eingereicht von: Claudia Stripf
geb. am 07. Juni 1972 in Karlsruhe

Geometry-based and Texture-based Visualization of Segmented Tensor Fields

Diplomarbeit

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

Berlin, den 1. August 2011

Erstgutachter:
Prof. Dr. Peter Eisert

Zweitgutachter:
Dr. Ingrid Hotz

Table of Contents

List of Figures

Zusammenfassung

Tensoren werden häufig verwendet um physikalische und mechanische Phänomene zu beschreiben. Der Begriff der Tensoren wurde in der Physik eingeführt und später vor allem im Bereich der Tensoralgebra mathematisch präzisiert. Tensoren sind in verschiedenen wissenschaftlichen Bereichen geläufig, z. B. der Medizin, der Geologie, der Physik und der Mechanik. Dennoch sind Tensoren aufgrund ihrer Komplexität oft nur schwer zu analysieren und interpretieren, dies hat zur Entwicklung der Tensorfeldvisualisierung geführt. Die Tensorfeldvisualisierung ist ein Teilgebiet der wissenschaftlichen Visualisierung. Deren Methoden konzentrieren sich vor allem auf die Visualisierung skalarer Werte und Vektorfelder. Diese Methoden müssen für die Tensorfeldvisualisierung erweitert werden, da Tensoren mehr Informationen codieren. Die Bedeutung der einzelnen Komponenten eines Tensors ist ohne Kontext nicht offensichtlich. Vielmehr bildet ein Tensor eine Einheit, welche als Ganzes analysiert werden muss. Dies kann durch eine Analyse der Invarianten eines Tensors durchgeführt werden, z. B. der Eigenwerte und Eigenvektoren. Der Schwerpunkt in dieser Arbeit liegt auf Spannungstensoren aus der Mechanik.

In dieser Arbeit wurden zwei verschiedene Visualisierungsmethoden für zweidimensionale segmentierte Tensorfelder entwickelt. Eine diskrete Visualisierungsmethode, welche die Spannungen im Tensorfeld anhand von geometrischen Objekten sichtbar macht, und eine stückweise kontinuierliche Visualisierungsmethode, welche die Spannungen im Tensorfeld anhand von Texturen veranschaulicht. Bei der Segmentierung des Tensorfeldes werden Regionen mit ähnlichen Eigenschaften extrahiert. Diese Regionen sind als Zellen gegeben und dienen als Grundlage für die Visualisierungen. Die Aufgabe der Visualisierung mittels geometrischer Objekte ist es, lokale Repräsentanten in den Zellen zu platzieren, so dass die Eigenschaften des Tensorfeldes deutlich werden. Um diese Eigenschaften zu visualisieren, werden die Eigenwerte auf ein beschränktes positives Intervall abgebildet. Dieser Ansatz ist eine Modifikation der in der Tensorfeldvisualisierung bekannten Methode des „glyph placement“. Im Gegensatz zur diskreten Visualisierung des Tensorfeldes mittels geometrischer Objekte liefert der texturbasierte Ansatz eine stückweise kontinuierliche Visualisierung. Die charakteristischen Merkmale wie Druck und Zug werden dabei durch Texturen wider gegeben. Dabei wird einerseits eine Eingabetextur nach dem zugrunde liegenden Eigenvektorfeld ausgerichtet, andererseits durch die transformierten Eigenwerte skaliert und auf die einzelnen Zellen aufgetragen.

Beide Visualisierungsmethoden werden anhand von zwei simulierten Beispielen demonstriert und miteinander verglichen.

Abstract

Tensors are commonly used to describe physical and mechanical phenomena. The concept of tensors was introduced in physics and later refined in mathematics, especially in the field of tensor algebra. Tensors are used in different scientific areas, for example medicine, geology, physics and mechanics. However, due to the inherent complexity of tensor data, analysis and interpretation is often difficult; this has led to the development of tensor data visualization methods. Tensor field visualization is a sub-branch of scientific visualization. Scientific visualization focuses on scalar-based and vector-based methods. For tensor field visualization these methods are extended, as tensors are multivariate by nature and encode more information. The meaning of individual components of a tensor is not obvious without context; a tensor forms one entity that must be analyzed as a whole. The tensor invariants, e.g. eigenvalues and eigenvectors, provide a useful means of analysis. The focus in this thesis is on stress and strain tensors.

This work is concerned with two different visualization approaches for 2D segmented tensor fields. Based on a pre-computed segmentation of the original tensor field, a geometry-based discrete visualization and a piecewise continuous texture-based visualization are presented. The segmentation of the tensor field allows extraction of regions with similar properties. These regions are given as explicit cells and serve as the basis for the visualization. The task of the geometry-based visualization approach is to place a geometric icon in each extracted cell. The icon represents chosen tensor characteristics for the extracted cell. In order to visualize these characteristics, the eigenvalues are transformed to a restricted positive interval. This approach is a modification of glyph placement, which is a well-known in tensor field visualization method. In contrast to discrete visual representation of the tensor field by glyphs, texture-based visualization provides a piecewise continuous representation. The characteristic features of the tensor field - regions and directions of compressive and expansive forces - are represented by textures. An input texture pattern is aligned according to the eigenvector field, scaled by the transformed eigenvalues and mapped onto each cell.

As proof of concept, the methods developed in this thesis are applied to synthetic datasets and the visualization approaches are compared.

Chapter 1

Challenges in Tensor Field Visualization and Related Work

1.1 Introduction

The word *tensor* and the concept of tensor analysis were introduced in the middle of the nineteenth century. Tensor fields or stress tensors are common terms to describe physical and mechanical phenomena and quantities. The visualization of for example stress and strain tensor fields is used to gain more insight into structure and tool strain. Since tensors extend the concept of scalars, geometric vectors and matrices to higher orders, tensors and tensor fields encode information of higher complexity. A tensor can be represented by a multi-dimensional array of numerical values, with respect to a given frame of reference.

The goal of tensor field visualization is to render a given dataset in order that the characteristics of the tensor field can be recognized. One basic approach is to depict the tensor dataset based on scalar visualization techniques. In doing so, every numerical value of the multi-dimensional array in a tensor dataset will be displayed as a separate scalar field; however, this approach is accompanied by a loss of important information and coherencies. Visualizations based on a eigendecomposition display more characteristic information, in particular they are invariant under a change of the reference frame. By means of the eigendecomposition the eigenvalues and eigenvectors of a tensor can be determined. The eigenvectors form a local coordinate system, in which the quantity encoded by the tensor reaches extremal values. These extremal values are called the eigenvalues. The eigenvalues and eigenvectors describe the tensor completely and they provide a basis for many of the tensor field visualization methods. For 2D tensor fields, a common approach is to observe the minor and major eigenvalues and the corresponding eigenvectors as separate "vector fields". Tensor field visualizations have to extend scalar and vector based visualization approaches, because tensors extend the concept of scalars and geometric vectors. The challenge in tensor field visualization is to encode complex data such that the important features and coherencies of tensor fields can be displayed, moreover the focus of the observer

can be directed to specific characteristics, which would not have been visible by a scalar-based or vector-based visualization method.

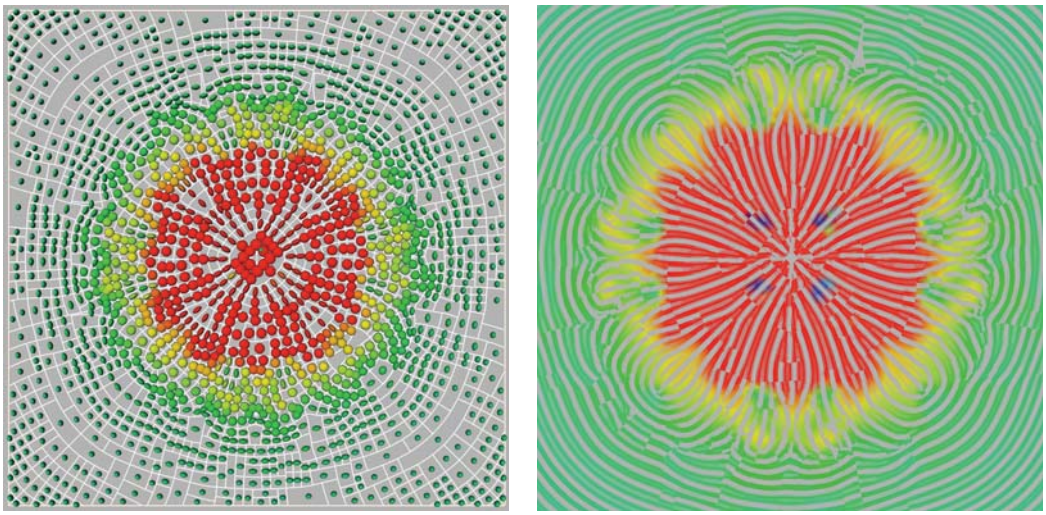
In order to reduce the amount of data that is necessary to encode the information of a tensor dataset, recent research has focused on a segmentation of tensor fields [3, 23, 27]. One possible approach for the segmentation is to consider the topological structure. This segmentation aggregates regions of similar behavior of the tensor field in a given domain. They are bounded by the tensor lines of the minor and major eigenvector fields. The starting point for the work presented here is the segmentation algorithm presented by Auer et al. [3]. Most of the extracted regions are curvilinear cells with specific characteristics. These specific characteristics are the motivation for a new geometry-based and texture-based visualization approach.

One simple and fast way to encode the tensor data in a discrete manner is glyph placement. In glyph placement the tensor dataset values are mapped onto geometric icons. A geometric icon (glyph) reflects various attributes of the tensor. These attributes can be mapped onto location, direction, orientation, size and color. Glyph placement is a discrete visualization method and cannot convey information about every point. Texture-based tensor field visualization concepts, in contrast, present the information of the tensor field to the observer in a piecewise continuous signal by the use of textures. A well-known texture-based visualization method is tensor line integral convolution (LIC), which encodes the direction of single eigenvector fields.

In our framework the extracted cells of the segmentation are used for both glyph placement and texturization. Due to the simplification and classification of the tensor field into characteristic regions we are able to encode more information in a single visualization pass as a single glyph represents a whole region. The similar structure of the extracted segments is the primary the reason for the texture mapping. Most of the cells have three or four corner points, only few of them are of higher complexity. Moreover, the segmentation inherently provides the parametrization for the texture mapping.

The realization of our framework deals with different challenges with regard to the field of visualization and uses the methods of computer graphics. The task is to evolve a different geometry-based and texture-based visualization method, founded on a given segmentation. For the discrete, geometry-based approach, the center of each arbitrary shaped planar cell has to be computed. This is achieved according the algorithm developed by Rustamov et al. [22]. For the piecewise continuous texture-based visualization method the algorithm presented by Hummel et al. [18] is extended. In contrast to Hummel, the illustrative rendering of integral surfaces is applied to tensor fields. Therefore, the texture coordinates of every extracted cell, which takes into account the shape of the polygon and the information about the bounding tensor lines, require computation. Moreover, the framework presented here explores and evaluates the practice of different texture patterns. The input pattern can either be procedural or originate from an arbitrary texture image. Figure 1.1 shows an example of the geometry-based and texture-based visualization approach developed in this

diploma thesis.



(a) (b)

Figure 1.1
Tensor field visualization: (a) geometry-based and (b) texture-based.

The visualization methods presented here are implemented in Amira [25], a software system for visual data analysis. These methods are demonstrated by means of two mechanical examples, the one-point load and the two-point load. The one-point load displays the stress in a 3D-volume due to pressure. The two-point load displays the stress in a 3D-volume due to pressure and strain. At the beginning of the thesis a brief overview of the mathematical fundamentals of tensors, tensor fields and tensor field segmentation is given. The "Related Work" section reviews well-known tensor field visualization concepts. The geometry-based and texture-based visualization methods, which are founded on a given segmentation, form the main part. An evaluation of the developed tensor field visualization methods concludes this diploma thesis; in particular, the advantages and drawbacks of the visualization methods are discussed, and ideas for possible improvements are suggested.

1.2 Mathematical Fundamentals

Tensors have a long tradition in physics, engineering, and mathematics. In the next section, a brief overview of tensors and their properties is given. Since the underlying mathematics is very complex and beyond the scope of this work the interested reader is referred to the subject literature for further information [1, 14, 10]. This work is restricted to an intuitive but still very common definition. This will familiarize the reader with the theory that is necessary to understand tensor field visualization.

1.2.1 Notation

If not other specified the following notation holds for the work presented here. For tensors $\mathbf{T}$ as well as for matrices $\mathbf{M}$ bold capital letters are used; $\mathbf{I}$ refers to the identity matrix. We use $\vec{v}$ and $\vec{w}$ when referring to a directional vector, for eigenvectors we use $\overleftrightarrow{v}$ and $\overleftrightarrow{w}$ to signify the property of bidirectionality. Greek letters (i.e. α) denote scalars – particularly, $\lambda_1, \ldots, \lambda_n$ denote the eigenvalues. Lower case letters from the middle of the alphabet are used for integers (i, j, k, l, m, n). Furthermore, lower case letters from the end of the alphabet are used for real numbers (r, s, t, x, y).

1.2.2 What is a Tensor?

Physicists and engineers may give a different answer to the question: *What is a tensor?* The differing definitions of tensors describe the entity from different perspectives; moreover, they reflect the importance of tensors in various application areas. The thesis presented here uses the definition of a tensor as numerical array – a definition that remains popular in many physics and engineering text books. For further definitions, the interested reader is referred to subject literature [1, 14, 10].

Tensor as Multidimensional Array

As mentioned above, tensors extend the concept of scalar values, geometric vectors and matrices. Such a tensor $\mathbf{T}$ can be described by a multi-dimensional array of numerical values. A scalar is a zeroth-order tensor and a vector is a first-order tensor. A $k \times k$ matrix $\mathbf{T}$ can describe a second-order tensor. In general, an order-n tensor can be described by an n-dimensional array of k components with respect to a given reference frame. The matrix can be interpreted as a multilinear mapping from $T : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$:

$$T(\vec{v}, \vec{w}) = \vec{v}^T \cdot \mathbf{T} \cdot \vec{w} \ , \tag{1.1}$$

where $\cdot$ is the standard matrix multiplication.

1.2.3 Properties of Tensors, their Decomposition and Important Qualities

In a tensor field or tensor data set, every point in the domain corresponds to a tensor. Most of the tensor field visualization applications deal with second-order tensors. The two mechanical examples used to evaluate and demonstrate the developed visualization concept are three dimensional tensor fields: two cubic domains with different loads applied. For the segmentation the dimension is reduced by slicing. Therefore, in the following chapters of this work a tensor alludes to a 2-dimensional symmetric second-order tensor. This holds in particular for the this section, which describes the properties of symmetric second-order tensors.

