^2 * ln(r)$. Because of r^2 the thin-plate spline can be denoted as TPS_2. Depending on the exponent of r several TPS_n are possible for all n, where n is even. However, this algorithm only uses TPS_2.
Bookstein uses the notation *log* instead of *ln* in his description of the thin-plate spline algorithm. The notation for the logarithm in the source code by Schroeder et al. is also *log*, since the *log* function computes the natural logarithm to the base-e in C++. To avoid confusion between the familiar notation *log* for the base-10 logarithm and the notations of Bookstein and Schroeder et al., this thesis refers to the natural logarithm as *ln*.
Let N be the number of source landmarks (in the current implementation $N = 32$) and D the constant dimension with $D = 3$.
At first, a few matrices have to be defined: $W[rows][columns]$ is the output weights matrix with $rows = N + D + 1$ and $columns = D$. W is the essential matrix that is necessary for the transformation.
The size of the combined linear rotation and scale matrix is given by $A[3][3]$ and the linear translation is given by the vector $c[3]$.
Furthermore, the following input matrices are needed: $L[N + D + 1][N + D + 1]$ and $X[N + D + 1][D]$. With the help of these matrices, the weights matrix W can be calculated in a later step.

---

[1] `https://github.com/Kitware/VTK/blob/master/Common/Transforms/vtkThinPlateSplineTransform.cxx`, Accessed: 20.08.2017

[2] `http://www.vtk.org/doc/nightly/html/classvtkMath.html`, Accessed: 20.08.2017

[3] `http://www.vtk.org/doc/nightly/html/classvtkObjectFactory.html`, Accessed: 20.08.2017

[4] `http://www.vtk.org/doc/nightly/html/classvtkPoints.html`, Accessed: 20.08.2017

[5] `http://www.vtk.org/doc/nightly/html/classvtkWarpTransform.html`, Accessed: 20.08.2017

[6] `http://www.vtk.org/doc/nightly/html/classvtkAbstractTransform.html`, Accessed: 20.08.2017

The matrix L consists of four submatrices:

$$L = \begin{Vmatrix} K & P \\ P^T & O \end{Vmatrix} \tag{4.1}$$

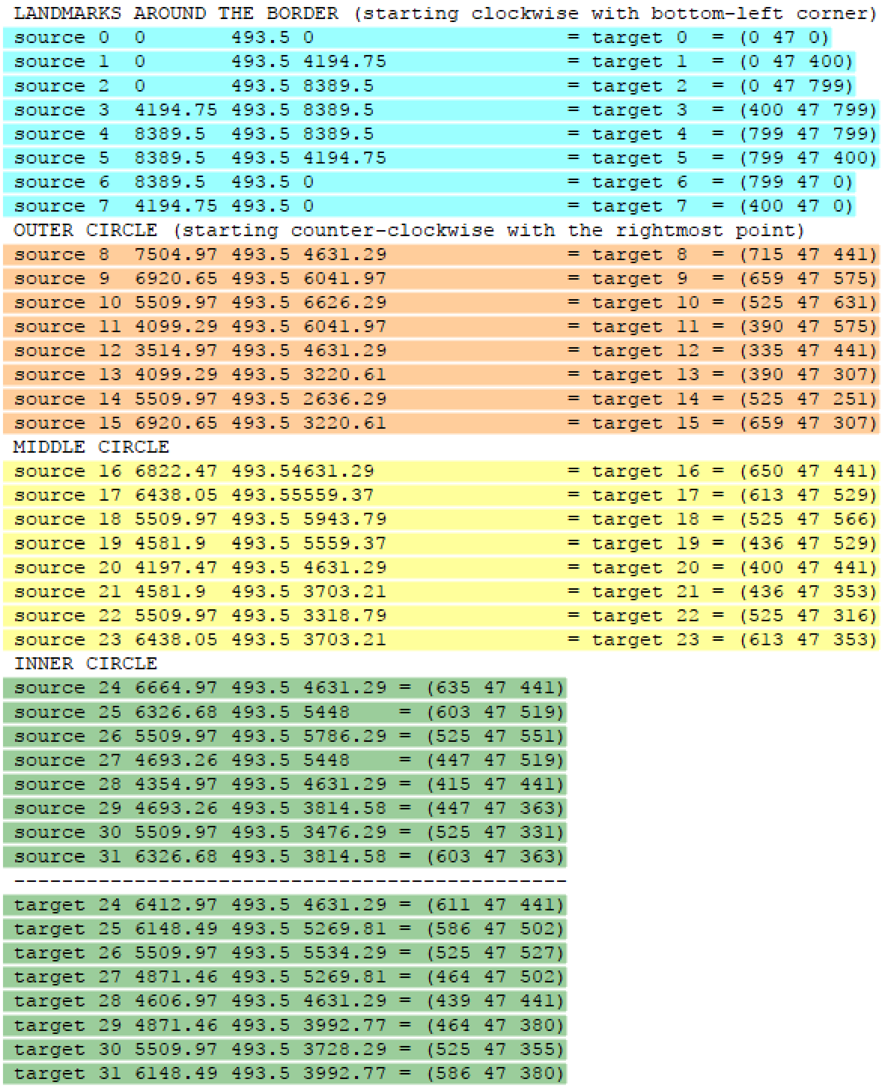
It has to be noted, that the bottom-right $(N \times (D+1))$-submatrix O consists of zeros. P^T is the transposed matrix of P. P consists of 1's and the 32 source landmark coordinates that are listed in Figure 4.1.

$$P = \begin{Vmatrix} 1 & x_0 & y_0 & z_0 \\ 1 & x_1 & y_1 & z_1 \\ ... & ... & ... & ... \\ 1 & x_{N-1} & y_{N-1} & z_{N-1} \end{Vmatrix} = \begin{Vmatrix} 1 & 0 & 493.5 & 0 \\ 1 & 0 & 493.5 & 4194.75 \\ ... & ... & ... & ... \\ 1 & 6326.68 & 493.5 & 3814.58 \end{Vmatrix}, (D+1) \times N, and \tag{4.2}$$

$$P^T = \begin{Vmatrix} 1 & 1 & ... & 1 \\ 0 & 0 & ... & 6326.68 \\ 493.5 & 493.5 & ... & 493.5 \\ 0 & 4194.75 & ... & 3814.58 \end{Vmatrix}, (N) \times (D+1). \tag{4.3}$$

Since the landmarks from Figure 4.1 have been calculated in the XZ-slicer view, the Y-coordinate is always the same. The coordinates in the first two rows P correspond to the bottom-left and the middle-left border landmarks (red circles) in Figure 4.2, the coordinate of the last row corresponds to the 31$^{\text{st}}$ landmark (red circle) at the innermost landmark circle. In Figure 4.2, the small red circles show the position of the source landmarks, the green dots the position of the target landmarks.

The reasons why it has been decided to use 32 source and target landmarks are explained as follows: The 8 source and target landmarks around the border of the slicer image make sure that pixels far outside the fish-eye lens are not so much affected by the TPS$_2$ anymore. If there were only 4 landmarks at each corner of the image, moving the fish-eye lens towards the edges of the image would result in edges that are strongly bent inwards because of the TPS$_2$. Another 8 source and target landmarks are needed to define the region of the distortion by the fish-eye lens (yellow numbered landmarks in Figure 4.2). Fewer landmarks would lead to an angular shaped fish-eye distortion. These 16 landmarks alone do not suffice to prevent the part of the image outside the fish-eye lens from strong distortions due to the TPS$_2$. This is why two additional stabilizing source and target landmark circles (with 8 landmarks each) are arranged around the fish-eye lens. They keep the distortion restricted. Adding more of these stabilizing circles would lead to even better results concerning the restricted distortion, but it would also result in a considerable loss in performance.



```
LANDMARKS AROUND THE BORDER (starting clockwise with bottom-left corner)
source 0  0       493.5 0                     = target 0  = (0 47 0)
source 1  0       493.5 4194.75               = target 1  = (0 47 400)
source 2  0       493.5 8389.5                = target 2  = (0 47 799)
source 3  4194.75 493.5 8389.5                = target 3  = (400 47 799)
source 4  8389.5  493.5 8389.5                = target 4  = (799 47 799)
source 5  8389.5  493.5 4194.75               = target 5  = (799 47 400)
source 6  8389.5  493.5 0                     = target 6  = (799 47 0)
source 7  4194.75 493.5 0                     = target 7  = (400 47 0)
OUTER CIRCLE (starting counter-clockwise with the rightmost point)
source 8  7504.97 493.5 4631.29               = target 8  = (715 47 441)
source 9  6920.65 493.5 6041.97               = target 9  = (659 47 575)
source 10 5509.97 493.5 6626.29               = target 10 = (525 47 631)
source 11 4099.29 493.5 6041.97               = target 11 = (390 47 575)
source 12 3514.97 493.5 4631.29               = target 12 = (335 47 441)
source 13 4099.29 493.5 3220.61               = target 13 = (390 47 307)
source 14 5509.97 493.5 2636.29               = target 14 = (525 47 251)
source 15 6920.65 493.5 3220.61               = target 15 = (659 47 307)
MIDDLE CIRCLE
source 16 6822.47 493.54631.29                = target 16 = (650 47 441)
source 17 6438.05 493.55559.37                = target 17 = (613 47 529)
source 18 5509.97 493.5 5943.79               = target 18 = (525 47 566)
source 19 4581.9  493.5 5559.37               = target 19 = (436 47 529)
source 20 4197.47 493.5 4631.29               = target 20 = (400 47 441)
source 21 4581.9  493.5 3703.21               = target 21 = (436 47 353)
source 22 5509.97 493.5 3318.79               = target 22 = (525 47 316)
source 23 6438.05 493.5 3703.21               = target 23 = (613 47 353)
INNER CIRCLE
source 24 6664.97 493.5 4631.29 = (635 47 441)
source 25 6326.68 493.5 5448    = (603 47 519)
source 26 5509.97 493.5 5786.29 = (525 47 551)
source 27 4693.26 493.5 5448    = (447 47 519)
source 28 4354.97 493.5 4631.29 = (415 47 441)
source 29 4693.26 493.5 3814.58 = (447 47 363)
source 30 5509.97 493.5 3476.29 = (525 47 331)
source 31 6326.68 493.5 3814.58 = (603 47 363)
-----------------------------------------------
target 24 6412.97 493.5 4631.29 = (611 47 441)
target 25 6148.49 493.5 5269.81 = (586 47 502)
target 26 5509.97 493.5 5534.29 = (525 47 527)
target 27 4871.46 493.5 5269.81 = (464 47 502)
target 28 4606.97 493.5 4631.29 = (439 47 441)
target 29 4871.46 493.5 3992.77 = (464 47 380)
target 30 5509.97 493.5 3728.29 = (525 47 355)
target 31 6148.49 493.5 3992.77 = (586 47 380)
```

Figure 4.1: IDs' and coordinates in physical space (bounds) and image space (extent) of the 32 source and target landmarks depicted in Figure 4.2. The blue, orange and yellow landmarks have identical source and target positions and serve as stabilisation for the transform. The green source and target landmarks are responsible for the actual lens distortion. Because of the XZ-slice, the Y values are constant. The values between the brackets (x, y, z), represent the same landmark coordinates but in image space.

For the submatrix K the distances between two source landmarks, P_i and P_j, are needed, which can be written as $r_{ij} = |P_i - P_j|$. The distance r_{ij} is the variable for the radial basis function $U(r_{ij})$.
The resulting submatrix K looks as follows:

$$K = \begin{Vmatrix} 0 & U(r_{0,1}) & ... & U(r_{0,N-1}) \\ U(r_{1,0}) & 0 & ... & U(r_{1,N-1}) \\ ... & ... & ... & ... \\ U(r_{N-1,0}) & U(r_{N-1,1}) & ... & 0 \end{Vmatrix}, (N \times N). \tag{4.4}$$

Next, r_{ij} has to be calculated for all source landmarks P_i and P_j.

$$\begin{aligned}
r_{0,0} &= 0, \\
r_{0,1} &= |\begin{pmatrix} 0 \\ 493.5 \\ 0 \end{pmatrix} - \begin{pmatrix} 0 \\ 493.5 \\ 4194.75 \end{pmatrix}| = \sqrt{-4194.75^2} = 4194.75 = r_{1,0}, \\
&\vdots \\
r_{0,31} &= |\begin{pmatrix} 0 \\ 493.5 \\ 0 \end{pmatrix} - \begin{pmatrix} 6326.68 \\ 493.5 \\ 3814.58 \end{pmatrix}| = \sqrt{-6326.68^2 + (-3814.58^2)} = 7387.69 = r_{31,0}, \\
r_{1,0} &= |\begin{pmatrix} 0 \\ 493.5 \\ 4194.75 \end{pmatrix} - \begin{pmatrix} 0 \\ 493.5 \\ 0 \end{pmatrix}| = \sqrt{4194.75^2} = 4194.75 = r_{0,1}, \\
r_{1,1} &= 0, \\
&\vdots \\
r_{1,31} &= |\begin{pmatrix} 0 \\ 493.5 \\ 4194.75 \end{pmatrix} - \begin{pmatrix} 6326.68 \\ 493.5 \\ 3814.58 \end{pmatrix}| = \sqrt{-6326.68^2 + 380.17^2} = 6338.09 = r_{31,1}, \\
&\vdots \\
r_{31,0} &= |\begin{pmatrix} 6326.68 \\ 493.5 \\ 3814.58 \end{pmatrix} - \begin{pmatrix} 0 \\ 493.5 \\ 0 \end{pmatrix}| = \sqrt{6326.68^2 + 3814.58^2} = 7387.69 = r_{0,31}, \\
r_{31,1} &= |\begin{pmatrix} 6326.68 \\ 493.5 \\ 3814.58 \end{pmatrix} - \begin{pmatrix} 0 \\ 493.5 \\ 4194.75 \end{pmatrix}| = \sqrt{6326.68^2 + (-380.17^2)} = 6338.09 = r_{1,31}, \\
&\vdots \\
r_{31,31} &= 0.
\end{aligned} \tag{4.5}$$

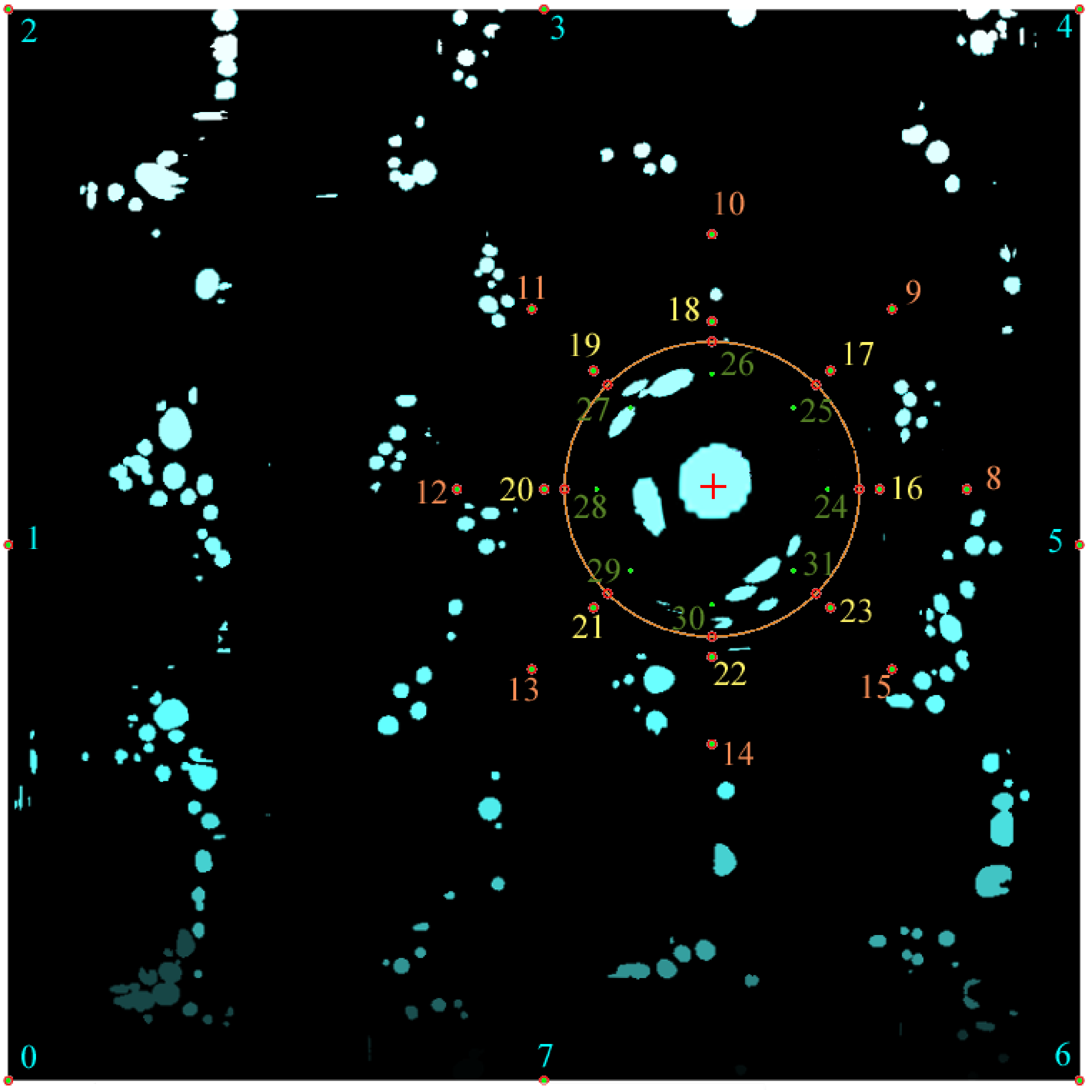


Figure 4.2: Source (red) and target (green) landmarks of the thin plate spline transform. The landmarks are arranged counter-clockwise and the landmarks' IDs and coordinates correspond to the values in Figure 4.1.

It is evident from $r_{i,j} = r_{j,i}$ that the matrix K is symmetric regarding the principal diagonal.

Given all r_{ij} the radial basis function for the TPS_2 can be determined. The RBF used in the source code of `vtkThinPlateSplineTransform.cxx` slightly differs from the algorithm described in Bookstein's paper [Boo89]. Bookstein uses the RBF $U(r) = r^2 * ln(r^2)$, whereas Schroeder et al. use $U(\frac{r}{\sigma}) = (\frac{r}{\sigma})^2 * ln(\frac{r}{\sigma})$. For this application (and per default) $\sigma = 1.0$, which leads to the originally defined RBF $U(r) = r^2 * ln(r)$. σ simply defines the stiffness of the spline.

$$
\begin{aligned}
U(r_{0,0}) &= U(0) = 0,\\
U(r_{0,1}) &= U(4194.75) = 4194.75^2 * ln(4194.75) = 1.4678 \times 10^8\\
&\vdots\\
U(r_{0,31}) &= U(7387.69) = 7387.69^2 * ln(7387.69) = 4.8616 \times 10^8\\
U(r_{1,0}) &= U(4194.75) = 4194.75^2 * ln(4194.75) = 1.4678 \times 10^8\\
U(r_{1,1}) &= U(0) = 0,\\
&\vdots\\
U(r_{1,31}) &= U(6338.09) = 6338.09^2 * ln(6338.09) = 3.5167 \times 10^8\\
&\vdots\\
U(r_{31,0}) &= U(7387.69) = 7387.69^2 * ln(7387.69) = 4.8616 \times 10^8\\
U(r_{31,1}) &= U(6338.09) = 6338.09^2 * ln(6338.09) = 3.5167 \times 10^8\\
&\vdots\\
U(r_{31,31}) &= U(0) = 0.
\end{aligned}
\tag{4.6}
$$

With the solutions of the 32 RBFs, the submatrix K can be built:

$$
K = \begin{Vmatrix}
0 & 1.4678 \times 10^8 & ... & 4.8616 \times 10^8\\
1.4678 \times 10^8 & 0 & ... & 3.5167 \times 10^8\\
... & ... & ... & ...\\
4.8616 \times 10^8 & 3.5167 \times 10^8 & ... & 0
\end{Vmatrix}, (32 \times 32).
\tag{4.7}
$$

Finally, the matrix L is recomposed of the four submatrices K, P, P^T and O:

$$
L = \left\| \begin{array}{cccc|cccc}
0 & 1.4678 \times 10^8 & ... & 4.8616 \times 10^8 & 1 & 0 & 493.5 & 0\\
1.4678 \times 10^8 & 0 & ... & 3.5167 \times 10^8 & 1 & 0 & 493.5 & 4194.75\\
... & ... & ... & ... & ... & ... & ... & ...\\
4.8616 \times 10^8 & 3.5167 \times 10^8 & ... & 0 & 1 & 6326.68 & 493.5 & 3814.58\\
\hline
1 & 1 & ... & 1 & 0 & 0 & ... & 0\\
0 & 0 & ... & 6326.68 & 0 & 0 & ... & 0\\
493.5 & 493.5 & ... & 493.5 & ... & ... & ... & ...\\
0 & 4194.75 & ... & 3814.58 & 0 & 0 & ... & 0
\end{array} \right\|, (36 \times 36).
\tag{4.8}
$$

In the next step, the matrix $X[N + D + 1][D]$ is filled row-wise with the coordinates of the 32 target landmarks. The last four rows are completed with zeros:

$$X = \begin{Vmatrix} 0 & 493.5 & 0 \\ 0 & 493.5 & 4194.75 \\ ... & ... & ... \\ 6412.97 & 493.5 & 4631.29 \\ 6148.49 & 493.5 & 5269.81 \\ 5509.97 & 493.5 & 5534.29 \\ 4871.46 & 493.5 & 5269.81 \\ 4606.97 & 493.5 & 4631.29 \\ 4871.46 & 493.5 & 3992.77 \\ 5509.97 & 493.5 & 3728.29 \\ 6148.49 & 493.5 & 3992.77 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{Vmatrix}, (36 \times 3). \tag{4.9}$$

The first 24 target landmarks (ID 0 - 23 in Figure 4.1) have the same coordinates as the source landmarks (red and green points positions are identical). That is because the neighbourhood of those landmarks is not meant to be affected by the transform. If source and target landmarks are identical, the transform of the image looks like the original slicer image. Only the circular arranged source and target landmarks of the innermost landmark circle (Figure 4.2) vary in coordinates. Hence, this area visibly transforms according to the thin-plate spline transform.

For calculating the weights matrix $W[N+D+1][D]$, the following matrix multiplications have to be performed: $W = V * w^{-1} * V^T * X$.

$V[36][36]$ is the Jacobian matrix of L and includes the eigenvectors that are orthogonal, because L is a real symmetric matrix.

$w[36][36]$ is a diagonal matrix consisting of 36 inverted singular values. Since the matrix L is real symmetric it is also normal, which means, that the singular values simply correspond to the absolute values of the 36 eigenvalues of L. The inverse matrix w^{-1} is needed, therefore, the singular values (s) are inverted. A singular value is only inverted, when its value divided by the maximum singular value (s_{max}) is bigger than 10^{-16} $(s/s_{max} > 10^{-16}, withs_{max} = 6.64 \times 10^9)$, otherwise it is zero.