Tensor Invariance

Tensors are not dependent on a special reference frame. They are *invariant* under coordinate transformations. A tensor can be transformed from one coordinate system into another. Particularly, this means that the tensor components change according to the transformation laws; however, the characteristics of the tensor stay the same.

Symmetry

A two-dimensional second-order tensor $\mathbf{S}$ is called *symmetric* if the corresponding matrix is symmetric. For a second order tensor $\mathbf{S}$ this is $S(\vec{v}, \vec{w}) = S(\vec{w}, \vec{v})$ for all $\vec{v}, \vec{w} \in V$, where V is an two-dimensional vector space. A tensor $\mathbf{A}$ is called *antisymmetric* if the sign changes when exchanging two adjacent arguments. For a second-order tensor $\mathbf{A}$ this is $A(\vec{v}, \vec{w}) = -A(\vec{w}, \vec{v})$ for all $\vec{v}, \vec{w} \in V$. Every tensor can be decomposed into a symmetric and antisymmetric part $\mathbf{T} = \mathbf{S} + \mathbf{A}$, where $\mathbf{S} = 1/2(\mathbf{T} + \mathbf{T}^t)$ and $\mathbf{A} = 1/2(\mathbf{T} - \mathbf{T}^t)$. The symmetric part describes the deformation and the antisymmetric part refers to the rotation.

Eigendecomposition and Diagonalization

Eigendecomposition reveals the internal structure, the variance in the data structure of $\mathbf{T}$. Such an eigendecomposition represents $\mathbf{T}$ in terms of its *eigenvalues* and *eigenvectors*.

$$\mathbf{T} = \sum_{i=1}^{2} \lambda_i \overleftrightarrow{v_i} \otimes \overleftrightarrow{v_i} \ .^{1} \tag{1.2}$$

$\mathbf{T}$ is fully described by its eigenvalues λ_1, λ_2 and corresponding eigenvectors $\overleftrightarrow{v_1}$, $\overleftrightarrow{v_2}$, such that $\mathbf{T} \cdot \overleftrightarrow{v_1} = \lambda_1 \cdot \overleftrightarrow{v_1}$, $\mathbf{T} \cdot \overleftrightarrow{v_2} = \lambda_2 \cdot \overleftrightarrow{v_2}$ and always $\lambda_1 > \lambda_2$. They can be computed by solving the *characteristic equation*:

$$det(\mathbf{T} - \lambda \mathbf{I}) = 0 \tag{1.3}$$

The eigenvectors give the direction of extremal variation of the quantity encoded by the tensor of a given point; the eigenvalues give the values of those extremal variation. The notation $\overleftrightarrow{v}$ and $\overleftrightarrow{w}$ implies the fact that the eigenvectors have no orientation. For symmetric tensors the eigenvalues are real and the eigenvectors are mutually orthogonal. The eigenvectors form a basis for the diagonal representation of the tensor. The eigenvectors that correspond to the larger eigenvalue λ_1 are called major eigenvectors. The eigenvectors that correspond to the smaller eigenvalue λ_2 are called minor eigenvectors.

[1] $\otimes$ denotes the outer product, with $\begin{pmatrix} v_1 \\ v_2 \end{pmatrix} \otimes \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} v_1 w_1 & v_1 w_2 \\ v_2 w_1 & v_2 w_2 \end{pmatrix}$

Definiteness

Tensors can be distinguished by their *definiteness*. A tensor $\mathbf{T}$ is called

positive definite, if $\vec{x}^T\mathbf{T}\vec{x} > 0$, for all non-zero $\vec{x} \in \mathbb{R}^2$.

negative definite, if $\vec{x}^T\mathbf{T}\vec{x} < 0$, for all non-zero $\vec{x} \in \mathbb{R}^2$.

positive semi-definite, if $\vec{x}^T\mathbf{T}\vec{x} \geq 0$, for all non-zero $\vec{x} \in \mathbb{R}^2$.

negative semi-definite, if $\vec{x}^T\mathbf{T}\vec{x} \leq 0$, for all non-zero $\vec{x} \in \mathbb{R}^2$.

If none of the characteristics applies, the tensor is called *indefinite*. For positive definite tensors $\lambda_i > 0$. For positive semi-definite tensors $\lambda_i \geq 0$. For the evaluation of the work presented here stress and strain tensors are used, which are symmetric, but in general not positive semi-definite. Examples of positive semi-definite tensors are diffusion and deformation tensors.

Polar Decomposition

For a positive definite tensor $\mathbf{T}$, e.g. a diffusion tensor, the *polar decomposition* is useful. The polar decomposition decomposes the transformation represented by $\mathbf{T}$ into two stages: a rotation $\mathbf{R}$ and a right stretch $\mathbf{U}$ or a left stretch $\mathbf{V}$.

$$\mathbf{T} = \mathbf{R} \cdot \mathbf{U} = \mathbf{V} \cdot \mathbf{R} \tag{1.4}$$

If the tensor is symmetric and positive definite, the tensor is called *stretch*. If the tensor is orthogonal with the determinant equal to one, the tensor is called *rotation*.

Anisotropy and Isotropy

Anisotropy is the property of being direction dependent, i.e. an anisotropic medium exhibits different properties along different directions, whereas an **isotropic** medium exhibits the same property in all directions around a given point. A tensor $\mathbf{T}$ can be decomposed into an isotropic and anisotropic or deviatoric part

$$\mathbf{T} = \frac{1}{2}tr(\mathbf{T}) \cdot \mathbf{I} + \mathbf{D} \ . \tag{1.5}$$

The isotropic part represents a direction independent transformation, i.e. uniform scaling or uniform pressure. The deviatoric part represents the distortion. The anisotropy can be displayed by scalar visualization methods. To express the anisotropy several measures have been proposed. A common measure in context of diffusion tensors is the *fractional anisotropy* introduced in [5]

$$FA = \sqrt{\frac{1}{2}} \frac{\sqrt{\sum_{i=1}^{2}(\lambda_i - \mu)^2}}{\lambda_1^2 + \lambda_2^2} \ , \tag{1.6}$$

and the *relative anisotropy* also introduced in [5]

$$RA = \sqrt{\frac{1}{2}} \frac{\sqrt{\sum_{i=1}^{2} (\lambda_i - \mu)^2}}{\lambda_1 + \lambda_2} \ , \tag{1.7}$$

where $\mu = 1/2(\lambda_1 + \lambda_2)$ is the mean diffusivity.

Shear Stress

Stress is a measure of internal forces acting within a tensor field. These internal forces are a reaction to external forces applied to a medium. Concerning the whole tensor field, directions of cutting planes that exhibit the maximum tangential or maximum shear stress are of special interest. The directions as well as the magnitude are of importance for failure analysis. Different shape descriptors and failure models describe the magnitude of shear stress. Similar to anisotropy, these models encode scalar tensor field information and can be visualized by scalar tensor field visualization methods. The magnitude of shear stress can be computed by the following formula:

$$\sigma = \lambda_1 - \lambda_2 \ . \tag{1.8}$$

1.2.4 Tensors and their Application

Due to the well-founded mathematical framework, tensors are used in a variety of application areas. This section introduces commonly used types of tensors. The goal of this section is not to give a complete overview of all tensors. and their application areas. Rather, several common kinds of tensors are singled out, along with examples of their application and characteristic properties.

Diffusion Tensors

Medicine is one example of the use of *diffusion tensors*. A diffusion tensor describes the anisotropic diffusion behavior of water molecules in tissue for example when studying the control nervous system. Considering the whole diffusion tensor field, one is typically interested in the regions with high anisotropy, since the anisotropy is high in the direction of the neuronal fibers. These are regions with strongest diffusion. With this characteristic of the diffusion tensor field it is possible to examine the fibrous structure of the underlying tissue. This can give physicians information concerning the diseases and abnormalities of the central nervous system. A diffusion tensor contains the following information: the principal diffusion direction, its strength and its anisotropy. The rate of diffusion is the same in directly opposing directions. This means that in regard to the properties of a diffusion tensor it is symmetric and positive semi-definite.

Curvature and Metric Tensor

In geometrical applications *curvature* and *metric tensors* are often used. A *curvature tensor* describes the change in the surface normal of a geometric object in any given

direction. Often the user is interested in the principal curvature direction and its extremes. This information can be extracted by eigendecomposition; the eigenvectors describe the direction of maximal and minimal curvature, and the corresponding eigenvalues give the quantities of the curvature. The sign of the eigenvalues encode whether the surface is locally convex, concave or saddle shaped. Particularly, the characteristic properties of curvature tensors are symmetry and indefiniteness.

In geometry metric tensors are used to measure angles and the length of vectors, independently to a given reference frame. This encodes the commonly known dot product. Like curvature tensors, the metric tensors are symmetric; however, in Euclidian space the metric tensors are positive definite.

Stress and Strain Tensors

Stress and *strain tensors* are commonly used in mechanical engineering. Together, they specify the behavior of a continuous medium under load, giving information about a material's stability. Stress tensors encode information about the stress within the volume of a specific material under certain external loads: its direction, its strength, its anisotropy and its compressive or tensile forces, which can be distinguished by the sign of the eigenvalues. In mechanical engineering, negative eigenvalues refer to compressive forces and positive eigenvalues refer to tensile forces. Therefore, stress and strain tensors are symmetric but in general not positive semi-definite. Figure 1.2 shows the deformation of a unit probe under influence of a stress tensor [16].

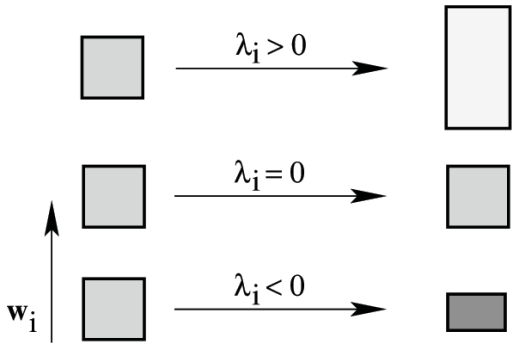
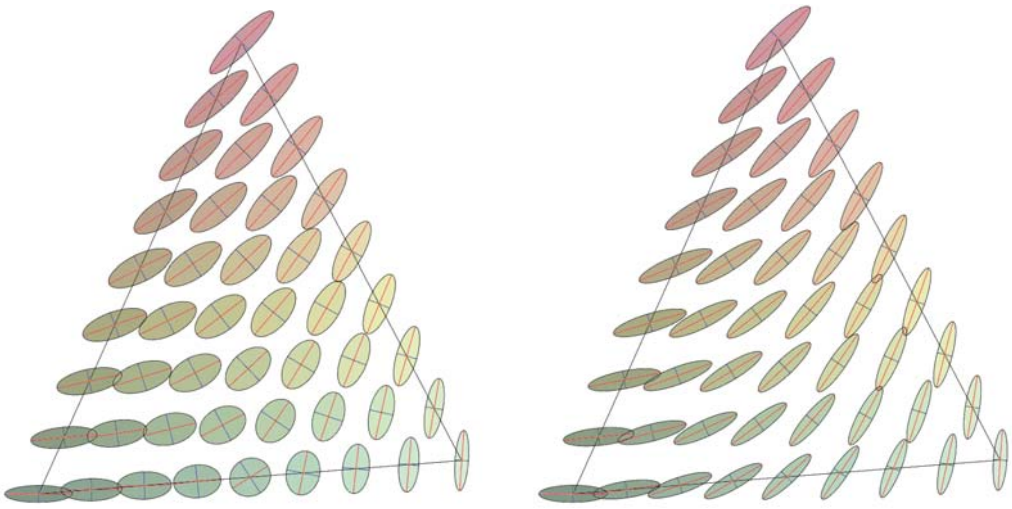


Figure 1.2
Deformation of a unit probe for a force parallel to eigenvector in direction w_i [16]. Image courtesy Hotz.

Tensor Field and Tensor Interpolation

In a **tensor field** or tensor dataset over a given domain $D \subset \mathbb{R}^2$ a matrix $\mathbf{T}(p)$ corresponds to every point $p \in D$. However, such a sampled dataset on a given domain or grid describes an intrinsically continuous quantity. To reconstruct or approximate the continuous function an interpolation scheme is necessary. Especially, for performing

the segmentation, the basis of our tensor field visualization framework, one has to adaptively refine the given tensor dataset. For these newly inserted grid points the tensor values have to be interpolated. Common interpolation schemes can be divided into two groups: tensor interpolation is applied to single components of the tensor and interpolation schemes that use the tensor invariants, i.e. the eigenvalues and the eigenvectors. The first interpolation scheme has the drawback that the invariants of a tensor are in general not linear. More advanced algorithms have been presented in [2, 21] to preserve the features of the tensor data. In the given segmentation and presented visualization framework the tensor data are interpolated by the latter method [17]. The method presented by Hotz et al. [17] decouples the direction and the shape; particularly, this has the advantage that the interpolation is shape-preserving (see Figure 1.3).



(a) (b)

Figure 1.3
Comparison between (a) linear component-wise interpolation and (b) shape-preserving linear tensor interpolation based on the tensor invariants. Image courtesy Hotz.

1.2.5 Topology and Singularities

Topology extraction is a method that separates a scalar, vector or tensor field into regions with qualitatively homogenous behavior. The topology is a graph with nodes and connecting edges. This leads to a topological skeleton of the field, concerning the eigenvector directions, and gives a global overview.

Vector Field Topology and Singularities

One way of starting to understand the topology of tensor fields is to look at the topology of stationary vector fields. For convenience a 2D vector field is defined as a

mapping $f : D \to \mathbb{R}^2$, where D is a subset of $\mathbb{R}^2$.

The flow behavior can be classified by the important quantities *divergence* and *vorticity* (see Figure 1.4). The divergence is the magnitude of a source or sink at a given point in a vector field. The vorticity is the tendency for elements of a vector field to rotate.

- "A positive divergence at p denotes that mass would spread from p outward. Positive divergence points are called *sources*" [26].
- "A negative divergence at p denotes that mass gets sucked into p. Negative divergence points are called *sinks*" [26].

(a) (b) (c)

Figure 1.4
(a) *Source point*, (b) *sink point* and (c) *hight-vorticity field*. From [26].

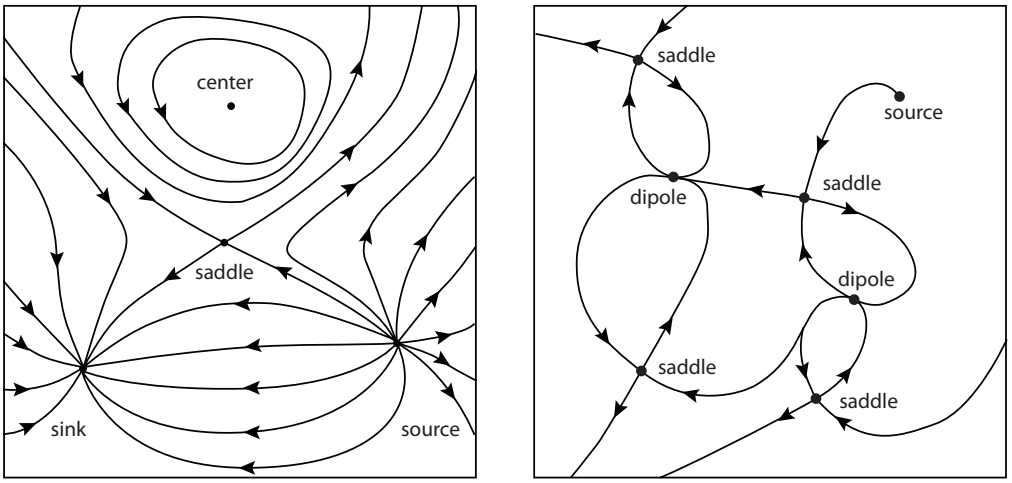
Such a vector field can be displayed by placing a line glyph, with the direction and magnitude of the vector, at the position p of a given Domain D. These glyphs can be seen as "*trajectories* over a short time interval Δt" [26]. *Stream objects* visualize such trajectories over a longer period of time. These stream objects or *stream lines* show the flow of a vector field, and the stream line in a given location p is tangent to the vector at p. A formal definition is given in [26]. An integral curve or stream line "is the curved path that an imaginary particle would pass over a given integral interval T and a given start location or *seed* p_0" [26], i.e. from a source to a sink. This leads to the formal definition of stream lines [26]:

$$S = \{p(\tau), \tau \in [0, T]\}, p(\tau) = \int_{t=0}^{\tau} \vec{v}(p)dt \text{ ,where } p(0) = p_0 \text{ and } \vec{v} = (v_1, v_2) \quad (1.9)$$

This equation can be numerically solved, for example, by Euler integration, by discretizing the time t and replacing the integral with the finite sum

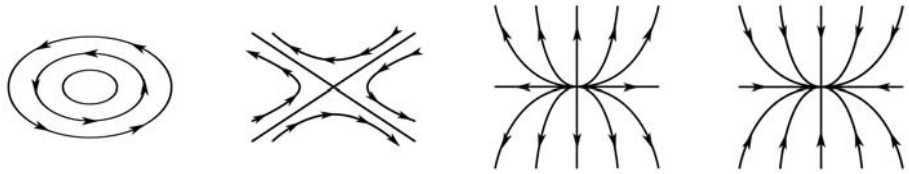
$$\int_{t=0}^{\tau} \mathrm{v}(p)dt = \sum_{i=0}^{\tau/\Delta t} \vec{v}(p_i)\delta t \text{ ,where } p_i = p_{i-1} + \mathrm{v}_{i-1}\Delta t. \quad (1.10)$$

The resulting stream line is approximated by a piecewise-linear curve or poly-line. Stream lines of a vector field are depicted in Figure 1.5.



(a) (b)

Figure 1.5
Outline of two different vector fields: (a) stream lines and (b) topology.



(a) (b) (c) (d)

Figure 1.6
Examples of first-order critical points: (a) center, (b) saddle point, (c) source and (d) sink.

Critical points in a vector field are singularities in the field such that $\vec{v}(p) = 0$. Critical Points of a vector field can be classified by the eigenvalues of the Jacobian[2] matrix $\mathbf{J}$ at a position P:

$$\mathbf{J}(P) = \begin{pmatrix} \frac{\partial v_x}{\partial x} & \frac{\partial v_x}{\partial y} \\ \frac{\partial v_y}{\partial x} & \frac{\partial v_y}{\partial y} \end{pmatrix}. \tag{1.11}$$

An overview of the first-order critical points is given in Figure 1.6.

Separatrices are distinguished stream lines that separate a vector field into regions with similar behavior.

The **topology** of a vector field (Figure 1.5(b)) is a graph with the critical points as nodes and the separatrices as edges. This means that in the topological graph or

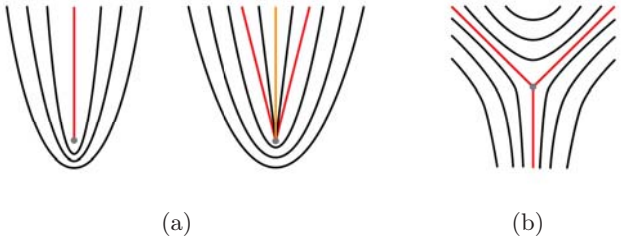
[2] $n \times m$-Matrix of all first-order partial derivatives of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}^m$

structure of a vector field a stream line from the inside of one extracted region cannot cross a separatrix and enter another extracted region.

Tensor Field Topology and Singularities

To review Section 1.2.4, in a tensor field or tensor dataset a matrix $\mathbf{T}(p)$ corresponds to every point $p \in D$, where $D \subset \mathbb{R}^2$. Tensors can be described by their invariants: the eigenvalues and eigenvectors. The eigenvector fields can be integrated. Integrating these two eigenvector fields yields two orthogonal families of continuous curves, the **tensor lines**. These tensor lines are used for the topology-based segmentation, which is introduced later in this chapter.

Degenerate points in a tensor fields correlate to critical points in vector fields. These singularities are points where the eigenvalues are equal to each other, hence the eigenvectors are no longer defined uniquely. In a tensor field there are two kinds of elementary degenerate points: wedges and trisectors (see Figure 1.7). These elementary degenerate points can combine to form more, generally unstable structures in a tensor field – such as saddles, nodes, centers or foci. For further details the reader is referred to [13].



(a) (b)

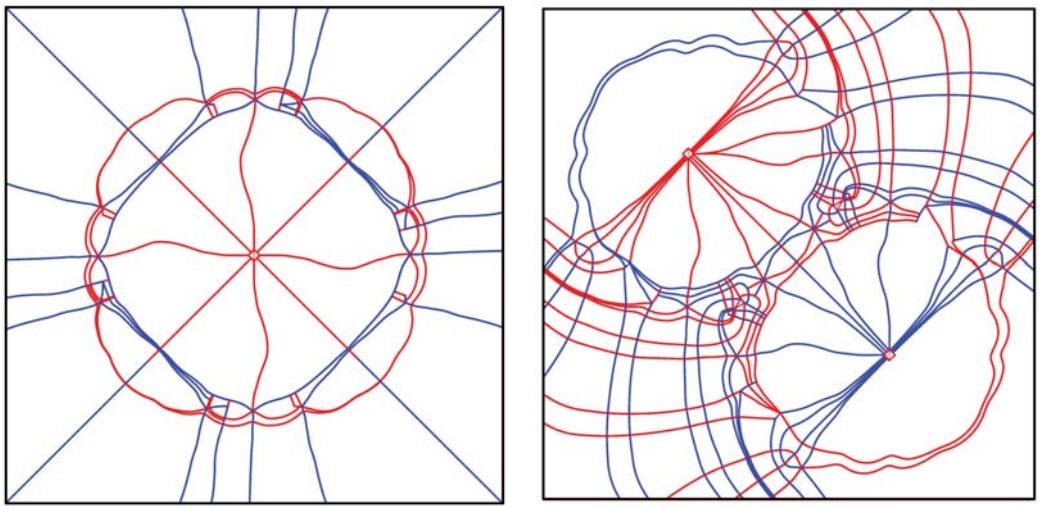
Figure 1.7
First-order degenerate points: (a) two types of wedge points and (b) trisector

Separatrices are distinguished tensor lines that *separate* the domain into regions with qualitatively homogenous eigenvector behavior. In a tensor field we have to distinguish between two classes of separatrices, one for the major eigenvector field and one for the minor eigenvector field. Similar to vector fields, a tensor line belonging to the major or minor eigenvector cannot cross the corresponding separatrix and enter into another extracted cell.

Similar to vector fields, the structure of a tensor field is defined by its topology. The **topology** of a tensor field gives global information about the tensor field. Distinguished points (degenerate points) and distinguished tensor lines (separatrices) form the skeleton for the topological structure. In one extracted region of the topological structure all tensor lines have a similar behavior. When computing the topological graph and the segmentation, the tensor lines must be computed. In the algorithm used for this framework, the tensor lines are computed by a Runge Kutta 4^{th} order integration scheme. The step size for the integration is adapted to the change of the eigenvector field.

1.3 Topology Extraction and Segmentation

As an input for the geometry-based or texture-based approach presented here, either the topological graph or the extracted segmentation is used. The topology extraction and segmentation used in this framework is based on the approach presented by Auer et al. [3]. In [3], the topological structure of both eigenvector fields are taken into account. These extracted topologies are based on distinguished points, the so called degenerate points, and connecting edges, the separatrices. The topological graphs of both eigenvector fields are combined.



(a) (b)

Figure 1.8
Topological skeleton of the major (red) and minor (blue) eigenvector field: (a) one-point load and (b) two-point load.

1.3.1 Topology Extraction

Separatrices exhibit certain characteristic behaviors in the vicinity of a degenerate point. Accordingly, one can distinguish between basic sectors with similar properties. In particular, they enter a degenerate point radially, constitute the edges of the topological graph and, moreover, depict the base for the segmentation. The neighborhood of a degenerate point and the nearby separatrices are characterized by a number of *half-sectors*. One can distinguish between three different sector types of similar characteristic behavior; *hyperbolic*, *parabolic* and *elliptic* sectors (see Figure 1.9):

- **Hyperbolic sector** – the tensor lines never reach the degenerate point
- **Parabolic sector** – the tensor lines end at the degenerate point
- **Elliptic sector** – all tensor lines begin and end at the same degenerate point

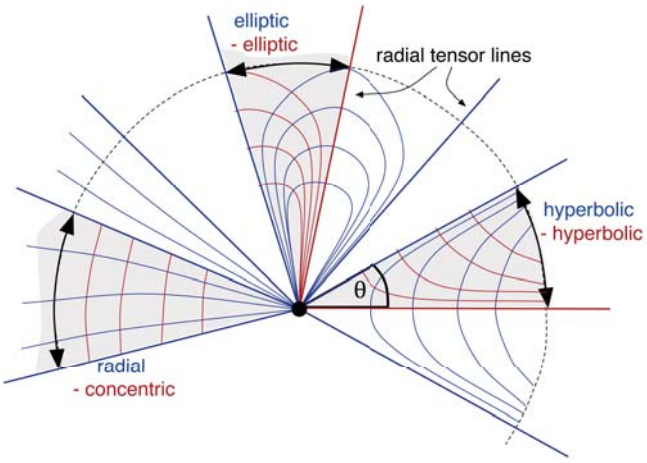


Figure 1.9
Half-sectors in the vinicity of degenerate points. Image courtesy Auer.

The topology extraction in this thesis only makes use of the separatrices that bound hyperbolic sectors.

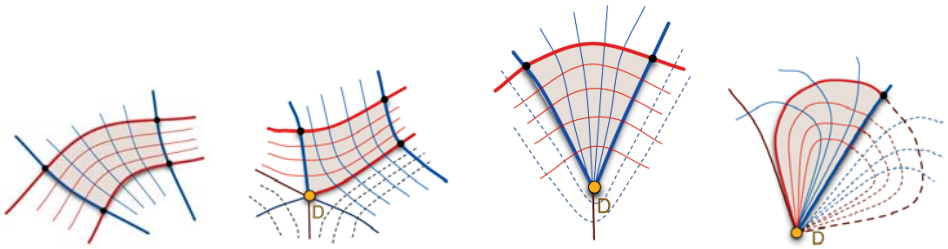


Figure 1.10
Cells defined by topological skeleton: (a) regular cell without degenerate points, (b) hyperbolic sector, (c) parabolic sector and (d) elliptic sector. Image courtesy Hotz.

The resulting topology extraction (see Figure 1.8 for the two- and one-point load), is composed of the topological skeleton of the major and minor eigenvalue field, the red and blue tensor lines, with the following four characteristics (cells containing degenerate lines were not taken into consideration, they can exhibit even more complicated structures).

1. Cells without degenerate points
   → quadrangle with alternating red and blue tensor lines; all "angles are orthogonal" [3] (see Figure 1.10(a))

2. Cells with one degenerate point lying on a hyperbolic sector
   → quadrangle with alternating red and blue tensor lines; "the angle at the

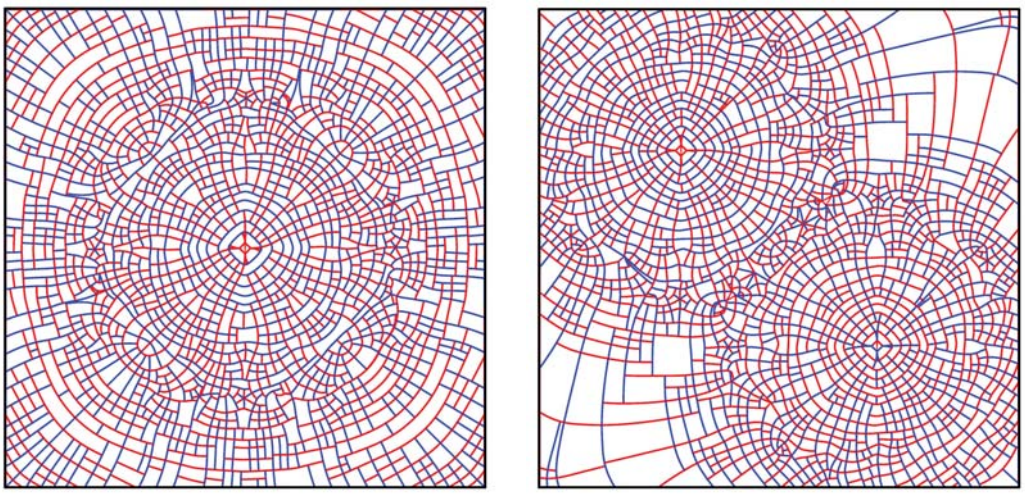
degenerate point is in general not orthogonal" [3] (see Figure 1.10(b))

3. Cells with one degenerate vertex lying on a parabolic sector
   → triangular shape (see Figure 1.10(c))

4. Cells lying on a elliptic sector
   → "cells with either two or three vertices are possible" [3] (see Figure 1.10(d))

1.3.2 Segmentation

The topological graph can be further adaptively refined to achieve a pre-determined resolution, accuracy and uniformity of the segmentation. This adaptive refinement workflow involves both subdivision and merging of cells, based on scalar invariants as similarity or dissimilarity measurements. These scalar invariants can be, for example, the anisotropy or shear stress. In the segmentation approach implemented in Amira several dissimilarities measures are combined. The final extracted segmentation can be considered as a "visualization of glyphs in the form of tiles" [3]. The geometry of the tiles or cells exhibits statistic similarities. Most of the cells exhibit triangular and quadrangular structures; however, other geometric structures appear mostly on the borders of the tensor field or in regions that contain degenerate points.