V^T is the transposed Jacobian matrix V and $X[36][3]$ contains the 32 target landmarks' coordinates (and four rows filled with zeros) as mentioned above.

$$V = \begin{Vmatrix} 0.3914 & -0.5418 & -0.1215 & ... & 0.2341 & -0.5272 \\ 0.2538 & 0.0031 & -0.2469 & ... & -0.1144 & -0.3411 \\ 0.3674 & 0.5301 & -0.1959 & ... & -0.4659 & -0.3319 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0.1146 & 0.0737 & -0.1814 & ... & 0.0862 & 0.1374 \\ 7.58 \times 10^{-10} & 5.56 \times 10^{-11} & 1.64 \times 10^{-10} & ... & 1.29 \times 10^{-13} & -5.29 \times 10^{-10} \\ 3.64 \times 10^{-6} & -2.12 \times 10^{-7} & -3.07 \times 10^{-6} & ... & -1.48 \times 10^{-6} & -5.96 \times 10^{-6} \\ 3.74 \times 10^{-7} & 2.74 \times 10^{-8} & 8.11 \times 10^{-8} & ... & 6.35 \times 10^{-11} & -2.61 \times 10^{-7} \\ 3.33 \times 10^{-6} & -1.72 \times 10^{-6} & 2.21 \times 10^{-6} & ... & 4.46 \times 10^{-6} & -3.46 \times 10^{-6} \end{Vmatrix}, (36 \times 36). \tag{4.10}$$

The transposed matrix V^T is not depicted, as the calculation is straight forward.

$$w^{-1} = \begin{Vmatrix} 1.51 \times 10^{-10} & & & & & & \\ & 1.76 \times 10^{-8} & & & & & \\ & & \ddots & & \text{\Large 0} & & \\ & & & 18.85 & & & \\ & \text{\Large 0} & & & 16980.8 & & \\ & & & & & \ddots & \\ & & & & & & 3.08 \times 10^{-10} \end{Vmatrix}, (36 \times 36). \tag{4.11}$$

Multiplying $V * w^{-}1 * V^T * X$ results in the weights matrix W.

$$W = \begin{Vmatrix} 3.78 \times 10^{-7} & -9.95 \times 10^{-21} & 4.85 \times 10^{-7} \\ -6.09 \times 10^{-6} & 1.09 \times 10^{-20} & -6.07 \times 10^{-7} \\ -2.90 \times 10^{-7} & -2.51 \times 10^{-22} & 3.29 \times 10^{-9} \\ -4.85 \times 10^{-6} & -9.88 \times 10^{-20} & 1.14 \times 10^{-5} \\ 5.12 \times 10^{-6} & 4.32 \times 10^{-20} & 5.39 \times 10^{-6} \\ 2.86 \times 10^{-5} & 5.12 \times 10^{-20} & -3.01 \times 10^{-6} \\ 2.39 \times 10^{-6} & -2.93 \times 10^{-21} & -3.18 \times 10^{-6} \\ -2.82 \times 10^{-6} & -3.57 \times 10^{-20} & -7.44 \times 10^{-6} \\ 0.0022 & 1.01 \times 10^{-18} & -1.22 \times 10^{-6} \\ 0.0015 & 3.03 \times 10^{-18} & 0.0015 \\ -1.20 \times 10^{-6} & 1.08 \times 10^{-18} & 0.0022 \\ \vdots & \vdots & \vdots \\ 0.0004 & 0.0020 & 1.37 \times 10^{-6} \\ 1.0108 & 9.83 \times 10^{-17} & 0.0001 \\ 0.2197 & 0.9999 & 0.0007 \\ 1.45 \times 10^{-5} & 1.58 \times 10^{-16} & 1.0139 \end{Vmatrix} \tag{4.12}$$

Before the thin-plate spline transform can be applied on every point of the 2D slice image, the linear portion of the warp transform is checked. According to Schroeder et al.[7] this check is currently very tenuous. For this check, the linear rotation and scale matrix A is needed. A consists of the last three rows of W:

$$A = \begin{Vmatrix} 1.0108 & 9.83 \times 10^{-17} & 0.0001 \\ 0.2197 & 0.9999 & 0.0007 \\ 1.45 \times 10^{-5} & 1.58 \times 10^{-16} & 1.0139 \end{Vmatrix} \tag{4.13}$$

It is simply checked, if the determinant of A is $< 10^{-16}$. In case it is, the absolute values of the three column vectors are determined, and again, it is checked, if the absolute values are $< 10^{-16}$:

$$\sqrt{a_{0,j}^2 + a_{1,j}^2 + a_{2,j}^2} < 10^{-16}, \quad for \quad j = 0, 1, 2. \tag{4.14}$$

For absolute values $< 10^{-16}$ of either the first and the second, the first and the third, the second and the third column vector or all three column vectors, the rotation and scale matrix A turns to the identity matrix:

$$A = \begin{Vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{Vmatrix} \tag{4.15}$$

If the first, the second or the third absolute value is $< 10^{-16}$, the resulting matrices look like this:

$$A_1 = \begin{Vmatrix} 1 & 0 & 0 \\ 0 & 1 & a_{1,2} \\ 0 & a_{2,1} & 1 \end{Vmatrix}, \ 1^{st} \text{ absolute value} < 10^{-16},$$

$$A_2 = \begin{Vmatrix} 1 & 0 & a_{0,2} \\ 0 & 1 & 0 \\ a_{2,0} & 0 & 1 \end{Vmatrix}, \ 2^{nd} \text{ absolute value} < 10^{-16},$$

$$A_3 = \begin{Vmatrix} 1 & a_{0,1} & 0 \\ a_{1,0} & 1 & 0 \\ 0 & 0 & 1 \end{Vmatrix}, \ 3^{rd} \text{ absolute value} < 10^{-16}. \tag{4.16}$$

$$\tag{4.17}$$

The values of $a_{i,j}$, with $i, j = 0, 1, 2$, are the same as in the initial matrix A in (4.13). At last, the linear translation vector $c[3]$. c consists of the cell values of the 32^{th} row of W from left to right:

$$C = \begin{Vmatrix} 0.0004 \\ 0.002 \\ 1.37 \times 10^{-6} \end{Vmatrix} \tag{4.18}$$

---

[7] `https://github.com/Kitware/VTK/blob/e68a7fe93758c1d46bd42486958b2ebb7819cbb3/Common/Transforms/vtkThinPlateSplineTransform.cxx#L322`, Comment from the source code, line 322. Accessed: 20.08.2017

In this sample iteration of the thin-plate spline transform algorithm, the determinant of A results in $det(A) = 1.02488$, which is $>> 10^{-16}$, so the calculations of the absolute values of the column vectors are not necessary and A stays the same.
The calculations and the construction of all the matrices given in this section are performed whenever the mouse moves over the 2D slice image and new source and target landmarks are selected. The `vtkThinPlateSplineTransform` function `InternalUpdate()` handles these processing steps. `InternalUpdate()` is called by the `Update()` function of `vtkAbstractTransform`.

Apart from `InternalUpdate()`, another function, namely `ForwardTransformPoint()`, is responsible for the actual transformation of every single point between the landmarks. `ForwardTransformPoint()` is called by `InternalTransformPoint()` of the superclass `vtkWarpTransform`. The arguments of both functions consist of a `double` array of point coordinates (`input[3]={x,y,z}`) to be transformed, and a `double` array in which the newly transformed point coordinates are going to be stored (`output[3]={` $\mathtt{x_{new}}$`,`$\mathtt{y_{new}}$`,`$\mathtt{z_{new}}$`}`).
`InternalTransformPoint()` is again called by `TransformPoints()` of the superclass `vtkAbstractTransform`. The input parameters of `TransformPoints()` are two `vtkPoints` arrays, one for the input points and one for the transformed output points.
The procedure of this function is described subsequently:
Essential for the computation of the transformation of a point is the previously built weights matrix W, the linear rotation and scale matrix A and also the translation vector c.
As already explained, the function `ForwardTransformPoint()` needs an input point that is transformed to an output point. In Figure 4.3, the red cross marks the position of the input point taken for the transformation. Its approximate coordinates are x = 507, y = 47 and z = 487].
At this point, is has to be noted, that the described algorithm by Schroeder et al.[8] does not take the point coordinates from the slice image, to be exact, the algorithm does not use the extents[9] of the image, but a physical extent (bounds).
The extent of an image (image space) is the number of voxels in each dimension, ignoring the size of a voxel. The pores.mhd dataset that is used for this algorithm example has an extent of $[0, 799]$ in the x-direction, $[0, 74]$ in the y-direction and $[0, 799]$ in the z-direction:

$$extent_{image} = \begin{bmatrix} x_{min} & x_{max} & y_{min} & y_{max} & z_{min} & z_{max} \end{bmatrix} = \begin{bmatrix} 0 & 799 & 0 & 74 & 0 & 799 \end{bmatrix} \tag{4.19}$$

This means that the pores.mhd dataset has a size of $800 \times 75 \times 800$ voxels, starting with the first voxel indices at 0 in every direction.

---

[8] `https://github.com/Kitware/VTK/blob/master/Common/Transforms/vtkThinPlateSplineTransform.cxx`, Accessed: 20.08.2017

[9] `http://www.vtk.org/Wiki/VTK/Tutorials/Extents`, Accessed: 20.08.2017

However, the physical extent (or bounds), that is needed for the thin-plate spline algorithm additionally takes into account the spacing of the voxels and the origin of the dataset in physical space. The pores.mhd dataset has a voxel spacing of 10.5 in every three dimensions and its origin at $[0, 0, 0]$. Therefore, the physical extent calculates from

$$bounds = extent_{image} * spacing + origin. \tag{4.20}$$

This is the cause, why the values of the landmark coordinates from Figure 4.1 differ from the actual size of the dataset.
Knowing this, it is necessary to convert the input point coordinates $[507, 47, 487]$ that have been selected from the slice image for the `ForwardTransformPoint()` function from image space to physical space.

$$p_{input} = \begin{bmatrix} 507 * 10.5 + 0 \\ 47 * 10.5 + 0 \\ 487 * 10.5 + 0 \end{bmatrix} = \begin{bmatrix} 5323.5 \\ 493.5 \\ 5113.5 \end{bmatrix} \tag{4.21}$$

With the input point p_{input} in physical space coordinates, finally, the point transformation can be performed to get the preliminary output point coordinates $p_{interim}$. First, the non-linear part of the transformation is calculated.
For this reason, the distances (absolute values) of the input point (p_{input}) to every one of the 32 source landmarks (sp_i) are computed. Those distances r_i, with $0 \leq i \leq 31$ serve as argument for the function $U(r_i) = r_i^2 * ln(r_i)$. Actually, r_i would be multiplied by the inverse σ (the value for the stiffness of the spline): $r_i * 1/\sigma$. But since $\sigma = 1$ this term drops out.
$U(r_i)$ is then multiplied by the i^{th} row of the weights matrix $W[32][3]$ and is summed up for all $i = 0, .., 31$. The resulting point is q:

$$\begin{aligned} r_i &= |p_{input} - sp_i|, \text{ for } 0 \leq i \leq 31; \\ q_x &= \sum_{i=0}^{31} U(r_i) * w_{i,1} = -38.181, \\ q_y &= \sum_{i=0}^{31} U(r_i) * w_{i,2} = -7.61 \times 10^{-12}, \\ q_z &= \sum_{i=0}^{31} U(r_i) * w_{i,3} = -402.834. \end{aligned} \tag{4.22}$$