(a) (b)

Figure 1.11
The bounding major tensor lines of the cells are marked in red and the bounding minor tensor line are marked in blue: (a) segmentation of the one-point load and (b) the two-point load.

1.4 Related Work

Often the tensor field visualization methods arise from vector field visualization. These methods frequently concentrate only on the eigenvectors and neglect important quantities of the tensor, for example the eigenvalues or the anisotropy of a tensor field. Furthermore, these visualization methods are not able to provide an intuitive physical interpretation. A lot of researchers focussed on diffusion tensor field visualization; however, often these methods are not appropriate for other contexts. In the following section the visualization concepts related to the framework developed in this thesis are introduced.

1.4.1 Related Visualization Concepts

Among the different visualization concepts for tensors one can distinguish between local and global visualization concepts. Geometric objects, for examples *glyphs* and tensor *splats* [6] provide local methods. These can display the local tensor properties very well, but they have the drawback that they cannot display coherencies between the sampled data points. Local methods fail to provide a global overview. Global visualization methods, on the other hand, are more appropriate if one wants to display an overview and emphasize regional coherency; however, often they are restricted to scalar-valued or vector-valued features. Line integral convolution (LIC) [8] is a global visualization method for vector fields. HyperLIC [28] adapted the LIC algorithm to tensor fields. Hotz et al. [16] extended both algorithms, with the focus on the physical interpretation of tensor fields.

Geometry-Based Visualization Concepts

One basic approach of tensor field visualization is using glyphs. A glyph is a geometric shape that represents the tensor at a given point. One simple example for a glyph is an ellipsoid. The glyph is aligned to the eigendirections and scaled according to the corresponding eigenvalues. Other examples are cuboids, cylinders and superquadrics [4]. Although the glyphs represent the invariants of the tensor it is difficult to obtain a global overview of the complete tensor field. The choice of the geometry determines the tensor information that can be encoded. One problem is the trade-off between the sampling density and the glyph geometry or, in other words, the trade-off between the clarity of perception and the information content. Hence, the choice of the geometry should be connected to the glyph placement algorithm and the sampling density. To avoid cluttering and occlusion in dense glyph visualization the glyphs are scaled down. Therefore, dense glyph visualization should use less complex geometries, whereas the visualization of selected regions (probing) can use more elaborate geometries. Complex glyphs, encoding more features than the basic invariants, were introduced by Leeuw et al. [12] in the context of flow probes.

For stress and strain tensors the visualization goal is to encode compressive and expansive forces. They are encoded in the sign of the eigenvalues. A simple method

to depict compressive and expansive forces is to encode them in the color of the glyph surface. Other approaches normalize the eigenvalues or apply a positive-definite metric to the shape descriptor of the geometry. Such a metric maintains the physical meaning of the tensors [16]. Glyph-based visualization methods are very well-known and commonly used in tensor field visualization, although there are a lot of unsolved problems: cluttering and occlusion, the sign of the eigenvalues demands special glyph design or eigenvalue mapping, and the perception of the encoded information depends on the sampling density. The information of a single glyph can be depicted very well; in contrast, the global information, the coherency between single values of the glyphs, cannot be displayed.

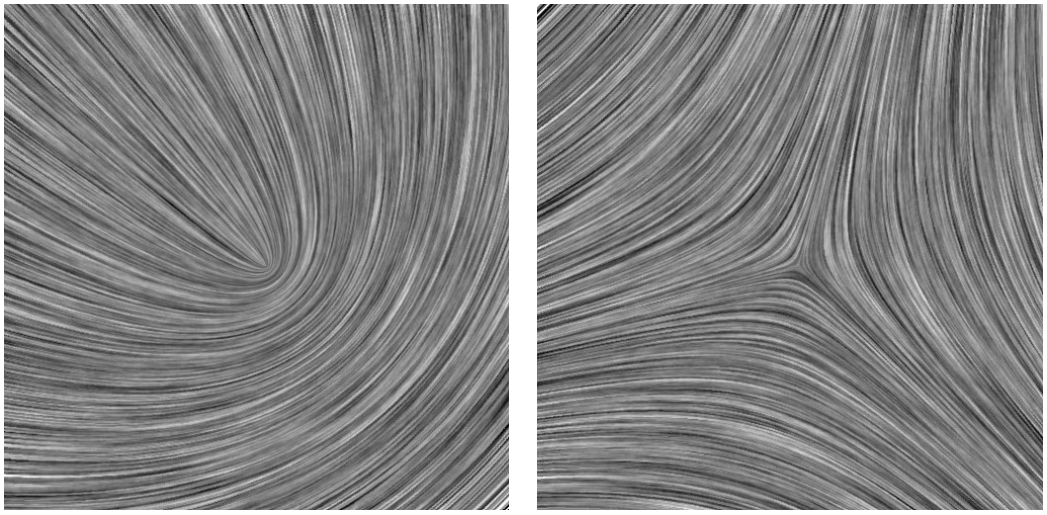
The basic idea of tensor splats [6] is to employ flat, planar icons that are able to encode the same information as ellipsoids. In contrast to ellipsoids, the icons have a color value, an opacity value and fade out smoothly. Thus, visual clutter is reduced and regions of interest are emphasized. For example, for diffusion tensors the anisotropy can be encoded in the opacity. Regions of high opacity emphasize highly anisotropic regions. The color can be assigned with respect to eigenvalues.

The advantages of glyphs and tensor lines, which follow the eigenvector directions, are combined in hyperstreamlines. Hyperstreamlines are stream lines or stream tubes constructed from one eigenvector field; the other eigenvectors are encoded in the cross section. For further details the interested reader is referred to [26], as they are mainly used in 3D volumes.

Texture-Based Visualization Concepts

LIC was introduced by Cabral et al. [8]; it is a concept from vector field visualization, where the stream lines or flow from a vector field are displayed as a texture. The stream lines of a vector field and a filter kernel with a noise texture provide the input for the texture generation. The input texture is convolved with a filter kernel along a stream line of a vector field. The texture color at a pixel is computed by convolving the kernel, i.e. gray values of a noise texture, along the stream line. In so doing, the uncorrelated gray values of the filter kernel correlate in the direction of the vector field. The original algorithm has the drawback of redundant computation; however, the LIC algorithm was developed further into the FastLIC algorithm by Stalling and Hege [24]. The LIC algorithm can also be applied to tensor fields, the minor or major eigenvector field. The disadvantage of this method is that one can only display one eigenvector field, the minor or major vector field, in one single render pass. Furthermore, the anisotropy cannot be displayed.

The drawbacks of vector field visualization methods, particularly the LIC algorithm, can be avoided by the use of HyperLIC developed by Zheng and Pang [28]. HyperLIC depicts the direction of one eigenvector field as well as the anisotropy. Instead of computing a 1D convolution, a 2D convolution is performed. Thereby, the color value of a given pixel is computed by convolving the kernel along the trajectory of the major and minor eigenvector field. The size of the integration interval is determined by the ratio between the eigenvalues. Subsequently, the computed integral

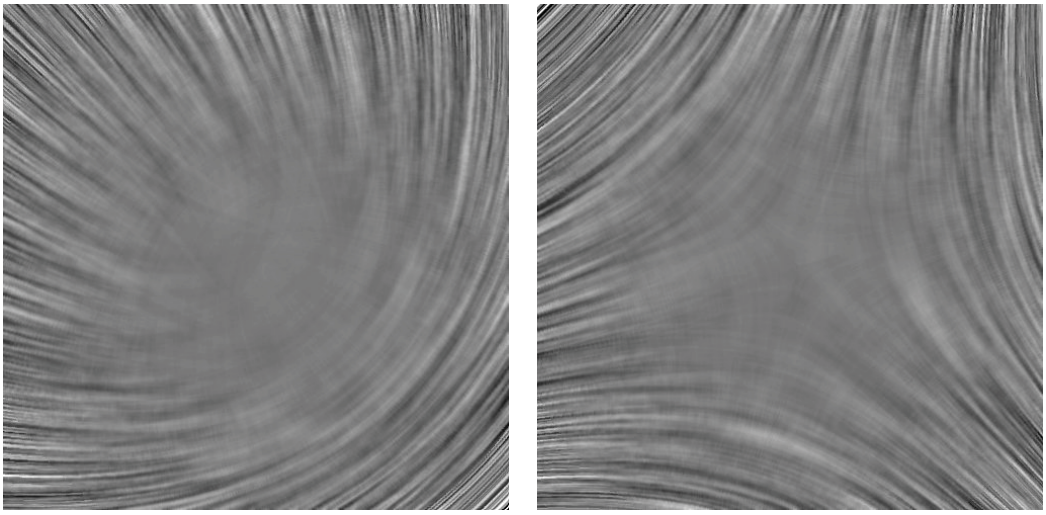


(a) (b)

Figure 1.12
LIC: a) wedgepoint b) trisector. Image courtesy Breßler.

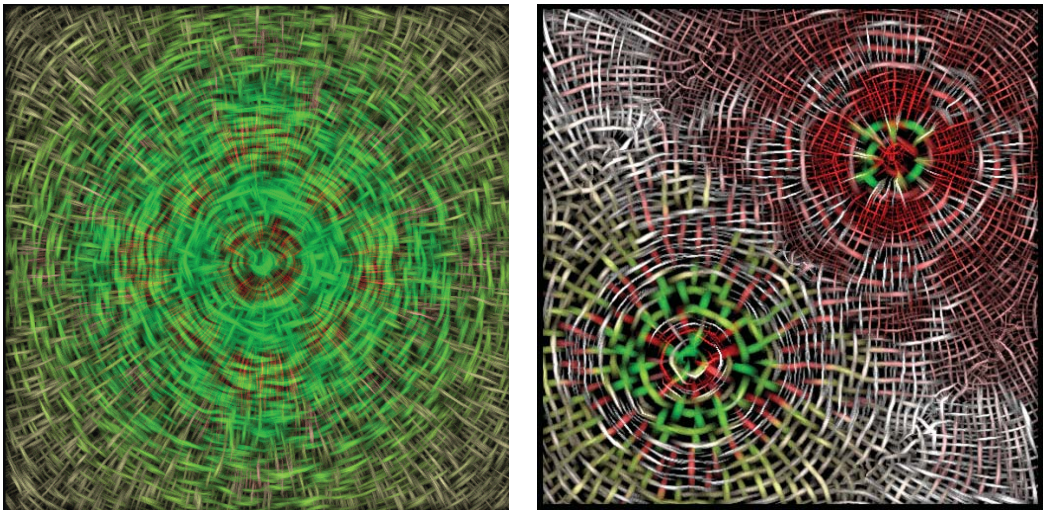
is arithmetically averaged. Anisotropic regions are convolved with a narrow kernel. These regions show similar results to those results computed by the LIC algorithm; they show one eigendirection. Isotropic regions, in contrast, are convolved with a broad or square kernel; these regions are blurred (see Figure 1.13). HyperLIC has the drawback that regions with opposite eigenvalue signs are illustrated as isotropic regions. In Amira a module for LIC and HyperLIC was implemented by Breßler [7].

A physically-based method for tensor field visualization was introduced by Hotz et al. [16]. This method provides a continuous representation of the tensor field and emphasizes the physical meaning of the tensor field, the regions of expansion and compression. Initially, a positive definite metric is defined, which preserves the topological structure of the tensor field. This metric maps the eigenvalues onto a restricted interval. Afterwards, for every eigenvector field an LIC-image is computed; however, instead of using white noise texture, a texture that resembles a piece of fabric is used. This kernel texture has several free parameters that encode properties of the metric. The texture is stretched and compressed according to the values resulting from the metric. Finally, the two LIC-images are overlaid. The resulting image (Figure 1.14) displays the eigendirections and, moreover, compression and expansion, which is strictly restricted to the sign of the eigenvalues. Animating the free parameters of the metric can enhance the impression of stretching and compression. For further details the interested reader is referred to [16].



(a) (b)

Figure 1.13
HyperLIC: a) wedgepoint b) trisector. Image courtesy Breßler.



(a) (b)

Figure 1.14
Physically-based methods for tensor field visualization: (a) one-point load and (b) two-point load. Image courtesy Hotz.

1.4.2 Related Methodological Work

A segmentation of tensor fields yields regions with similar tensor characteristics. The cells of the segmentation can be considered as planar glyphs of arbitrary shape; how-

ever, these cells can also be used for an improved glyph placement or as a basis for texture mapping.

Computing the Barycentroid

A natural method of segmentation-based glyph placement is to place the glyph in the center of the cell. For planar convex polygons the center of mass is inside the cell; however, for planar non-convex polygons the center of mass may lie outside the cell. To overcome this problem, Rustamov et al. [22] proposed an algorithm for computing the center of arbitrary shaped polygons, the *barycentroid*. This algorithm is based on a new metric, the *interior distance*. The *barycentroid* captures the semantic center of the shape is defined as the point for which the average *interior distance* is minimal. Further details of the algorithm and the implementation used in the framework presented here are given in Section 2.3.

Adaptive Texture Scaling

For the texture mapping, the parameterization is specified by the segmentation, and a texture is generated that encodes the characteristics of the given tensor cell, i.e. the eigendirections, the eigenvalues and anisotropy. Hummel et al. [18] developed an algorithm for texture generation that conveys the local direction of a vector field without distorting the texture. This texturing approach uses a single input texture or pattern and adjusts the sampling frequency. Thereby, anisotropic stretching of the texture coordinates can be compensated. Furthermore, the image-space pattern density of the surface is independently preserved from the size of the patches or cells. Further details of the algorithm and the implementation used in the framework presented here are given in Section 2.4.

1.5 A System for Visual Data Analysis – Amira

The visualization methods presented in this thesis have been developed as a module for the software Amira [25]. Amira is an interactive system for data analysis and visualization. Figure 1.15 shows the user interface of Amira. The user can load data and modules into the pool (left upper view) and make connections. In the view below the user can see and modify the settings of the module. The main view (right upper view) displays the result rendered by the selected visualization module. Additional information is displayed in the console (right lower view). Modules for Amira are implemented in C++, and can be extended with many third party libraries. In the modules presented here, the third party libraries MATLAB, OpenInventor, CGAL and OpenGL have been used.

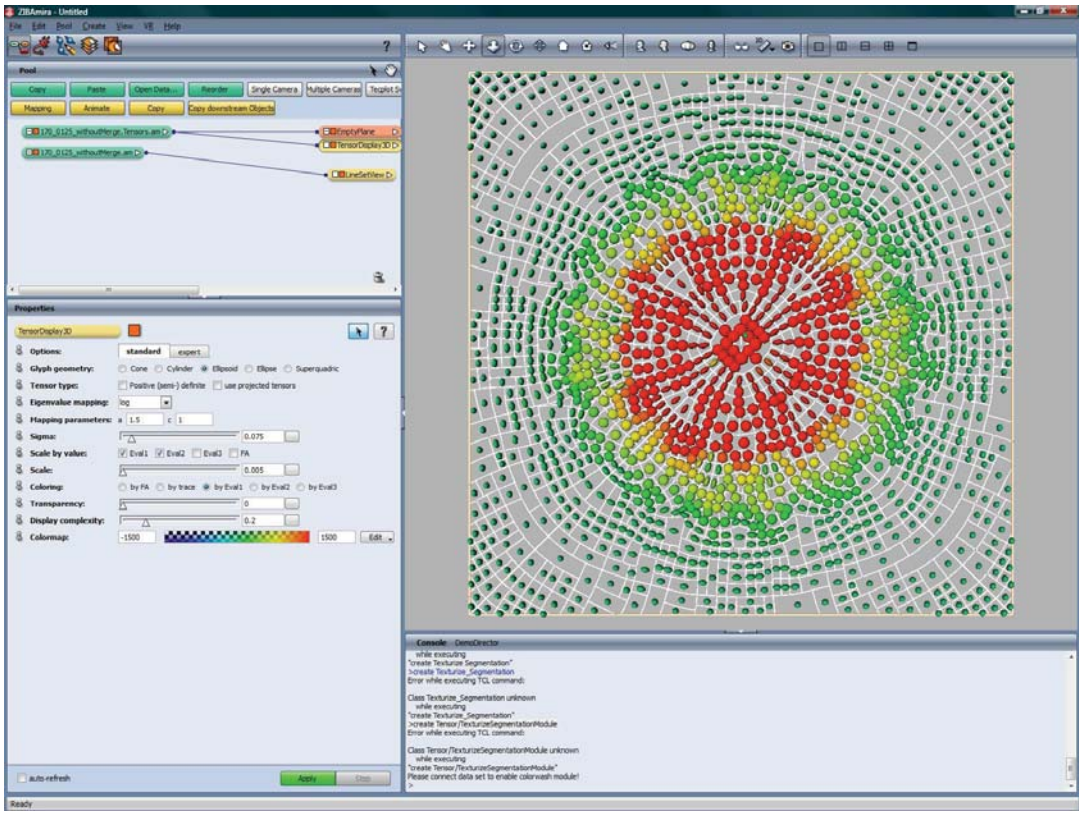


Figure 1.15
Highly interactive system for data analysis and visualization: Amira.

1.6 Preview

This chapter introduced the reader to the notation used in this thesis, the mathematical fundamentals of tensors and tensor fields, as well as to commonly known concepts of tensor field visualization and related works. Chapter 2 starts with a specification of the requirements and data structures and provides details of the implementation. These details are discussed in two sections: one for the geometry-based and one for the texture-based visualization approach. Each section concludes with a reflection of the constraints. In Chapter 3 the visualization methods implemented are analyzed and evaluated. At the beginning of Chapter 3, the datasets that have been used for the development and evaluation of the framework are described. Subsequently, the results are presented. In addition, the input texture design is discussed and the geometry-based and texture-based visualization approach are compared. The conclusion places these visualizations developed here in the context of related works, which have been introduced in this chapter. The last chapter ends with a preview of future work.

Chapter 2

Conceptual Details and Implementation

As introduced in the first chapter, the starting point of the developed visualization methods is a topological graph or a segmentation (see Figure 1.8 and Figure 1.11). The topological graph aggregates regions with qualitatively homogenous eigenvector behavior. The segmentation aggregates regions with qualitatively homogenous eigenvector and eigenvalue behavior. The cells of the topological graph or segmentation are passed to the visualization module in an Amira unique data structure. For the discrete, geometry-based approach, the center of each arbitrary shaped planar cell is computed. This is achieved according the algorithm developed by Rustamov et al. [22]. In doing so, choosing global parameters and requirements for all cells posed a challenge. For the piecewise continuous texture-based visualization method the algorithm presented by Hummel et al. [18] is applied to tensor fields. Furthermore, the algorithm is extended by additional parameters: the transformed eigenvalues. The transformed eigenvalues are used to visualize the physical behavior of the tensor field – regions of expansive and compressive forces. Since the original eigenvalues can exhibit very small negative and very large positive values, the eigenvalues must be transformed to a restricted positive interval. This is achieved by a metric defined by Hotz et al. [16]: eigenvalues below a determined threshold refer to compressive forces and eigenvalues above a determined threshold refer to expansive forces. Moreover, the framework presented here explores and evaluates the practice of different input texture pattern designs (see Section 3.2.3).

Initially in this chapter, the data structures, the requirements and the transformation of the eigenvalues, which is used in the implementations, are explained. Later the concepts and the details of the implemented visualizations are given (see Section 2.3 and 2.4). The data structure forms the interface between computing the segmentation and the visualization methods presented here. The requirements specify the properties our segmentation needs to fulfill. The conceptual idea of each visualization method is introduced at the beginning of the sections on the geometry-based and texture-based visualization. Thereafter, the key steps of the algorithms are ex-

plained in detail. The implementation summarizes these key steps. The sections of the geometry-based and texture-based visualization approach conclude with a discussion of constraints.

2.1 Data Structures and Requirements

The topological graph or segmentation of a two-dimensional slice of the 3D tensor field is computed with an Amira segmentation module developed by Auer et al. [23]. The output of the segmentation is a set of cells, which are bounded by tensor lines. Dependent on the specific segmentation criteria the size and the shape of the cells may be very different. Cells that are not adjacent to degenerate points or the domain boundary have four corner points and are bounded by two major and minor tensor lines segments. Cells adjacent to degenerate points can have more general shapes. The treatment of these cells is especially challenging because they do not follow a predefined pattern. The boundaries of the extracted cells, are stored in an Amira specific data structure and can be passed to other modules. Furthermore, the original tensor data for both visualization approaches is provided in a data structure. The texture-based visualization approach requires also some input patterns. More details are given later in Section 2.4, when the implementation of the texturization of segmented tensor fields is discussed.

2.1.1 Data Structure

A data structure provides the input for the geometry-based as well as the texture-based visualization module. This data structure stores the information of the topological graph or segmentation. The information stored is: an array with all vertices, the cell boundaries, and for each vertex an array with additional information. The cell boundaries are passed as poly-lines, where the data structure stores the indices of vertices for each poly-line. The first entry of the data array, which is attached for each vertex, stores the information about the vertex itself. Only one of the following can apply for a vertex (see Figure 2.3.2):

1. the vertex is a simple corner point (this implies so tensor line information is provided),
2. the vertex is a degenerate point (this implies the vertex is also a corner point),
3. the vertex lies on the boundary of the domain and no tensor line information is provided, or
4. the vertex is a point inside an edge and belongs to a minor or major tensor line.

The second entry of the data array stores information about the shape of the poly-line, namely the number of corner points. The third entry of the data array stores information about the tensor line which the vertex belongs to: a major or a minor

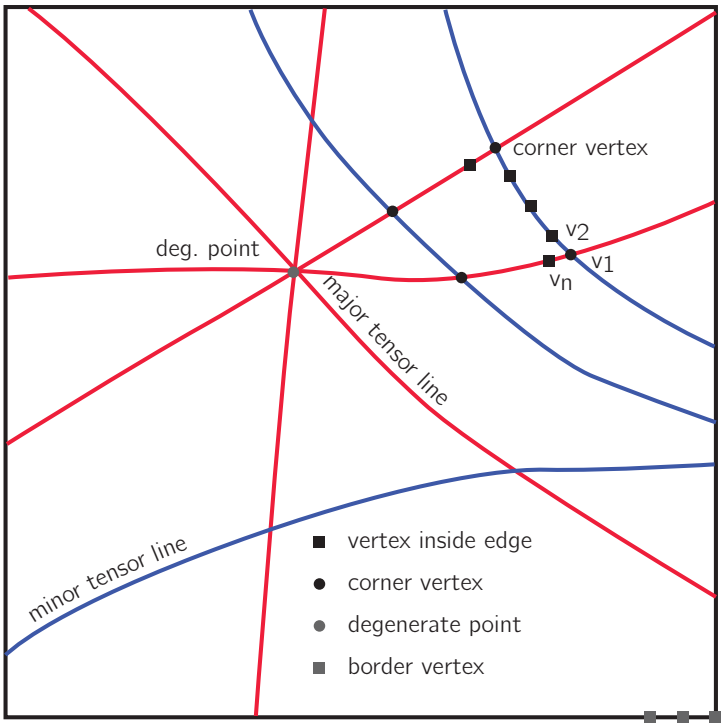


Figure 2.1
Outline of the cells of the segmentation: degenerate points, vertices, corners and tensor lines.

tensor line. These information are important for the texturization, as the input texture pattern has to be aligned according to the major and minor tensor lines and the texture coordinates are computed according to the shape of the cells.

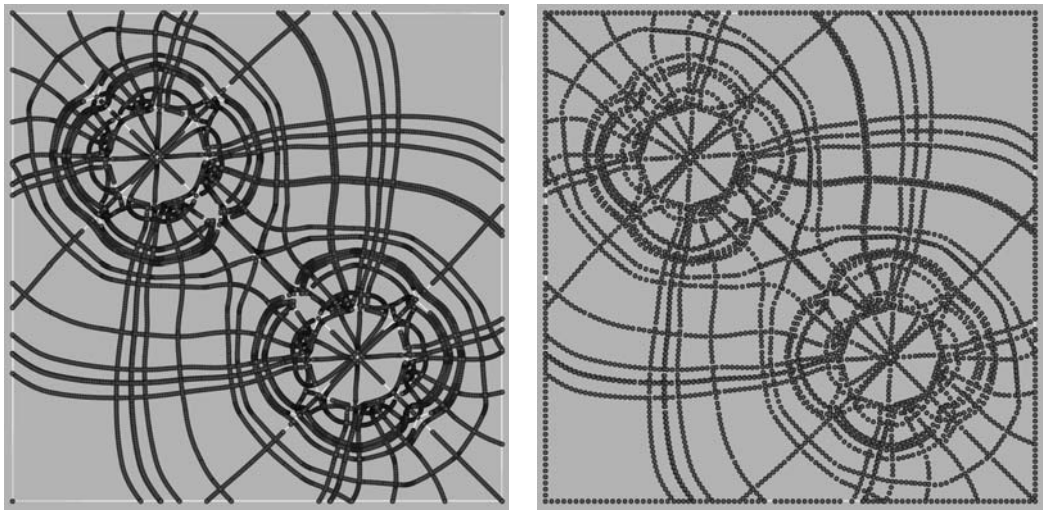
The data structure for the discrete tensor field, which is independent from the data structure of the segmentation, is based on a triangulated lattice. For each grid point a tensor is given. Within this lattice the tensor field can be reconstructed continuously by linear interpolation (see Section 1.2.4 for details of the tensor interpolation). For placement of glyphs in the topological graph or in the segmentation, this data structure is evaluated at the computed barycentroids. For the texture-based visualization approach, a tensor for each vertex is computed.

2.1.2 Requirements and Pre-processing

As introduced in Section 1.2.5, the tensor lines of the segmentation are computed by an integration scheme with an adapted step size. This leads to irregular distances between the vertices of the poly-lines; however, the results of both visualization approaches depend strongly on the sampling of the poly-lines. In particular, the topological

graph of the tensor field consists of arbitrary shaped polygons. In order to compute the center of the area of each polygon and to place a representative glyph inside the cell, the bounding poly-lines need to be finely and equidistantly sampled; furthermore, equidistantly and finely sampled poly-lines of the segmentation also improve the result of the texture mapping. In addition, double vertices are removed.

To obtain finely and equidistantly sampled cells a pre-processing step is performed. Vertices on a bounding poly-line are removed where they are too close to their neighbors and form an approximately straight line with the preceding and following vertices. This pruning of the poly-line is controlled by a distance threshold and an angle criterion. Conversely, new vertices are inserted if two neighboring vertices on a poly-line are too far away from each other. This is similarly controlled by a minimum distance threshold. Figure 2.2 shows the poly-lines of the boundaries before and after pre-processing. The pseudo-code for the pre-processing can be found in the Appendix A.1.



(a) (b)

Figure 2.2
Poly-lines of the topology graph: (a) before and (b) after the preprocessing.

2.2 Transformation of Eigenvalues

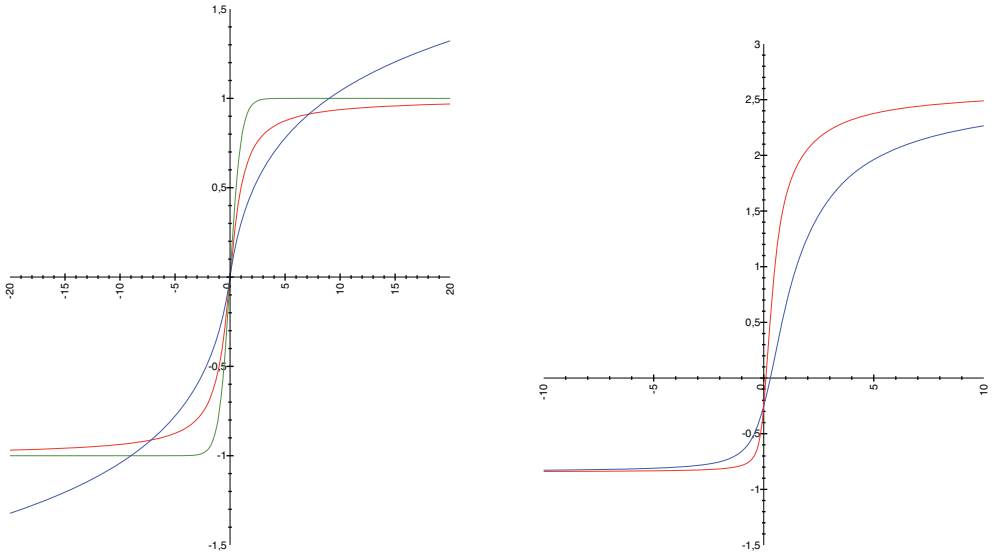
To review Section 1.2.4 regarding stress and strain tensor fields, positive eigenvalues correspond to tensile forces and negative eigenvalues correspond to compressive forces. Based on this observation Hotz et al. [16] have defined a metric which preserves the tensor field topology. This metric is used to transfer the eigenvalues into a restricted positive interval. This is required as the eigenvalues can exhibit very small negative and very large positive values. These transformed eigenvalues are used to encode the characteristic properties of the tensor data into the glyphs or textures.