Second and last, the affine transformation is applied to q to get $p_{interim}$. Therefore, the translation vector c is added to p_{input} and p_{input} is then multiplied by the rotation and scale matrix A:

$$\begin{aligned} p_{interim_x} &= q_x + c_{0,0} + (p_{input_x} * a_{0,0}) + (p_{input_y} * a_{1,0}) + (p_{input_z} * a_{2,0}) = 5451.4, \\ p_{interim_y} &= q_y + c_{1,0} + (p_{input_x} * a_{0,1}) + (p_{input_y} * a_{1,1}) + (p_{input_z} * a_{2,1}) = 493.5, \\ p_{interim_z} &= q_z + c_{2,0} + (p_{input_x} * a_{0,2}) + (p_{input_y} * a_{1,2}) + (p_{input_z} * a_{2,2}) = 4782.89. \end{aligned} \tag{4.23}$$

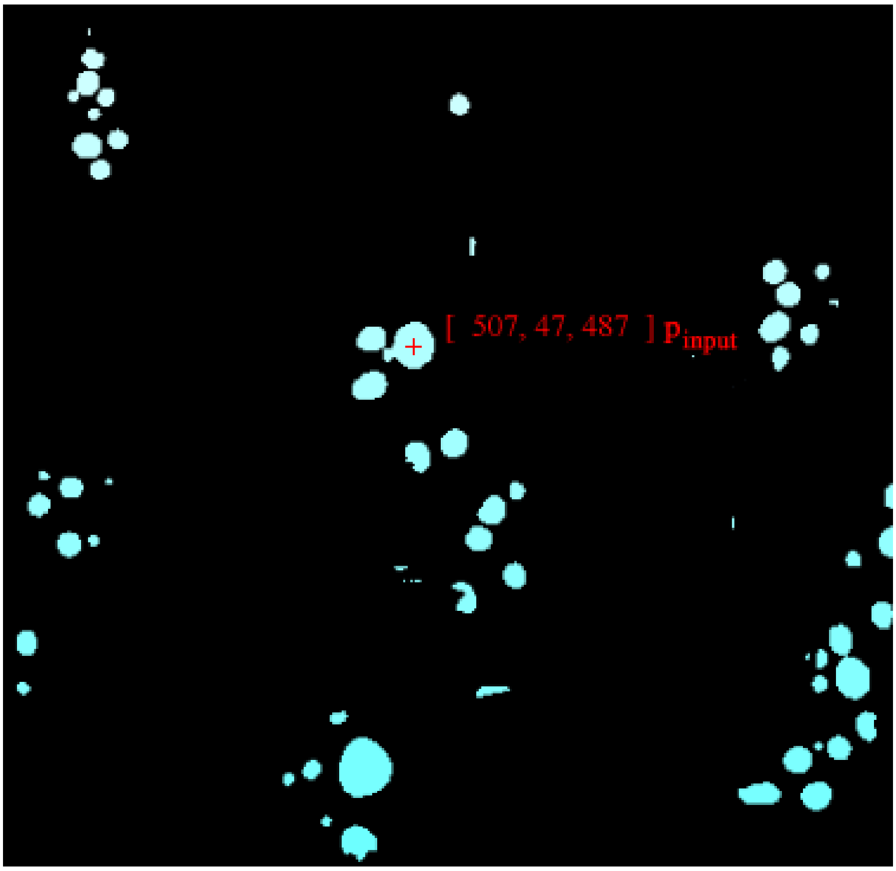


Figure 4.3: The red cross indicates the position of the point p_{input} that is taken for the `ForwardTransformPoint()` function. The point's x-y-z-coordinates in image space are approximately [507, 47, 487]. In physical space they are [5323.5, 493.5, 5113.5].

Because of the many decimal places and the partially very small and big values of the intermediate results, rounding has been necessary to provide somewhat clear results throughout this algorithm example. Those clear results, however, implicate quite a loss in precision that cannot be ignored, which is why the exact intermediate results and matrix values that have been computed with C++ are used for the preliminary transformed output point $p_{interim}$.
To have a better comparison of the input and output point coordinates, $p_{interim}$ is converted back to the image extents (pixel indices):

$$index = bounds \div spacing - origin; \tag{4.24}$$

$$p_{interim} = \begin{bmatrix} 5451.4 \\ 493.5 \\ 4782.89 \end{bmatrix} \div 10.5 - 0 \approx \begin{bmatrix} 519 \\ 47 \\ 456 \end{bmatrix}. \tag{4.25}$$

Altogether, with every mouse move over one of the three slice images, new source

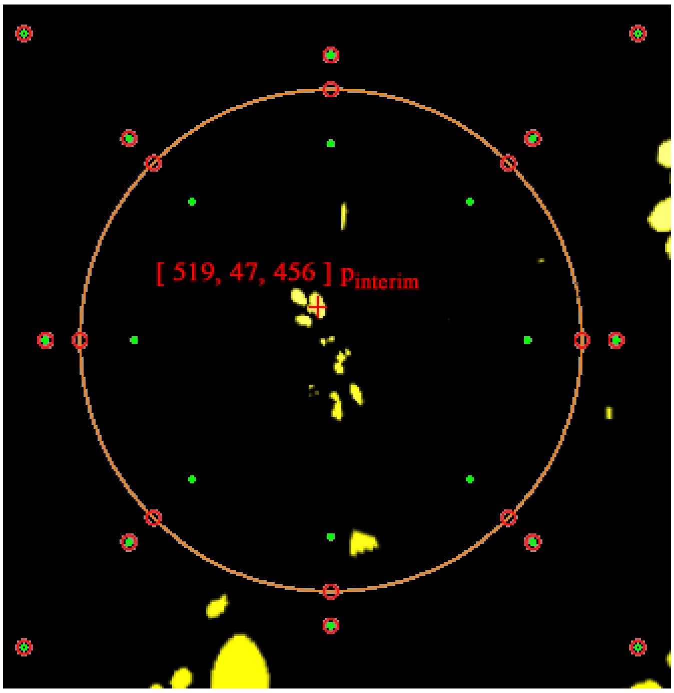


Figure 4.4: The preliminary output point coordinates of $p_{interim}$ correspond to the point in this inversely transformed slice image. The inverse transform of the lens-like magnification distortion results in a downsized distortion of the area inside the lens.

and target landmarks are assigned to the thin-plate spline transform, that processes them and calculates a new weights matrix W and resulting rotation and scale matrix A, and transformation vector c. With those two matrices and the vector, the points of the slice image that are nearby specific source landmarks are transformed towards the corresponding target landmarks.
Obviously, the coordinates of the preliminary output point $p_{interim}$ [519, 47, 456] that have been retrieved throughout this algorithm example do not match the transformed coordinates [494, 47 521] of the actual output point p_{output} in Figure 4.5. However, the coordinates of $p_{interim}$ match with the inverse thin-plate spline transformation that can be seen in Figure 4.4. Information to this matter can be found in the class reference of `vtkImageReslice`[10]. This class is responsible for the three slices along different axes (XY, YZ, XZ) of the given dataset.

The thin-plate spline transform has to be forwarded to `vtkImageReslice` by means of the function `SetResliceTransform()` and is then applied to a resampling grid. This grid has been defined by the reslice axes (here XZ) and the output origin [0, 0, 0], spacing [10.5, 10.5, 10.5] and extent [0 799, 0, 74, 0, 799], and has therefore the size of the slice. Here, ‚output‘ designates the 2D slice that has been cut out of the original 3D

---

[10] `http://www.vtk.org/doc/nightly/html/classvtkImageReslice.html`, Accessed: 20.08.2017

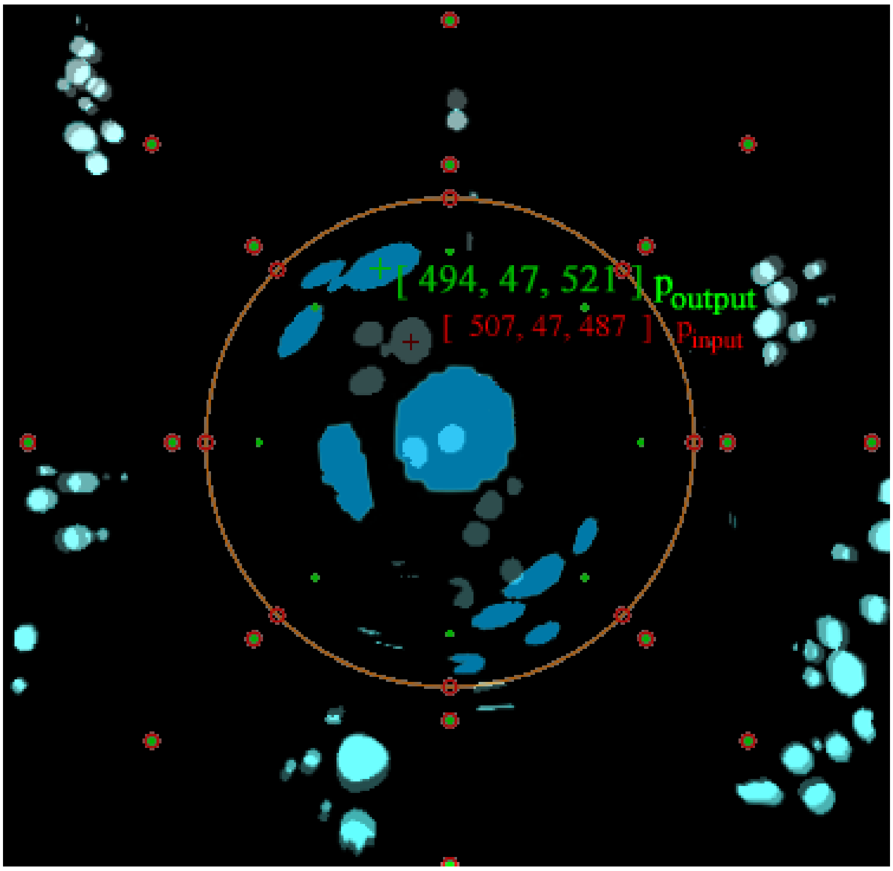


Figure 4.5: The transformed and the original slice from Figure 4.3 have been laid on top of each other. The green cross, p_{output} with the coordinates [494, 47, 521], marks the transformed position of the point p_{input} from Figure 4.3. Despite the stabilizing fix landmarks around the centre of distortion, it is visible, that also the pixels of the rest of the slice image are slightly transformed.

volume data (input).
Now, it is important to know, that applying the transform to this resampling grid is equal to applying the transform's inverse to the input volume data, which is done automatically in VTK. That is because warp transforms are typically inverted. The inverting is done by Newton's method[11], an iterative technique with which the inverse coordinates of a point are approximated.
With this new insight, the preliminary output point coordinates [519, 47, 456] of $p_{interim}$ can be explained. Actually, $p_{interim}$ is calculated for the resampling grid, whereas the transformation that is visible for the user on the resliced image is the inverted one. To get the final output point p_{output}, that corresponds to the inverted transformation which

---

[11] `https://www.vtk.org/doc/nightly/html/classvtkWarpTransform.html#afc508fcd1567aeefd48454187fb5154b`, Accessed: 20.08.2017

is visible for the user, the preliminary point coordinates [519, 47, 456] of $p_{interim}$ have to be inverted too. The algorithm for inverting a point can be looked up in the source code of the `vtkWarpTransform`[12] class, and will not be explained any further, since it shall only explain the correctness of the calculated preliminary output point by the thin-plate spline algorithm (and is not part of the thin-plate spline algorithm itself).

$$\begin{bmatrix} 519 \\ 47 \\ 456 \end{bmatrix} \xrightarrow{\textit{invert point with Newton's method}} \begin{bmatrix} 494 \\ 47 \\ 521 \end{bmatrix} = p_{output} \tag{4.26}$$

4.3 Functionality of the Fish-Eye Lens

In this section, the TPS_2 is integrated into the remaining fish-eye lens implementation which is responsible for the user interaction with the lens. The functionality of the lens is illustrated with the aid of three pseudocode examples.