2.2.1 The Metric

This section summarizes the eigenvalues transformation, as introduced by Hotz et al. [16]. The function F, which defines the metric, transfers "positive eigenvalues to eigenvalues greater than a and negative eigenvalues to eigenvalues smaller than a but larger than zero" [16]. Following the notation introduced by Hotz et al. [16], the transfer function can be defined by the scalars a, σ and the function f

$$F(\lambda) = a + \sigma \cdot f(\lambda) \ . \tag{2.1}$$

The region where the sign changes is of special interest. The function should have "a large slope in the neighborhood of zero" [16] (see Figure 2.3). The slope can be adjusted by a parameter. In the following, the functions f used in the texurization and geometry-based approach presented here are introduced.



(a) (b)

Figure 2.3
Functions f used in the metric: (a) arctangent (red), logarithmic (blue), hyperbolic tangent (green) and (b) non-symmetric function with different slopes $c = 3$ (red) and $c = 1$ (blue).

- "Anti-symmetric functions $f : f(-\lambda) = -f(\lambda)$" [16]:

  1. Logarithmic function

$$f(\lambda; c) = \begin{cases} \log(c \cdot \lambda + 1) \ , \text{for } \lambda \geq 0 \\ -\log(1 - c \cdot \lambda) \ , \text{for } \lambda < 0 \end{cases} \ . \tag{2.2}$$

The resulting metric should be positive definite. Therefore, $\sigma < a/\log(c \cdot \lambda_{max} + 1)$ is limited.

2. Asymptotic functions

$$f(\lambda; c) = \arctan(c \cdot \lambda) \ , \tag{2.3}$$

$$f(\lambda; c) = \tanh(c \cdot \lambda) \ , \tag{2.4}$$

where $\sigma < 2a/\pi$ (for Equation (2.3)) and $\sigma < a$ (for Equation (2.4)) are independent of λ_{max}.

- Hotz et al. have also defined an exponential function, which takes into account the "nonlinear perception of texture attributes" [16]. The function is defined as

$$F(\lambda) = a \cdot \exp(\sigma \cdot \arctan(c \cdot \lambda)) \ . \tag{2.5}$$

Here σ is not limited.

For all four examples, the constant $c \in \mathbb{R}$ adjusts the slope at the zero crossing. For high values of c, the function becomes steeper. $\sigma \in \mathbb{R}$ is the scale factor and $a \in \mathbb{R}$ is the offset. For further details on the properties of the metric, the interested reader is referred to [16].

2.3 Topology-based and Segmentation-based Glyph Placement

The goal of the topological-based and segmentation-based glyph placement is to find one local exponent which displays the characteristics of the extracted cell. As explained in the previous chapter the characteristics of the tensor field – the eigenvectors and eigenvalues of the major and minor eigenvector field – are similar inside each extracted cell. Hence, the stress and strain in the tensor field can be discretely visualized by representative glyphs. The density of the glyphs depends on the topological graph and the segmentation, respectively. The glyphs are displayed with an Amira glyph display tool.

2.3.1 The Algorithm

Since the algorithm for placing the glyph should work for the topological graph as well as the segmentation the center of area of each arbitrary shaped cell must be computed. The center of mass p_c of a polygon P can be defined as

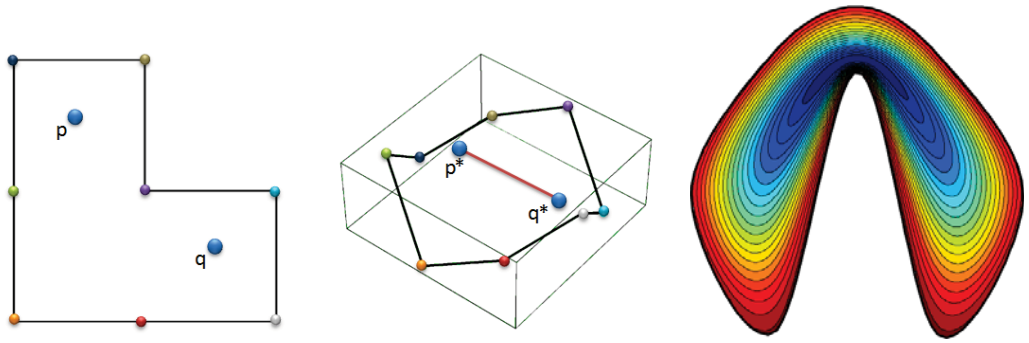
$$p_c = \frac{1}{A} \int \int_P \begin{pmatrix} x \\ y \end{pmatrix} dxdy \ , \tag{2.6}$$

where A is the area of the polygon. The center of mass of a polygon is the point where the average Euclidian distance to the boundary points is minimal. This center

of mass may lie outside the cell. An algorithm for computing the barycentroid of arbitrary shaped polygons was presented by Rustamov et al. [22]. This algorithm is based on an interior distance measurement. The barycentroid has the characteristic that it captures the semantic center of the polygon and lies inside the polygon. For convex polygons the barycentroid is equal to the center of mass. The barycentroid is the point where the average interior distance to the boundary points is minimal. For computing the barycentroid, the boundary vertices are mapped into high-dimensional space in such a way that the pairwise boundary distance is preserved, and computing the barycentroid is equal to minimize a potential with respect to p. See Figure 2.4(b) for a potential plot. The potential is defined by the following formula:

$$U^2(p) = \vec{w}(p)^T \mathbf{A} \vec{w}(p) \ , \tag{2.7}$$

whereas, $\vec{w}(p)$ are the weights of the barycentric coordinates of a given point p within the polygon and $\mathbf{A}$ denotes the *Gram matrix*[1] of the embedded vertices v_i^*. In simplified terms, the Gram matrix implicitly provides the distance information of the boundary vertices and the shape information of the polygon is provided by the barycentric weights.



(a) (b)

Figure 2.4
Visualization of the two key steps for computing the center of area of a polygon: (a) embedding of polygon into high-dimensional space and (b) visualization of isolines of $U(\cdot)$. From [22].

Embedding

For computing the Gram matrix, the boundary vertices are embedded into high-dimensional space in a way that the pairwise distances are preserved (see Figure 2.4(a)). This section outlines the embedding into high-dimensional space as developed by Rustamov et al. [22]. For convenience and clarity, the summary of the embedding follows partly literally the original publication, since this algorithm is not the main focus of this work. The diffusion distance [9] is such an embedding of the

[1]The *Gram matrix* contains the dot-product of the embedded vertices $a_{ij} = \langle v_i^*, v_j^* \rangle$

boundary vertices into high-dimensional space. The diffusion distance is computed using the Laplace-Beltrami operator of the form $\mathbf{M}^{-1}\mathbf{L}$, where $\mathbf{L}$ is the cotangent Laplacian of the polygon vertices and $\mathbf{M}$ the diagonal matrix that contains the "voronoi area" of the polygon vertices. To obtain the Laplace matrix $\mathbf{L}$ for two-dimensional polygons the stencil $[-1/d^-, 1/d^- + 1/d^+, -1/d^+]$ at each vertex is used [22]. To obtain the diagonal matrix $\mathbf{M}$, Rustamov defines the "voronoi area" of each vertex as $(d^+ + d^-)/2$, where d^+ and d^- are the distances to the two neighbors of the current vertex [22]. The Laplace matrix $\mathbf{L}$ and the diagonal matrix $\mathbf{M}$ with the "voronoi area" provide the input for the eigendecomposition of the generalized eigenvalue problem $\mathbf{L}\phi_k = \lambda_k \mathbf{M}\phi_k$.

The embedding can be computed according to the following formula:

$$v_i \mapsto (e^{-\lambda_1 t}\phi_1(v_i), e^{-\lambda_2 t}\phi_2(v_i), \ldots, e^{-\lambda_n t}\phi_n(v_i)) \in \mathbb{R}^n \ , \tag{2.8}$$

where n is the number of vertices, $\phi_k(v_i)$ is the i^{th} entry of the eigenvector ϕ_k and the squared diffusion distance is given by the squared Euclidian distance between the embedded vertices.

$$d^2(v_i, v_j) = \sum_{k=1}^{n} e^{-2\lambda_k t}(\phi_k(v_i) - \phi_k(v_j))^2 \ . \tag{2.9}$$

Our implementation follows the instruction of Rustamov such that the eigenvalues are ordered in non-decreasing order and the eigenvectors are normalized to have the unit M-norm with $\phi_k^T \mathbf{M}\phi_k = 1$.

Only a limited number of eigenvalues are necessary to approximate the exact embedding. The eigenvectors have to satisfy $e^{-\lambda_k t} > \epsilon$, where $\epsilon = 10^{-6}$ and the time scale parameter t is proportional to the first non-zero eigenvalue – the first eigenvalue should be zero – $(t = (8\lambda_2)^{-1})$ [11, 22]. Thus, the high dimensional space has m dimensions with $m < n$, where n is the number of vertices. For further details the interested reader is referred to Rustamov et al. [22].

Barycentric Coordinates of Arbitrary Shaped Polygons

For minimizing the potential, the column vector $\vec{w}(p) = (w_1(p), w_2(p), \ldots, w_n(p))$ with the weights of the normalized barycentric coordinates of a given point p with respect to an arbitrary shaped polygon is computed. This section summarizes the fundamentals of computing the barycentric coordinates of arbitrary shaped polygons and provides the details of the implementation. For convenience and clarity, the summary of the algorithm follows partly literally the original publication as presented by Hormann and Floater [15]. In particular, this holds for the notation.

Ψ is an arbitrary shaped planar polygon with $n \geq 3$ distinct vertices $v_1, \ldots, v_n$ and without intersecting or open edges. For $i = 1, \ldots, n$ the Euclidian distance between any vertex v and v_i is defined as $r_i(v) = \|v_i - v\|$ and $\alpha_i(v)$ denotes the

signed angle in the triangle $[v, v_i, v_{i+1}]$ at the vertex v. By implication, the *signed* areas of the triangles (see Figure 2.5) $[v, v_i, v_{i+1}]$ and $[v, v_{i-1}, v_{i+1}]$ are given by [15]

$$A_i(v) = r_i(v)r_{i+1}(v)sin(\alpha_i(v))/2 \tag{2.10}$$

$$B_i(v) = r_{i-1}(v)r_{i+1}(v)sin(\alpha_{i-1}(v) + \alpha_i(v))/2 \tag{2.11}$$

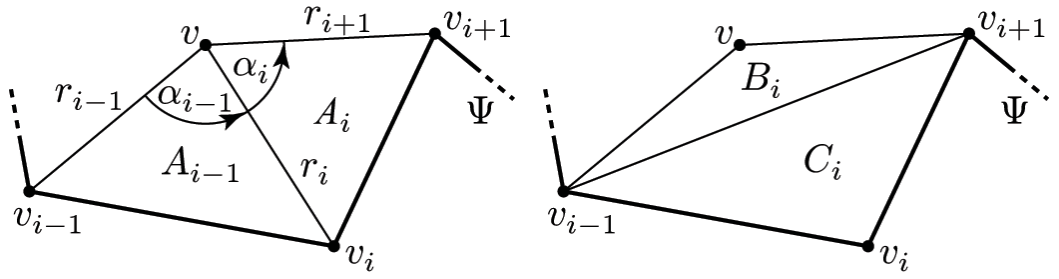


Figure 2.5
Notation used for angles, areas and distances, from [15].

$A_i(v), -B_i(v)$ and $A_{i-1}(v)$ are the homogenous barycentric coordinates of v with respect to the triangle $\triangle_i = [v_{i-1}, v_i, v_{i+1}]$. $B_i(v)$ becomes negative if v is inside the triangle and positive if v is outside the triangle $\triangle_i = [v_{i-1}, v_i, v_{i+1}]$; however, all barycentric coordinates must be positive for a point v inside triangle $\triangle_i$ and $B_i(v)$ must be negative for a point v outside triangle $\triangle_i$. Therefore, the sign of $B_i(v)$ must be flipped.

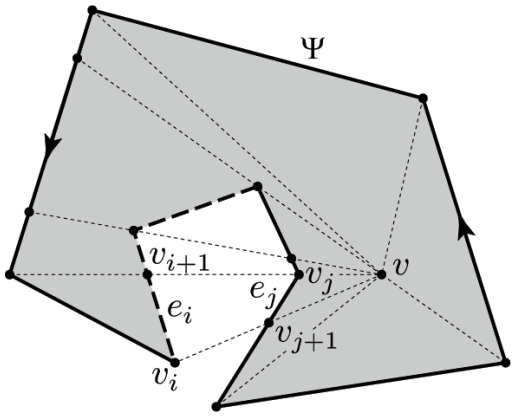


Figure 2.6
Partitioning of a polygon. From [15].

To generalize the homogenous coordinates of triangles to polygons, the polygon can be split into a triangle partitioning (see Figure 2.6) and the homogenous

coordinates of a vertex v with respect to all triangles $\triangle_1, \ldots, \triangle_n$ of the polygon are considered. For each vertex v_i three coordinates are given, one for each triangle $\triangle_{i-1}, \triangle_i$ and $\triangle_{i+1}$. Hormann defines

$$w_i(v) = b_{i-1}(v)A_i(v) - b_i(v)B_i(v) + b_{i+1}(v)A_{i-1}(v) \tag{2.12}$$

where $b_i : \mathbb{R}^2 \to \mathbb{R}$ is an arbitrarily chosen weight function and, moreover, he shows that the functions w_i are homogenous barycentric coordinates. The homogenous barycentric coordinates are normalized by the weight

$$b_i = \frac{r_i}{A_{i-1}A_i} \ .$$

In our framework this particular choice of b_i is implemented. This leads to the equation

$$w_i(v) = \frac{r_{i-1}(v)A_i(v) - r_i(v)B_i(v) + r_{i+1}(v)A_{i-1}(v)}{A_{i-1}A_i} \ . \tag{2.13}$$

For further details on the mathematical properties of the barycentric coordinates the interested reader is referred to Hormann and Floater [15]. Now, computing the barycentric weights of a given point p with respect to the polygon with $n \geq 3$ distinct vertices $v_1, \ldots, v_n$ is straightforward and follows the pseudo-code, see Algorithm 3 in Appendix A.2.

The Gradient descent

The barycentric weights are not defined for points on the boundaries of the poly-lines, since for these points the denominator in Equation (2.13) becomes zero. Therefore, an initial point inside the polygon has to be found for the gradient decent. The initial point, the Gram matrix and the vertices provide the input for minimizing the potential with respect to p. For the evaluation of the potential, the barycentric weights of each point p are computed. This step is repeated until the difference between the newly computed potential and the potential of the previous step is below a given tolerance threshold or a maximal number of updates is reached. Otherwise, p is updated until the potential reaches a minimum.

2.3.2 Implementation

The reader should be familiar by now with the requirements for computing the barycentroid (Algorithm 4 in Appendix A.2 shows the pseudo-code). Particularly, this means computing the Gram matrix of the embedded vertices and the barycentric weights in order to minimize the potential. In the following a step-by-step illustration of our implementation is given. This summarizes the embedding.

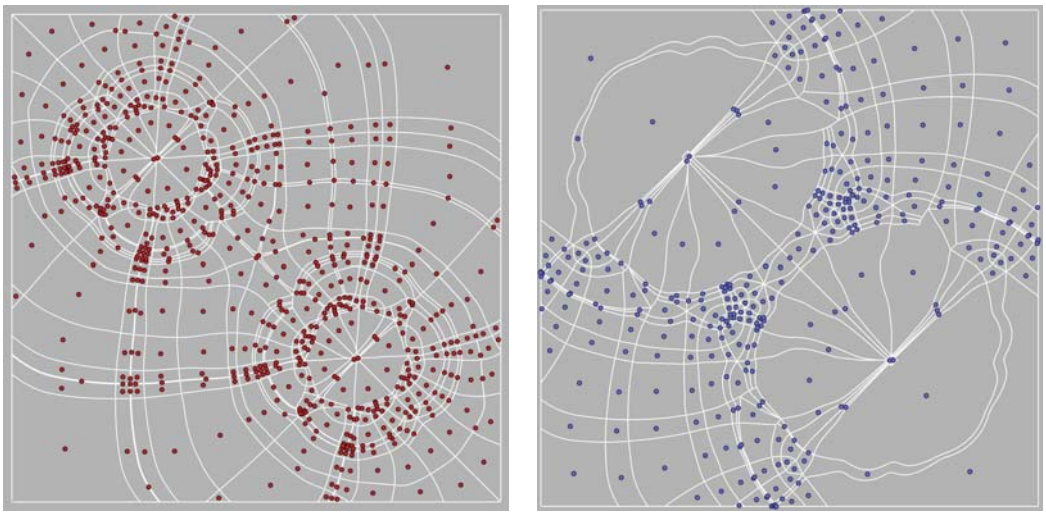
Since computing the Gram matrix and minimizing the potential is expensive, each polygon is tested to see if it is convex or non-convex. If the polygon is convex, the

barycentroid is equal to the center of mass and only the center of mass is computed; this can be approximated by the following formula:

$$p = \frac{\sum_{i=1}^{n} v_i}{n} \ , \tag{2.14}$$

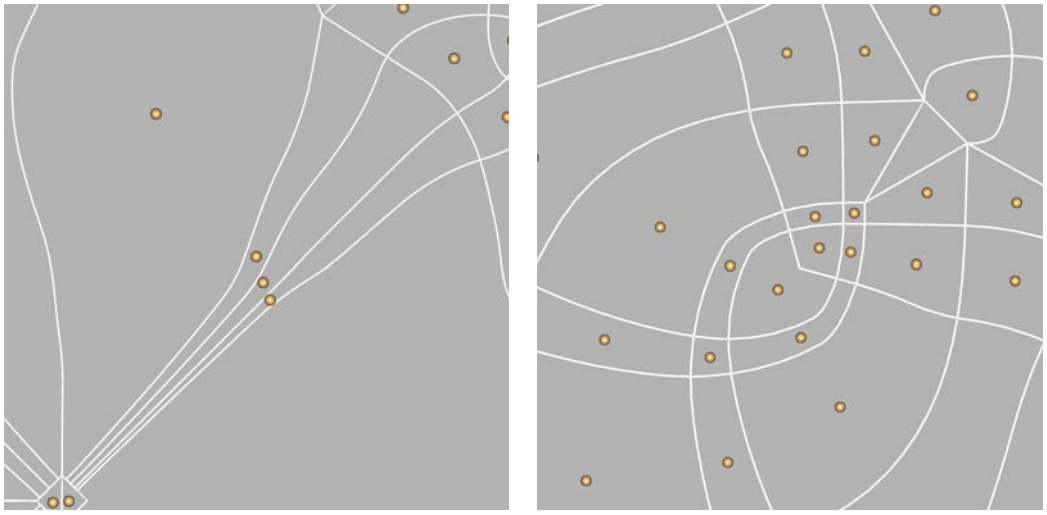
where $v_1, \ldots, v_n$ are the vertices of the polygon. Since the boundaries are finely and equidistantly sampled this is reasonable accurate. If the polygon is non-convex, the following steps are performed. The Laplacian matrix $\mathbf{L}$ and the matrix with the "voronoi area" $\mathbf{M}$ are computed. These matrices provide the input for computing the embedded vertices. For the embedding, the eigenvalues and eigenvectors are then computed by solving the general eigenvalue problem $\mathbf{L}\phi_k = \lambda_k \mathbf{M}\phi_k$ with MATLAB. As suggested by Rustamov, the resulting eigenvectors are normalized to unit M-norm. Then, the embedded vertices are computed according to formula 2.8. For the Gram matrix $\mathbf{A}$, the dot-product of the embedded vertices is computed. Then the potential (see Equation (2.7)) with the Gram matrix is minimized with respect to p. For the gradient descent, the barycentric weights of each point p is computed and the potential is evaluated until it reaches a minimum (see Section 2.3.1). Figure 2.3.2 shows the topological graph with the barycentroids and Figure 2.3.2 shows a close-up of characteristic cells.

After the barycentroids are computed, the glyphs are placed at the corresponding point. The glyphs are rotated into the coordinate system defined by the eigenvectors and the axes are scaled according to the mapped eigenvalues and colored by the selected color scheme. The results are shown in Section 3.2.1.



(a) (b)

Figure 2.7
Topological graph with barycentroids based on two different slices of the two-point load: (a) slice 2 and (b) slice 1.



(a) (b)

Figure 2.8
Characteristic cells.

2.3.3 Parameters

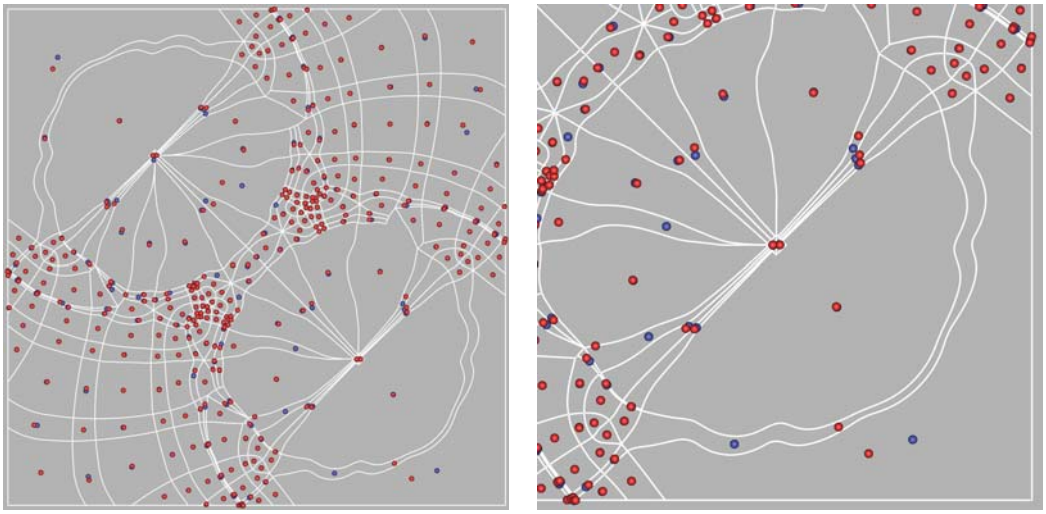
This section presents some examples of computing the barycentroids for the cells of the topological graph as well as for the segmentation. Particularly, experiences implementing the algorithm and the impact of the requirements and different parameters are discussed: the quality of the polygon sampling and the parameters chosen for the gradient descent. Thereby, some requirements and parameters have a greater effect on the result, whereas the influence of other parameters and requirements on the result may be neglected, as they do not discernibly influence the result. Furthermore, choosing global parameters and requirements for all cells posed a challenge. Some parameter and requirements give very good results for a single cell; however, they yield poor result for other cells.

Sampling

It was found that the sampling of the boundaries is crucial. The boundaries of the polygons must be finely and equidistantly sampled. It is not enough to provide solely the corner information for a polygon. Long edges have to be sampled and vertices have to be added. Conversely, vertices that are too close to each other have to be removed. Closely spaced vertices make the computation of the Laplace operator inaccurate. Further details are discussed in Section 2.3.4.

Figure 2.9 shows the barycentroids placed in the topological graph with two different pre-processing parameters (see Algorithm 1 and 2 in Appendix A.1). The pre-processing of the boundaries for the red spheres is performed with an angle threshold parameter of 175 ° and a sampling distance parameter of 0.5 %. The pre-processing

of the boundaries for the blue spheres is performed with an angle threshold parameter of 165 ° and a sampling distance parameter of 1.25 %. The percentages are based on the maximum side length of the domain.



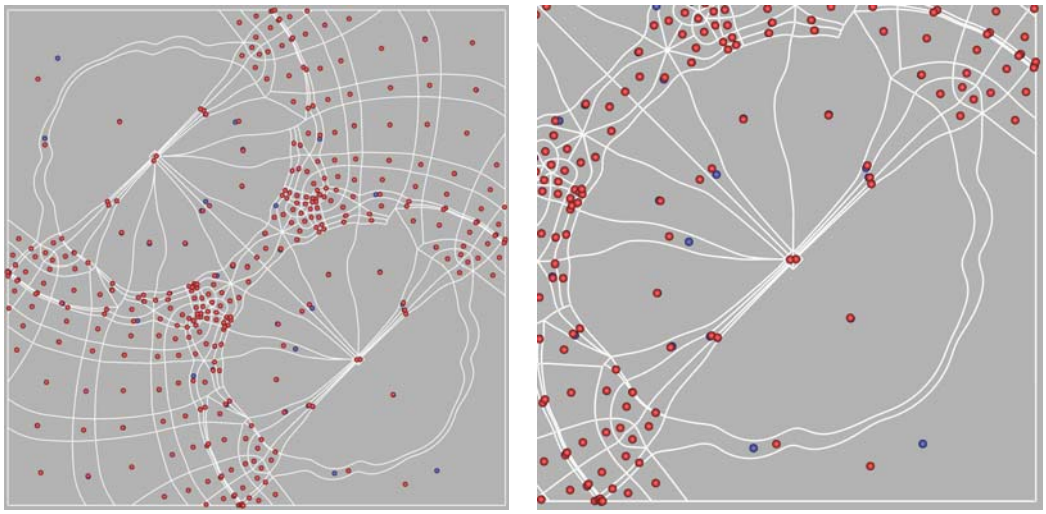
(a) (b)

Figure 2.9
Topological graph with barycentroids, computed with different boundary sampling parameters: dense sampling (red spheres), sparse sampling (blue spheres)

Gradient Descent

For the gradient descent, five parameters may be adjusted. Most of the parameters can be neglected because they do not perceptibly influence the result; however, the initial point and the *initial explore step* are crucial in regard to the quality of the result. The *initial explore step* is the initial step width for the gradient descent. This step width is gradually scaled down until it reaches the *final explore step*. As the initial point for the gradient descent, the center of mass may be chosen for most cells. If the center of mass is outside the polygon, a heuristic is used to find an initial point inside the polygon. For the heuristic used in the visualization framework, the poly-lines are iteratively scaled down in a way that the center of mass is only computed for a part of the polygon area. This is repeated until the newly computed center of mass is inside the polygon and an initial point for the gradient descent is found. The heuristic performs well for the poly-lines of our segmentation as well as the topological graph; however, the heuristic may not work in general for all arbitrary shaped polygons. The *initial explore step* is critical for larger cells. A value of 10 % gives good results for most cells (see Figure 2.7(b)). Figure 2.10 shows a comparison of both results. The remaining parameters are less critical. For completeness, the remaining parameters are set to the following values: the *final explore step* is set to 0.01 %, the tolerance

threshold is set to $1e{-}04\,\%$ and the maximal number of updates is set to 1000. The percentages are based on the maximum side length of the domain.



(a) (b)

Figure 2.10
Topological graph with barycentroids, computed with different gradient descent parameters: *intial explore step* $1\,\%$(red spheres) and *intial explore step* $10\,\%$.(blue spheres)

2.3.4 Constraints

As discussed in the previous section, the parameters of the gradient descent as well as the sampling of the polygon boundaries are crucial for computing the potential and the barycentroid. Pre-processing for the potential plots discussed in this section was performed with an angle threshold parameter of $165\,^\circ$ and a sampling distance parameter of $1.25\,\%$ of the maximum side length of the domain.

Numerical Inaccuracies

Figure 2.11 shows the potential of three cells. The one-dimensional Laplace operator provides the distance information for the polygon. The shape information is supplied by the barycentric weights. The potential decreases rapidly from the boundary to the center; however, this potential shows an extended area of minimal values (see Figure 2.11(a)). This makes the computation of the potential vulnerable to numerical inaccuracies that may occur while computing the Gram matrix. To review Section 2.3.1, the eigenvalues are ordered in an ascending order and the first eigenvalue should be zero. A measurement of these inaccuracies can be defined by the mean squared difference between the first eigenvalue and zero. The mean squared error increases if the sampling of the boundaries exposes irregularities. For computing

the Laplace matrix the inverse distance to the neighboring vertices is used. Therefore, computing the Laplace matrix becomes unstable if the distance between two vertices is too small. This is observable in outliers in the potential.

(a) (b) (c)

Figure 2.11
Potential plot of different cells. Red refers to large values and blue refers to small values of the potential. The potential shows an extended area where the values are small.

2.4 Texturization of Segmented Tensor Fields

As introduced in Chapter 1, the texturization of tensor fields is based on a given segmentation of stress and strain tensor fields. The tensor data and two input patterns (one for the major tensor field and one for the minor tensor field) are required. In most of the visualization examples discussed in Chapter 3, directional input texture patterns are used, where an input pattern with vertical stripes corresponds to the major eigenvector field and an input pattern with horizontal stripes corresponds to the minor eigenvector field. This has the advantage that the (s, t) unit square texture coordinates are only computed once, as the major and minor eigenvector fields are orthogonal to each other. The input texture pattern exerts an important influence on the resulting visualization. In particular, the frequency and direction of the input pattern is crucial in regard to the information that is perceptible in the visualization. Some algorithms only allow low frequency input patterns, whereas other algorithms demand input patterns with a higher frequency. The texture design is discussed in Section 3.2.3. A chosen pattern texture may be mapped onto the cells of the segmentation in such a way that the expansive and compressive forces encoded by the tensor are visualized. This is achived by scaling the pattern according to the transformed eigenvalues. By default, both eigenvector fields are superimposed; however the user has the option to observe the eigenvector fields separately, apply additional color information and perform a post-processing filter.