In Algorithm 4.1, the activation and deactivation of the fish-eye lens is shown. Whenever the fish-eye lens is not activated, it can be enabled by pressing the `O-key` on the keyboard. This only works, when the mouse has been clicked at least once inside the respective slice view.

Then, all the necessary variables for the transform and the appearance of the fish-eye lens are initialized. The class names of the variables and their function are listed as follows:

- `vtkRenderer`[13]: the renderer of the current slice view is needed for drawing the fish-eye lens onto the slice image.
- `vtkThinPlateSplineTransform`: the thin-plate spline transform that has been described in the previous section.
- `vtkPoints`: 32 source and target landmarks are initialized.
- `vtkRegularPolygonSource`[14]: creates a regular polyline for the lens shape.
- `vtkPolyDataMapper`[15]: maps the polygonal lens shape to a graphic primitive so it can be used by the actor.
- `vtkActor`[16]: is added to the renderer and represents the mapped lens shape in the rendered scene, so the lens is visible for the user.

---

[12] `https://github.com/Kitware/VTK/blob/master/Common/Transforms/vtkWarpTransform.cxx`, Accessed: 20.08.2017

[13] `http://www.vtk.org/doc/nightly/html/classvtkRenderer.html`, Accessed: 20.08.2017

[14] `http://www.vtk.org/doc/nightly/html/classvtkRegularPolygonSource.html`, Accessed: 20.08.2017

[15] `http://www.vtk.org/doc/nightly/html/classvtkPolyDataMapper.html`, Accessed: 20.08.2017

[16] `http://www.vtk.org/doc/nightly/html/classvtkActor.html`, Accessed: 20.08.2017

Algorithm 4.1: Activation and Deactivation of the fish-eye lens

```
fish-eye lens active ← $false$
renderer ← assign renderer of selected 2D slice view
Require: key on keyboard was pressed
if pressed key is O then
    if fish-eye lens not active then
        fish-eye lens active ← $true$
        pass the mouse cursor position to the renderer
        initialize all necessary variables for the fish-eye transform
        update the parameters for the fish-eye transform (Algorithm 4.2)
    else
        fish-eye lens active ← $false$
        remove lens from renderer of the 2D slicer view
        transform data of the slicer back to normal
    end if
end if
```

After the initialisation, the fish-eye lens transformation is calculated and rendered onto the slice image, as depicted in Algorithm 4.2.

The cursor position is needed to get the centre of the transformation. The slicer data (e.g. pores.mhd) is required to obtain the bounds (physical space) and the spacing of the data.

The bounds are needed to retrieve the coordinates of the 32 source and target landmarks in physical space and the spacing is needed to convert the slice number (which is in pixel space) into physical space as well. The XY-slicer's slice number corresponds to the Z-coordinate of the slice image. For the pores.mhd dataset, the XY-slice view has 799 slices which corresponds to the size of the extent in the Z-dimension.

Having the size of the slice image in physical space, the landmarks' positions can be calculated. Around the fish-eye lens, there are two additional landmark circles (only visible for debugging reasons) that aim to stabilize the transformation and try to keep the distortion preferably inside the lens.

One stabilizing circle lies 15 pixels outside and around the lens, the other circle 80 pixels. These two distances have turned out to be very efficient when it comes to keeping the distortion restricted. The target landmarks of the lens circle lie around a circle (with custom radius) inside the lens circle. Those landmarks are decisive for the appearance of the transformation.

When all the landmarks have been calculated, they are passed to the thin-plate spline transform, and the radius of the lens is set accordingly. Then, the slice view is updated and the fish-eye lens becomes visible.

Algorithm 4.2: Update fish-eye transform

```
Require: cursor position, slicer data, outer radius and inner radius
get bounds and spacing of slicer data
get slicer mode // mode is either XY, YZ or XZ
get the number of current slice
// calculating all 32 source and target landmarks for the thin plate spline transform
taking slicer mode and slice number into account
set 8 source and target landmarks around the border of the 2D slice image
for landmark 8 to 16 do
    calculate a point every 45° around a circle 15 pixels outside the lens radius
    set source and target landmark to that point
end for
for landmark 16 to 24 do
    calculate a point every 45° around a circle 80 pixels outside the lens radius
    set source and target landmark to that point
end for
for landmark 24 to 32 do
    calculate a point every 45°, around the lens radius and around the inner lens
    radius
    set source landmark to the point on the lens radius
    set target landmark to the point on the inner lens radius
end for
(... code for drawing all the calculated source and target landmarks onto the data ...)

set the centre of the lens to the position of the mouse cursor
set the radius of the lens to outer radius
set the source and target landmarks for the transform
apply the transform to the slicer data
(update the slicer)
```

While the lens is active, the radius of the lens as well as the degree of distortion inside the lens can be customized via keyboard input. Algorithm 4.3 is responsible for this functionality.

In this implementation, a smooth adjustment of the radius and the distortion is not possible due to the choice of the keyboard inputs. The radius of the fish-eye lens can be changed by pressing the `plus-key` or the `minus-key`, the distortion can be changed by pressing `STRG + plus-key` or `STRG + minus-key`. With these given interaction options, smooth adjustments would be troublesome. Because of this, it has been decided to change the radius and the distortion according to predefined intervals. The default lens radius is 80 pixels but can be augmented to a maximum of 220 pixels and reduced to a minimum of 2 pixels (in case of extremely small datasets), by steps of 10 pixels each. If the radius is < 10 the radius changes by steps of 1 pixel.

The landmark circle inside the lens, that influences the degree of distortion, has its default radius at 70 pixels. It can shrink down to a minimum of $\mathit{lens\ radius} - 22$ pixels for a $\mathit{radius} > 10$. For a $\mathit{radius} < 10$ the minimum is 1 pixel. A distortion lens radius of $< \mathit{lens\ radius} - 22$ with a $\mathit{radius} > 10$ would not make sense, since the distortion would become too strong to be visually useful.

The possibility to reduce the lens radius to a minimum of 2 pixels was only offered for

the sake of completeness so that also small datasets (e.g. $200 \times 50 \times 100$) can be explored, although most of the datasets are bigger than $800 \times 75 \times 800$.
The maximum radius can reach the size of the lens radius, which means that no distortion occurs. This radius is changed by steps of 2 pixels, for a $radius < 10$ by steps of 1 pixel. It has to be noted, that a small distortion radius leads to a very strong distortion, whereas a big distortion radius leads to very little up to no distortion. In Algorithm 4.3 the outer radius is the lens radius and the inner radius is the radius for the distortion.

Algorithm 4.3: Adapting lens radius and degree of distortion

```
Require: key on keyboard pressed and fish-eye lens active
pass mouse cursor position to the renderer
if keyboard modifier STRG pressed then
  if minus-key pressed then
    if inner radius + 2.0 $\leq$ outer radius then
      inner radius + 2.0
      update parameters for fish-eye transform with new inner radius // unmagnifies the distortion
    end if
  end if
  if plus-key pressed then
    if inner radius - 2.0 $\geq$ minimal inner radius then
      inner radius - 2.0
      update parameters for fish-eye transform with new inner radius // magnifies the distortion
    end if
  end if
else if no keyboard modifier pressed then
  if plus-key pressed then
    if outer radius + 10.0 $\ngtr$ default outer radius + 140.0 then
      outer radius + 10.0
      inner radius + 10.0
      minimal inner radius + 8.0
      update parameters for fish-eye transform with new inner and outer radii // magnifies lens
    end if
  end if
  if minus-key pressed then
    if outer radius - 10.0 $\nless$ default outer radius - 20.0 then
      outer radius - 10.0
      inner radius - 10.0
      minimal inner radius - 8.0
      if inner radius < minimal inner radius then
        inner radius = minimal inner radius
      end if
      update parameters for fish-eye transform with new inner and outer radii // unmagnifies lens
    end if
  end if
end if
```

CHAPTER 5

Evaluation and Results

In this chapter, applications of the fish-eye lens on different 3D volume datasets are presented and the results are evaluated.
A colour transfer function has been applied to the 3D volume datasets to make them easier to explore. For choosing optimal colour combinations for the transfer function the online tool *Colorbrewer 2.0*[1] by Cynthia A. Brewer is used. *Colorbrewer 2.0* takes colour blindness into account and helps to select appropriate colour schemes for specific purposes.
In the following, the purpose of the fish-eye lens is demonstrated with the help of three different 3D volume datasets.

5.1 Rock Crystal Dataset

The layout of the open_iA main window is shown in Figure 5.1. It depicts the 3D absorption contrast data of a rock crystal specimen. The fish-eye lenses have been enabled in every slice view. The rock crystal specimen was scanned with the *Bruker SkyScan1294*[2] Talbot-Lau grating interferometry X-ray computed tomography device[3], which allows simultaneous extraction of differential phase contrast as well as absorption contrast and dark-field images. Dark cracks and pores inside the rock crystal can be perceived in the different slice views. The rock crystal dataset has a size of $570 \times 600 \times 431$ voxels and is scanned with a resolution of 22.8 microns.
In Figure 5.2 a 2D XY-slice image of the rock crystal is shown. The red circle on the left marks a very small pore, which is barely visible. This pore is strongly magnified with

---

[1] `http://colorbrewer2.org/`, Accessed: 18.08.2017

[2] `https://www.bruker.com/products/microtomography/in-vivo-micro-ct/skyscan-1294/overview.html`, Accessed: 19.08.2017

[3] `https://www.bruker.com/products/microtomography/in-vivo-micro-ct/skyscan-1294/technical-details.html`, Accessed: 20.08.2017

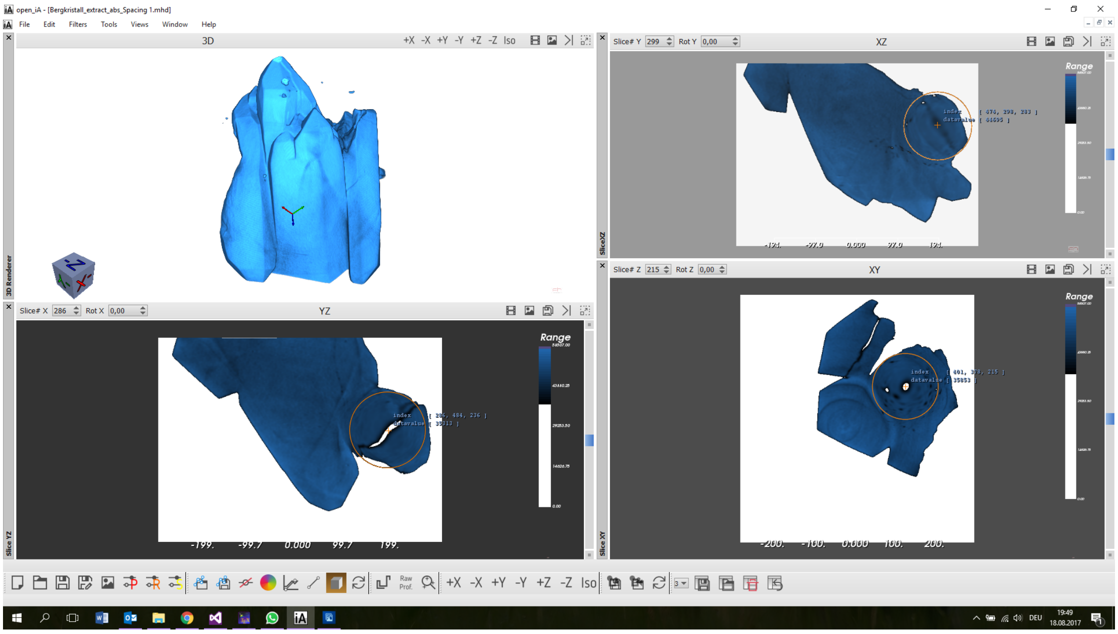


Figure 5.1: The main window of open_iA, showing the 3D rendered volume data of a rock crystal (top left). The XZ-slice view (top right), the YZ-slice view (bottom left) and the XY-slice view (bottom right), each having the fish-eye lens activated.

the fish-eye lens in Figure 5.3.
By using the fish-eye lens, the shape of the pore is clearly visible. Since the pore has been chosen to be in the centre of the lens, it is hardly distorted but magnified, (almost) keeping its original shape, compared to the other contours of the crystal inside the lens which get more and more distorted lengthwise and are squeezed together as they reach the border of the lens.
The fish-eye lens works the same way in the other two slicers. Because the lenses are already shown in the layout image of open_iA (see Figure 5.1), its demonstration on the other two slice images is left out.

The fish-eye lens performs its transformation on the voxels itself, meaning that after a transformation the voxel values are not correct anymore. This leads to the fact, that calculations (e.g. distance measurements) cannot be made on the distorted dataset without risking inaccurate results.
Furthermore, to achieve a preferably good magnification of a small detail, it is necessary to always put this detail in the centre of the lens (at the position of the mouse cursor). This guarantees that the detail's shape (despite the distortion) stays nearly the same, apart from the magnification. Whereas details away from the centre distort strongly and their original shape is not maintained.

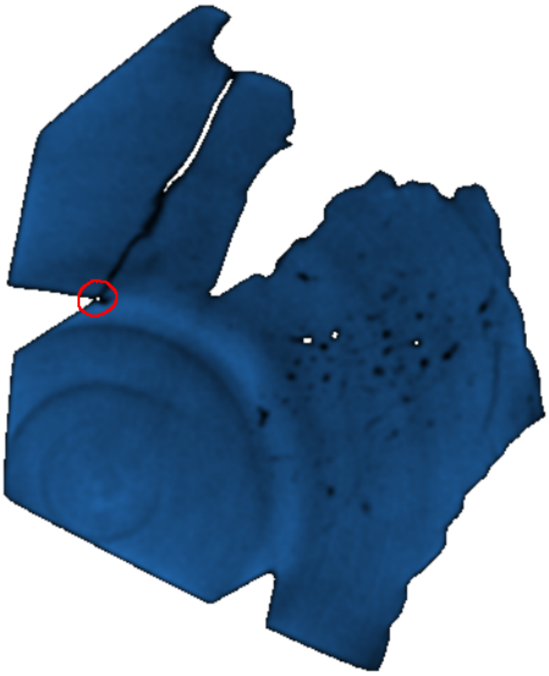


Figure 5.2: This is a close-up view of the XY-slice 215 of the rock crystal dataset. The red circle marks a very small pore at the border of the crystal which is magnified with the fish-eye lens in Figure 5.3.

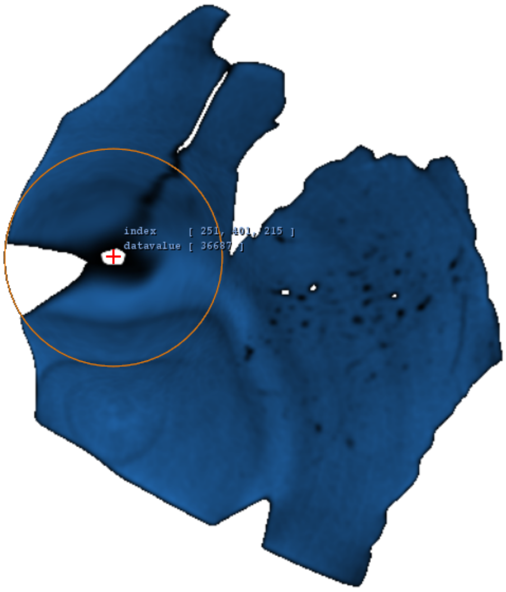


Figure 5.3: The fish-eye lens magnifies the small pore for data exploration.

5.2 Pore Dataset

The 3D pore volume dataset is already well-known, as it has been used in the previous chapters as an example dataset.

The pore volume that can be seen in Figure 5.4 has been extracted from a carbon fibre-reinforced polymer (CFRP) laminate specimen, using the proposed segmentation pipeline by Reh et al. [RPK+12]. The specimen contains pores which are introduced during the manufacturing process. The knowledge and specification of the pores, e.g. shape, number, position and distribution is crucial for identifying the final material properties (e.g. stiffness and strength) [WAG+16].

The CFRP specimen was scanned with a *GE Phoenix Nanotom M*[4], an X-ray computed tomography (XCT) system. The pores dataset has a size of 800 × 75 × 800 voxels and is scanned with a resolution of 22.8 microns.

Figure 5.5 depicts the XZ-slice 47. Again, a red circle marks very small pores that are hard to perceive. However, in Figure 5.6, due to the fish-eye lens, the pores are emphasized distinctively.

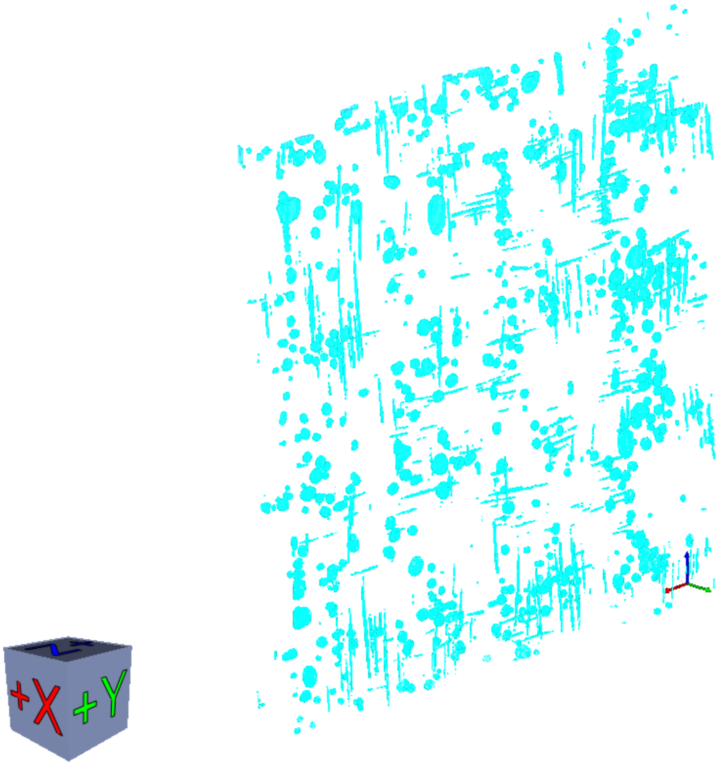


Figure 5.4: The 3D volume rendering of the segmented pores.

---

[4] `https://www.gemeasurement.com/inspection-ndt/radiography-and-computed-tomography/phoenix-nanotom-m-industrial-ct-3d-metrology-system`, Accessed: 20.08.2017

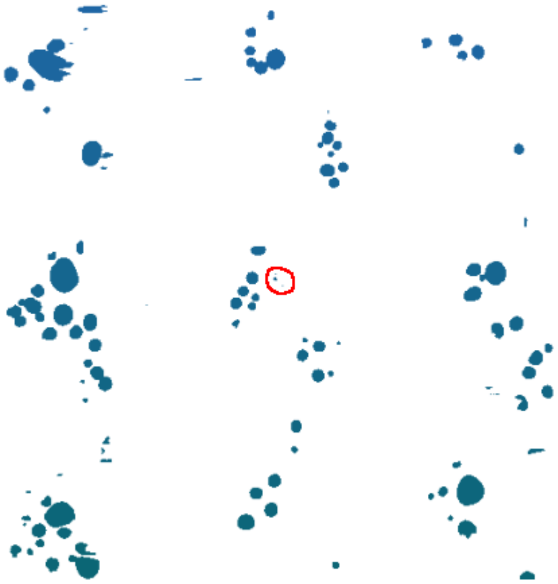


Figure 5.5: XZ-slice 47 of the pore dataset. The three little pores marked with the red circle are hardly visible.

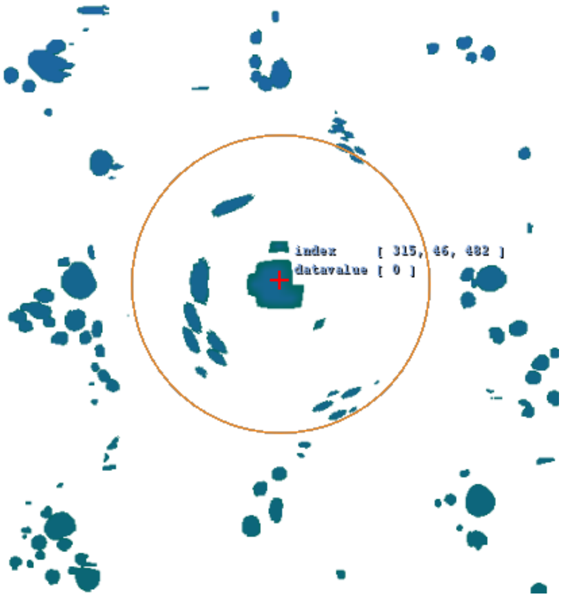


Figure 5.6: With the magnification of the fish-eye lens the three pores are clearly distinguishable.

5.3 Fibre Dataset

As a last dataset example, a glass fibre-reinforced polymer (GFRP) specimen with a polypropylene matrix (PP-SGF30) is presented. PP-SGF30 stands for polypropylene with a glass fibre weight content of 30%. The glass fibres are segmented, which means that every voxel belonging to a single fibre is mapped to a specific label. This information can be useful for further analysis. Depending on the distribution, the length and the alignment of individual glass fibres, characteristics such as stiffness, strength, ductility or durability of the reinforced polymers can be obtained [WAL$^+$14].
This specimen was also scanned with the *GE Phoenix Nanotom M* with a resolution of 2 microns and has a size of 400 × 400 × 400 voxels.
Figure 5.7 shows the 3D GFRP dataset with the segmented fibres. Figures 5.8 and 5.9 demonstrate the fish-eye lens on a cutout of the XY-slice, Figures 5.10 and 5.11 on a cutout of the YZ-slice and Figures 5.12 and 5.13 on a cutout of the XZ-slice.

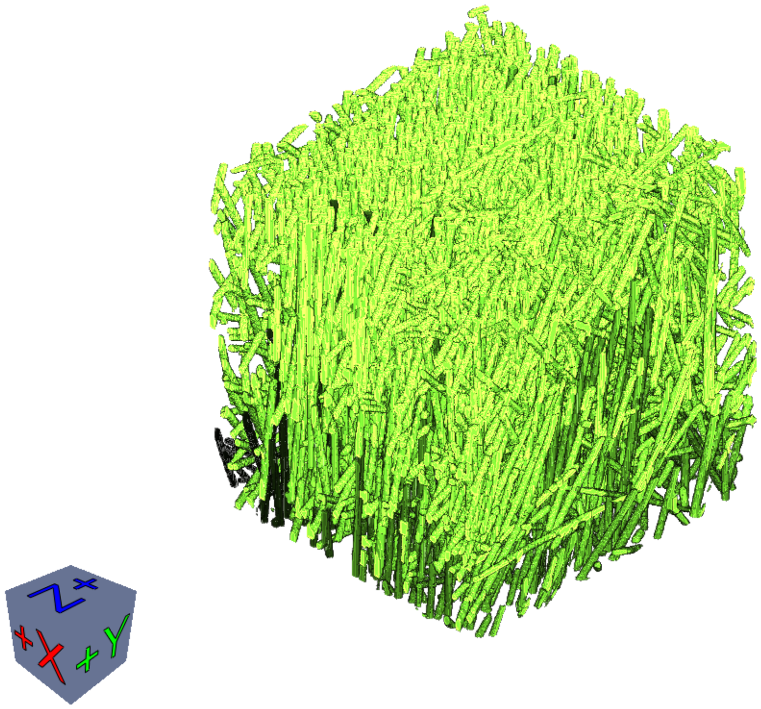


Figure 5.7: 3D rendering of the segmented glass fibres of the XCT scanned GFRP specimen.

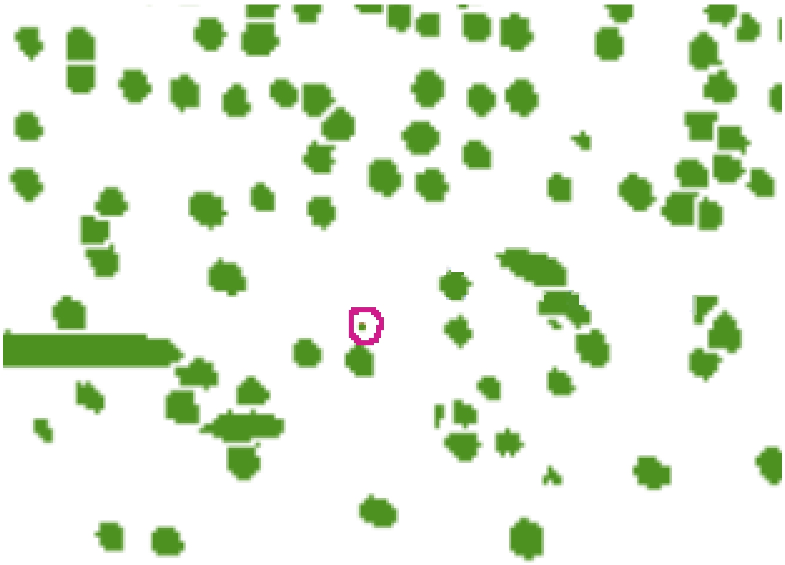


Figure 5.8: Cutout of the XY-slice 157 of the GFRP dataset before enabling the fish-eye lens. This slice mostly consists of fibres that are positioned orthogonally to the image plane. The glass fibre detail to be magnified is marked.

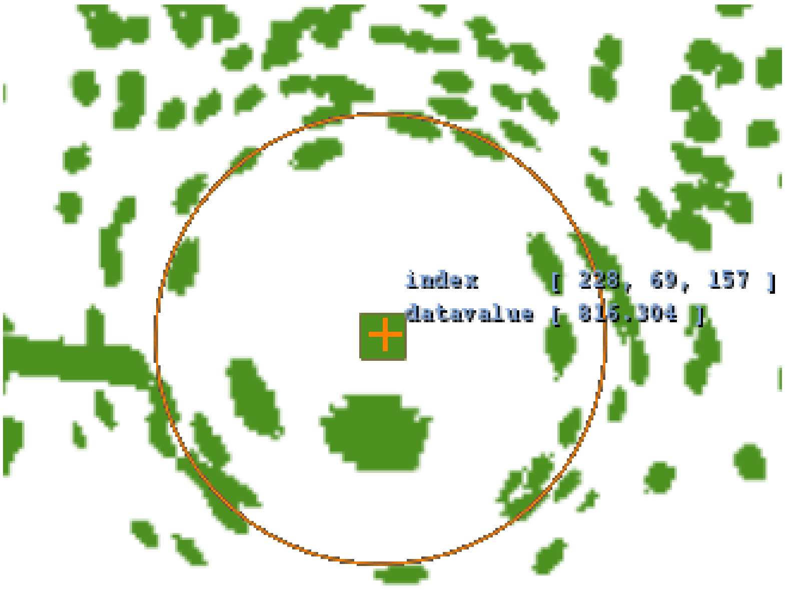


Figure 5.9: Cutout of the XY-slice 157 of the GFRP dataset with fish-eye lens magnification. The small glass fibre detail is now clearly noticeable.

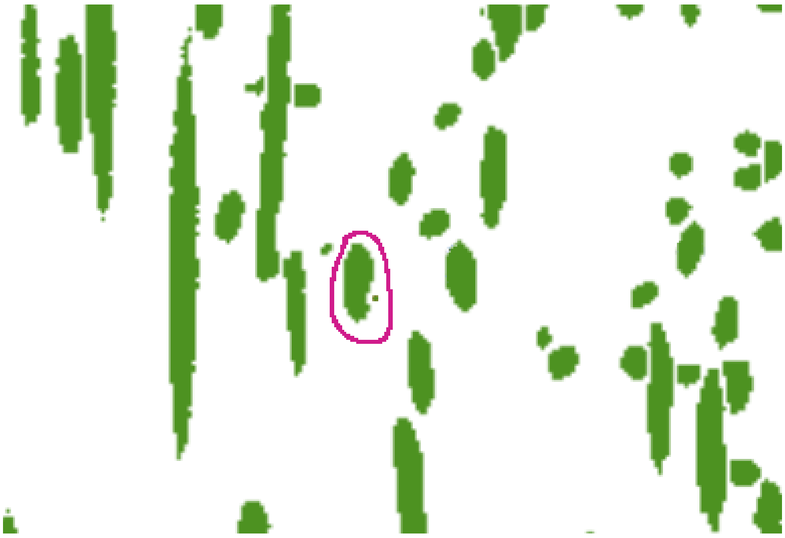


Figure 5.10: Cutout of the YZ-slice 199 of the GFRP dataset before enabling the fish-eye lens. The tiny fibre part to the right inside the pink marked circle is going to be magnified.

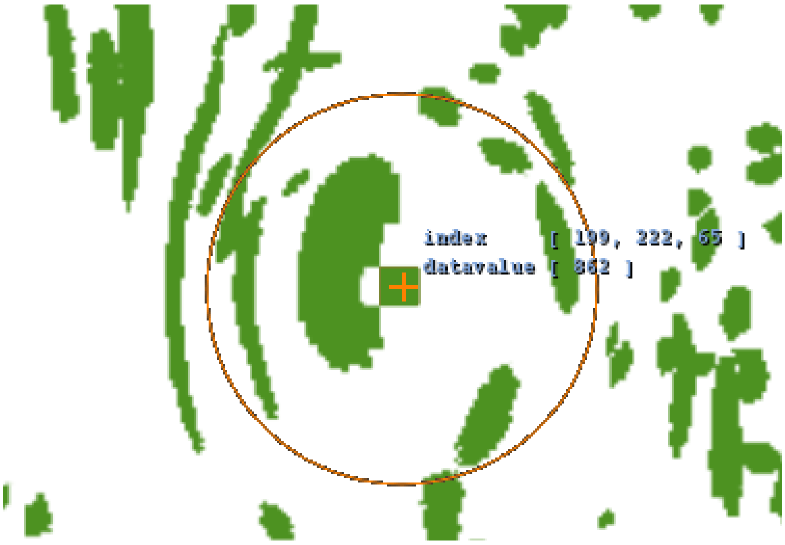


Figure 5.11: Cutout of the YZ-slice 199 of the GFRP dataset with fish-eye lens magnification.

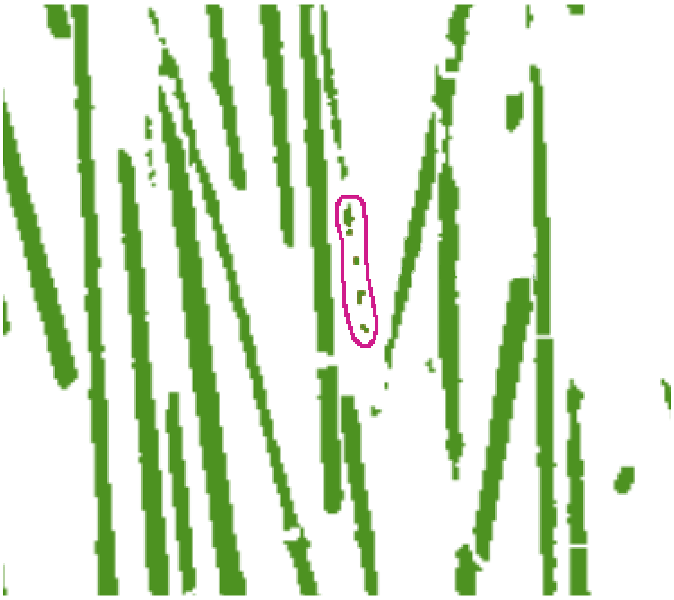
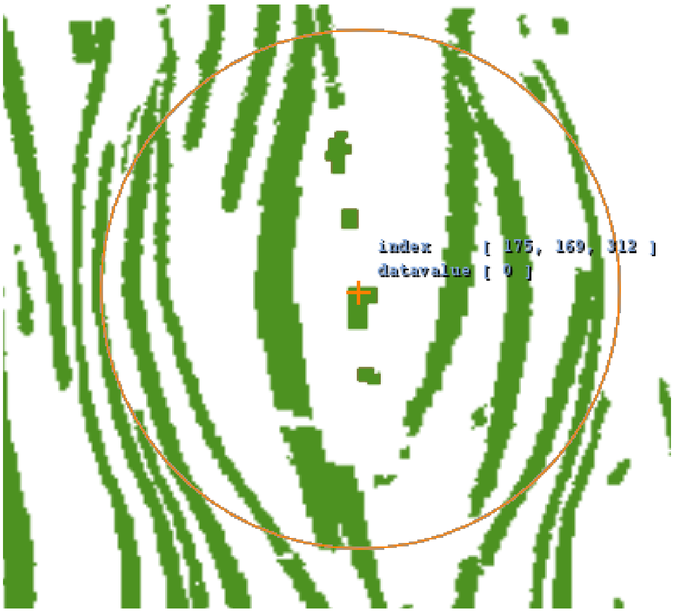


Figure 5.12: Cutout of the XZ-slice 170 of the GFRP dataset before enabling the fish-eye lens. The focus is on the pink circled fibre parts.



index [175, 169, 312]
datavalue [0]

Figure 5.13: Cutout of the XZ-slice 170 of the GFRP dataset with fish-eye lens magnification. The formerly pink circled fibre parts are better visible now.

CHAPTER 6

Conclusion

In this thesis the implementation of a 2D fish-eye lens and its integration into the open source software open_iA, a visual analysis and processing tool for X-ray computed tomography datasets, has been presented. With the help of the thin-plate spline transform, it was possible to achieve the characteristic transformation of a fish-eye lens. An exemplary calculation of the thin-plate spline transform was performed on the coordinates of a point from a 2D slice image. The fish-eye lens was then applied on three different datasets to present and discuss its usability.

The major advantage of the fish-eye lens is the magnification of a region of interest without losing the context to the surrounding area. This is due to the special transformation of the lens that provides a smooth transition between the inside and the outside of the lens. As a result, the magnification has to be the strongest in the centre of the lens. Towards the borders of the lens the magnifications diminishes and the details around the border of the lens get squeezed lengthwise. This deformation is an unpleasant side effect which is consciously accepted in order to ensure the context-preserving feature of the lens. However, the fish-eye lens enables to reveal very small features which would be barely visible otherwise. As the thin-plate spline transform is applied to the voxel positions, calculations based on the voxel grid are critical while the lens is activated, since the voxel positions do not correspond to the original positions anymore. Therefore, the fish-eye lens was mainly designed for visual exploration. Computations such as the evaluation of the diameter of a pore or the length of a fibre would not work, since the coordinates of the voxels are transformed.

Currently, the lens size and the degree of distortion can be magnified or diminished by means of predefined intervals. In the future, it might be possible to add a dialog, where the radius and the distortion can be entered manually.

Furthermore, the lens could be extended by offering different visualisations of the same dataset inside the lens.

List of Figures

2.1 The *ChronoLens* is transforming the underlying content for better exploration. 3
2.2 Several *Detail Lenses* are arranged around the centre of the map and magnify important POIs along the route. 4
2.3 DVR of an engine without (a) and with (b) magnification by the embedded *Magic Volume Lens*. 4
2.4 The *FLOWLENS* visualizes the blood flow inside an aneurysm with the help of streamlines. The flow pressure is depicted by the green saturation-coded contour lines. 5
2.5 The strongly saturated parallel coordinate system makes it difficult to perceive the lines on the left and right side (a). The *Sampling Lens* offers a clearer view of the dense areas to perceive any hidden trends (b). 5
2.6 The car parts inside the lens are listed next to it. The heatmap chart underneath every list entry describes the frequency (red = very frequent, beige = less frequent) in which the names of the car parts appeared in certain documents in an interval of 4 months. 6
2.7 Graph data without (a) and with the application of three *EdgeLenses* (b). The blue points mark the centre of the lenses. The green and pink coloured edges are ignored by the *EdgeLenses* and are not curved around the lenses. 7
2.8 The focus lies on the red-rimmed node (a). With a local edge lens, edges that do not belong to the focused node are removed (b). Then, the *Layout Lens* gathers all nodes adjacent to the focused node (c). Accumulated nodes in the centre are spread by a fish-eye lens (d). 7
2.9 Cutout of the subway map of Washington D.C. The applied fish-eye lens magnifies the focused area in the middle. On the border of the focused area, the map is strongly distorted by the lens to permit a smooth transition between the focused area and the context. 8

3.1 XZ-slice view of the pore dataset (pores.mhd) before (a) and after (b) activating the fish-eye lens (a 'cyan' colour transfer function has been applied to the data for better visibility). The orange circle represents the fish-eye lens. 11
3.2 XZ-slice view of the pore dataset (pores.mhd) with a fish-eye lens radius of 150.0 pixels and very little distortion (a) and strong distortion (b). The orange circle represents the fish-eye lens. 13

4.1 IDs' and coordinates in physical space (bounds) and image space (extent) of the 32 source and target landmarks depicted in Figure 4.2. The blue, orange and yellow landmarks have identical source and target positions and serve as stabilisation for the transform. The green source and target landmarks are responsible for the actual lens distortion. Because of the XZ-slice, the Y values are constant. The values between the brackets (x, y, z), represent the same landmark coordinates but in image space. 19
4.2 Source (red) and target (green) landmarks of the thin plate spline transform. The landmarks are arranged counter-clockwise and the landmarks' IDs and coordinates correspond to the values in Figure 4.1. 21
4.3 The red cross indicates the position of the point p_{input} that is taken for the `ForwardTransformPoint()` function. The point's x-y-z-coordinates in image space are approximately [507, 47, 487]. In physical space they are [5323.5, 493.5, 5113.5]. 28
4.4 The preliminary output point coordinates of $p_{interim}$ correspond to the point in this inversely transformed slice image. The inverse transform of the lens-like magnification distortion results in a downsized distortion of the area inside the lens. 29
4.5 The transformed and the original slice from Figure 4.3 have been laid on top of each other. The green cross, p_{output} with the coordinates [494, 47, 521], marks the transformed position of the point p_{input} from Figure 4.3. Despite the stabilizing fix landmarks around the centre of distortion, it is visible, that also the pixels of the rest of the slice image are slightly transformed. . . . 30

5.1 The main window of open_iA, showing the 3D rendered volume data of a rock crystal (top left). The XZ-slice view (top right), the YZ-slice view (bottom left) and the XY-slice view (bottom right), each having the fish-eye lens activated. 36
5.2 This is a close-up view of the XY-slice 215 of the rock crystal dataset. The red circle marks a very small pore at the border of the crystal which is magnified with the fish-eye lens in Figure 5.3. 37
5.3 The fish-eye lens magnifies the small pore for data exploration. 37
5.4 The 3D volume rendering of the segmented pores. 38
5.5 XZ-slice 47 of the pore dataset. The three little pores marked with the red circle are hardly visible. 39
5.6 With the magnification of the fish-eye lens the three pores are clearly distinguishable. 39
5.7 3D rendering of the segmented glass fibres of the XCT scanned GFRP specimen. 40
5.8 Cutout of the XY-slice 157 of the GFRP dataset before enabling the fish-eye lens. This slice mostly consists of fibres that are positioned orthogonally to the image plane. The glass fibre detail to be magnified is marked. 41

5.9 Cutout of the XY-slice 157 of the GFRP dataset with fish-eye lens magnification. The small glass fibre detail is now clearly noticeable. 41
5.10 Cutout of the YZ-slice 199 of the GFRP dataset before enabling the fish-eye lens. The tiny fibre part to the right inside the pink marked circle is going to be magnified. 42
5.11 Cutout of the YZ-slice 199 of the GFRP dataset with fish-eye lens magnification. 42
5.12 Cutout of the XZ-slice 170 of the GFRP dataset before enabling the fish-eye lens. The focus is on the pink circled fibre parts. 43
5.13 Cutout of the XZ-slice 170 of the GFRP dataset with fish-eye lens magnification. The formerly pink circled fibre parts are better visible now. 43

List of Algorithms

4.1 Activation and Deactivation of the fish-eye lens 32
4.2 Update fish-eye transform 33
4.3 Adapting lens radius and degree of distortion 34

Bibliography

[BMe^{+}14] R. Borgo, R. Maciejewski, I. Viola (editors, C. Tominski, S. Gladisch, U. Kister, R. Dachselt, and H. Schumann. A survey on interactive lenses in visualization. EuroVis State-of-the-Art Reports, Eurographics Association, 2014.

[Boo89] F. L. Bookstein. Principal warps: Thin-plate splines and the decomposition of deformations. *IEEE Trans. Pattern Anal. Mach. Intell.*, 11(6):567–585, June 1989.

[Boo97] Fred L. Bookstein. Shape and the information in medical images. *Comput. Vis. Image Underst.*, 66(2):97–118, May 1997.

[CC13] M. W. Chang and C. Collins. Exploring entities in text with descriptive non-photorealistic rendering. In *2013 IEEE Pacific Visualization Symposium (PacificVis)*, pages 9–16, Feb 2013.

[CKB08] Andy Cockburn, Amy K. Karlson, and Benjamin B. Bederson. A review of overview+detail, zooming, and focus+context interfaces. *ACM Comput. Surv.*, 41:2:1–2:31, 2008.

[Duc77] Jean Duchon. *Splines minimizing rotation-invariant semi-norms in Sobolev spaces*, pages 85–100. Springer Berlin Heidelberg, Berlin, Heidelberg, 1977.

[EBD05] Geoffrey Ellis, Enrico Bertini, and Alan Dix. The sampling lens: Making sense of saturated visualisations. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '05, pages 1351–1354, New York, NY, USA, 2005. ACM.

[FMH16] Bernhard Froehler, Torsten Möller, and Christoph Heinzl. Gemse: Visualization-guided exploration of multi-channel segmentation algorithms. 35, 06 2016.

[GNBP11] Rocco Gasteiger, Mathias Neugebauer, Oliver Beuing, and Bernhard Preim. The flowlens: A focus-and-context visualization approach for exploration of blood flow in cerebral aneurysms. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2183–2192, December 2011.

[Hau06] Helwig Hauser. *Generalizing Focus+Context Visualization*, pages 305–327. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[HS17] Christoph Heinzl and S Stappen. Star: Visual computing in materials science. 36:647–666, 06 2017.

[JMI15] Hans J Johnson, Matthew M McCormick, and Luis Ibanez. *The ITK Software Guide Book 1: Introduction and Development Guidelines*, 4th edition, 2015. ISBN 9781-930934-28-3.

[KCJ+10] P. Karnick, D. Cline, S. Jeschke, A. Razdan, and P. Wonka. Route visualization using detail lenses. *IEEE Transactions on Visualization and Computer Graphics*, 16(2):235–247, March 2010.

[LA94] Y. K. Leung and M. D. Apperley. A review and taxonomy of distortion-oriented presentation techniques. *ACM Trans. Comput.-Hum. Interact.*, 1(2):126–160, June 1994.

[Rob07] J. C. Roberts. State of the art: Coordinated multiple views in exploratory visualization. In *Fifth International Conference on Coordinated and Multiple Views in Exploratory Visualization (CMV 2007)*, pages 61–71, July 2007.

[RPK+12] A. Reh, B. Plank, J. Kastner, E. Gröller, and C. Heinzl. Porosity maps: Interactive exploration and visual analysis of porosity in carbon fiber reinforced polymers. *Comput. Graph. Forum*, 31(3pt3):1185–1194, June 2012.

[SB92] Manojit Sarkar and Marc H. Brown. Graphical fisheye views of graphs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '92, pages 83–91, New York, NY, USA, 1992. ACM.

[SB94] Manojit Sarkar and Marc H. Brown. Graphical fisheye views. *Commun. ACM*, 37(12):73–83, December 1994.

[Shn96] B. Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE Symposium on Visual Languages*, pages 336–343, Sep 1996.

[SML06] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualisation Toolkit*, 4th edition, 2006. ISBN 978-1-930934-19-1.

[TAS09] Christian Tominski, James Abello, and Heidrun Schumann. Cgv—an interactive graph visualization system. *Computers & Graphics*, 33(6):660 – 678, 2009.

[WAG+16] J. Weissenböck, A. Amirkhanov, E. Gröller, J. Kastner, and C. Heinzl. Porosityanalyzer: Visual analysis and evaluation of segmentation pipelines to determine the porosity in fiber-reinforced polymers. In *2016 IEEE Conference on Visual Analytics Science and Technology (VAST)*, pages 101–110, Oct 2016.

[WAL$^+$14] J. Weissenböck, A. Amirkhanov, W. Li, A. Reh, A. Amirkhanov, E. Gröller, J. Kastner, and C. Heinzl. Fiberscout: An interactive tool for exploring and analyzing fiber reinforced polymers. In *2014 IEEE Pacific Visualization Symposium*, pages 153–160, March 2014.

[WCG03] N. Wong, S. Carpendale, and S. Greenberg. Edgelens: an interactive method for managing edge congestion in graphs. In *IEEE Symposium on Information Visualization 2003 (IEEE Cat. No.03TH8714)*, pages 51–58, Oct 2003.

[WGK10] Matthew Ward, Georges Grinstein, and Daniel Keim. *Interactive Data Visualization - Foundations, Techniques, and Applications.* 01 2010.

[WZMK05] L. Wang, Y. Zhao, K. Mueller, and A. Kaufman. The magic volume lens: an interactive focus+context technique for volume rendering. In *VIS 05. IEEE Visualization, 2005.*, pages 367–374, Oct 2005.

[ZCPB11] J. Zhao, F. Chevalier, E. Pietriga, and R. Balakrishnan. Exploratory analysis of time-series with chronolenses. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2422–2431, Dec 2011.