2.4.1 Fundamentals

This section provides the reader with the methodological fundamentals of the texture-based visualization approach: the basics of the OpenGL Shading Language (GLSL)

and texture mapping. Texture mapping can be either performed in the OpenGL fixed function pipeline or with OpenGL Shading Language (GLSL). To keep the code consistent, the framework presented here uses only GLSL.

OpenGL Shading Language

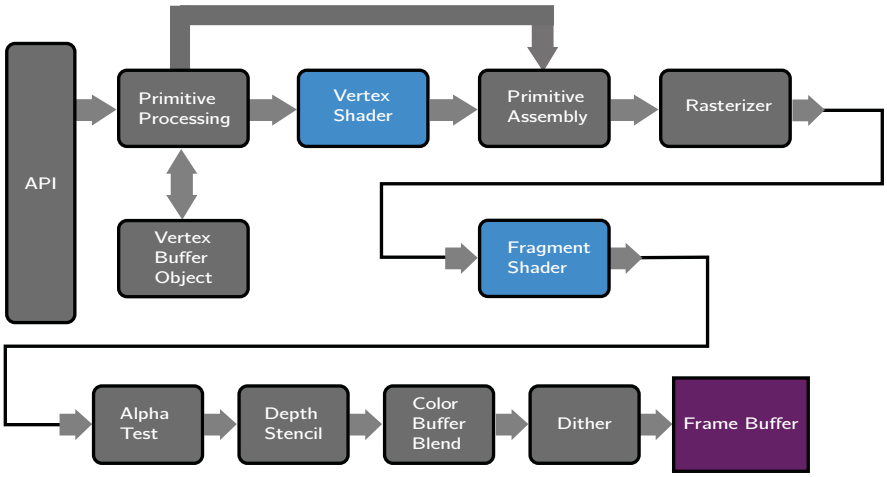


Figure 2.12
OpenGL programmable render pipeline

GLSL is a high level C/C++ like programming language for short programs on the graphic card. Figure 2.12 shows a simplified diagram of the programmable OpenGL pipeline. This render pipeline is explained according to the publication presented by Marroquim and Maximo [20]. The graphics pipeline is responsible for transforming geometric primitives into a raster image. The primitives are usually described as a set of triangles with connected points, the primitive vertices. They undergo transformations which map their object coordinates into the camera's coordinate system. The vertex shader operates on these transformed vertices. It is executed by the call glDrawArrays. If a vertex shader is used, all per-vertex operations of the OpenGL fixed function pipeline will be replaced. Then, the primitive vertices are assembled according to their connectivity. The rasterizer computes the pixels of the image. For each pixel, one fragment is created. Each fragment has interpolated attributes from its primitive vertices such as color, normals and texture coordinates. The fragment shader operates on these fragments, which are produced by the rasterization. If a fragment shader is used, all per-fragment operations of the OpenGL fixed function pipeline will be replaced. At the end of this render pipeline, different tests are performed (alpha test, depth test, etc.) before the image is rendered into the frame buffer. The frame buffer is a memory buffer that provides the color information of every pixel on the screen. The content of the frame buffer can

be either directly rendered to the screen or saved in an OpenGL frame buffer object. This content of the frame buffer object can be used as a texture for further render stages. In the texturization framework presented here, frame buffer objects are used for a post-processing filter. For further details the interested reader is referred to [20].

GLSL supports five different build-in types: int, float, double, bool and sampler. For the first four types, vectors are available. Different types of matrices exist for floats and doubles. Samplers represent textures. For a full list of build-in types and GLSL build-in functions see [19]. Code B.1 in Appendix B shows a simple vertex shader and a fragment shader for texture mapping.

Texture Mapping

In 3D graphics texture mapping is a well-known technique. Texture mapping is the process that maps a geometric point in space, i.e. a point on a surface to a color value in the texture domain. Two steps are performed (see Figure 2.13):

1. the point on the geometry is mapped on the unit square and
2. the point in the unit square is mapped to a texture of arbitrary size.

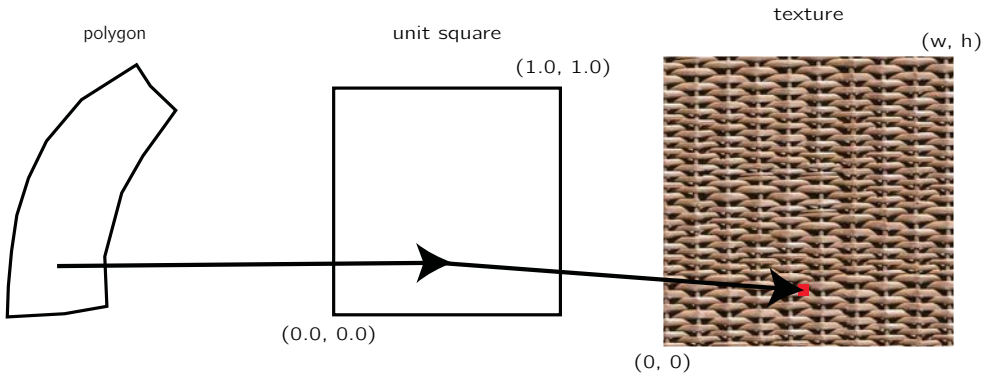


Figure 2.13
Texture mapping

For texture mapping the cells are triangulated. This triangulation provides the polygonal surface for the texture mapping. For each polygonal surface the (s, t) unit square texture coordinates are computed and specified per vertex. The texture coordinates are linearly interpolated across the fragments, which are produced by the rasterizer. Thereafter, a point (s, t) on the unit square corresponds to a point on the texture (which is of size (w, h)), so that the color value is the texture pixel

$(s \cdot w, t \cdot h)$. Since the continuous texture coordinates do not always conform to an unambiguous discrete point of the texture, interpolation can be performed. The interpolation method can be set in OpenGL (GL_NEAREST, GL_LINEAR, etc.). The texture coordinates are not restricted to the interval $[0, 1]$. If the texture coordinates exceed the interval $[0, 1]$, the wrap parameter for texture mapping can be set in OpenGL (GL_CLAMP, GL_REPEAT, etc.). For the framework developed here, the wrap parameter is set to GL_REPEAT. This ignores the integer part of the texture coordinates. Only the fractional part is used, which creates a repeating pattern.

2.4.2 The Algorithm

Roughly, the shading of the segmentation is based on two different approaches. The first approach encodes the pattern repetition in the texture coordinates. The texture coordinates are scaled by a factor. This factor is computed by the side length and the average of the eigenvalues for the cell. The second approach extends the pattern scaling algorithm presented by Hummel et al. [18]. This is achieved by adding two parameters to the algorithm, namely the transformed eigenvalues. The input pattern can be additionally scaled by the metric. Therefore, the transformed eigenvalues are passed to the shader for each vertex of the triangulated cell and are interpolated for each fragment produced by the rasterizer. In addition to the existing texture color, further color information can by applied. This makes it possible to direct the focus of the observer to different features of the tensor field. At the end of the render pipeline, an optional post-processing filter can be applied. The post-processing filter blurs the textured tensor field by the fractional anisotropy (see Equation (1.6)) or shear stress (see Equation (1.8)).

Computation of the Texture Coordinates

As explained in Section 1.3.2, the given cells of the segmentation exhibit similarities: most of the cells are triangular or rectangular in shape. In the vicinity of degenerate points more complicated structures are possible. In these regions the eigenvector behavior may expose irregularities. These cells are either not textured or just textured with a noise pattern to provide the color information.

The segmentation of the tensor field is computed with an Amira segmentation module and passed to the visualization framework in an Amira specific data structure. Due to dangling nodes during the segmentation, some mis-classifications can occur: regular cells, which are not located in the neighborhood of degenerate points, are classified as pentagons or hexagons. The algorithm for computing the texture coordinates has to intercept these mis-classifications.

To review Section 2.1.1: the data structure provides the vertices of each polyline and further information for each vertex. This information is available for the texture mapping. The input texture pattern must be parallel aligned according to the tensor lines. This can be done by different mappings of the boundary vertices to the unit square texture coordinates. Rectangular polygons are bounded by four

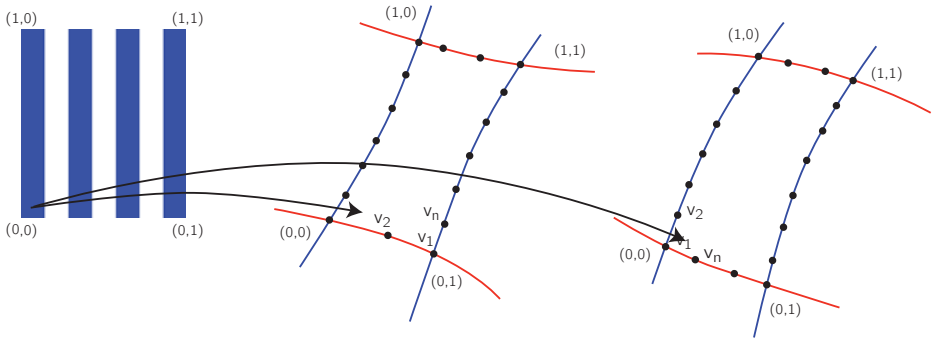


Figure 2.14
Computation of the texture coordinates: two rectangular cells with vertices $v_1, \ldots, v_n$ and (s,t) texture coordinates. One cell starts with a major tensor line and the other cell starts with a minor tensor line.

alternating edges. A polygon can either start with an edge belonging to a minor or to a major tensor line. There are two possible classifications [2]:

1. 0101 (the first edge of the rectangular cell belongs to the minor tensor field) or
2. 1010 (the first edge of the rectangular cell belongs to the major tensor field).

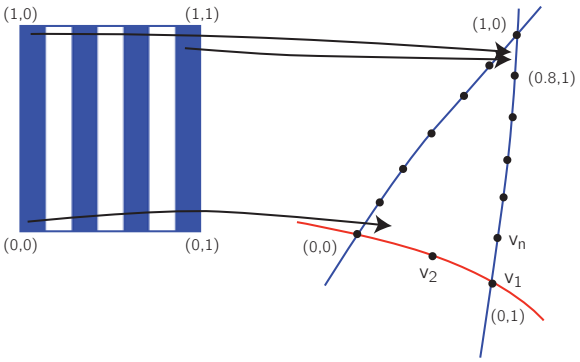


Figure 2.15
Texture mapping for triangle: triangle with code 100.

Triangles are bounded by three edges. They exhibit more classification possibilities:

- triangles with two edges belonging to minor tensor lines and one edge belonging to a major tensor line:

[2] "0" represents a minor tensor line and "1" represents a major tensor line.

1. 100 (the first edge of the triangular cell belongs to the major tensor field),
2. 010 (the first edge of the triangular cell belongs to the minor tensor field) or
3. 001 (the first edge of the triangular cell belongs to the minor tensor field).

- triangles with two edges belonging to major tensor lines and one edge belonging to a minor tensor line:

1. 011 (the first edge of the triangular cell belongs to the minor tensor field),
2. 101 (the first edge of the triangular cell belongs to the major tensor field) or
3. 110 (the first edge of the triangular cell belongs to the major tensor field).

The texture coordinates for one edge are then computed, according to the classification code, as the ratio between the total edge length and the integrated edge length for vertex v_i of the current edge. Figure 2.14 depicts the texture mapping for two rectangular cells and Figure 2.15 depicts the texture mapping for a triangle.

Falsely classified pentagons and hexagons are converted to rectangles. For vertices that are neither degenerate points nor corner points, unique tensor line information is provided and stored as color information. If one corner point lies between two such interior edge vertices, the corner point is retained if the color of the tensor line changes. Otherwise, the corner point is discarded and converted to an interior edge vertex. If the corner point lies between one or two other corner points (this can happen if the boundaries a sparsely sampled), the tensor line information for the preceding and following edges is taken into account.

Scaling of the Texture Coordinates

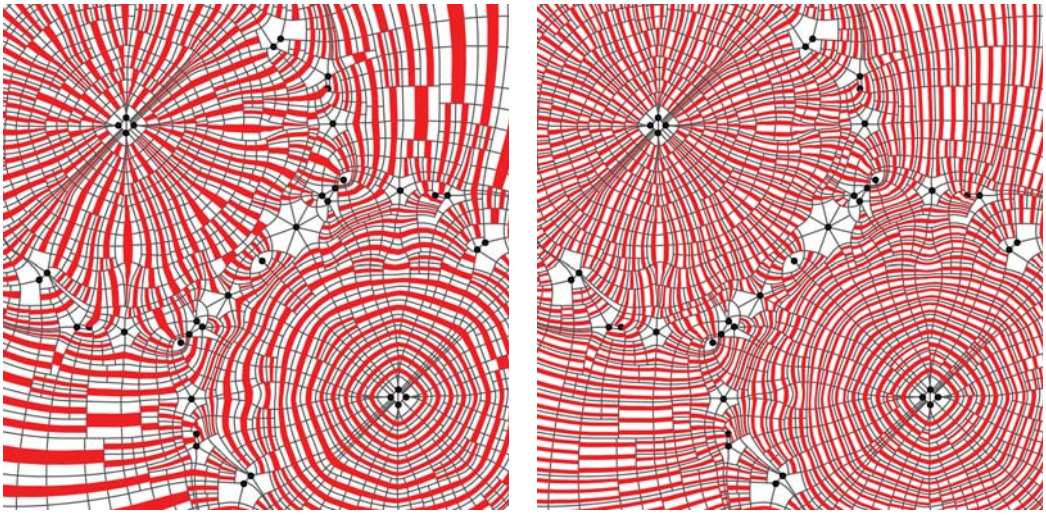
The user can select different visualization methods. According to the selected method the texturization framework makes use of different kinds of texture coordinates:

1. (s, t) unit square texture coordinates. Figure 2.16(a) shows the texture mapping with unit square texture coordinates.

2. The (s, t) unit square texture coordinates are multiplied by the factors n and m: $(s \cdot n, t \cdot m)$. The factors n and m depend on a global user determined minimum side length threshold. The lengths of opposing edges are averaged and the factors n and m for the texture coordinates are computed by rounding the fractions:

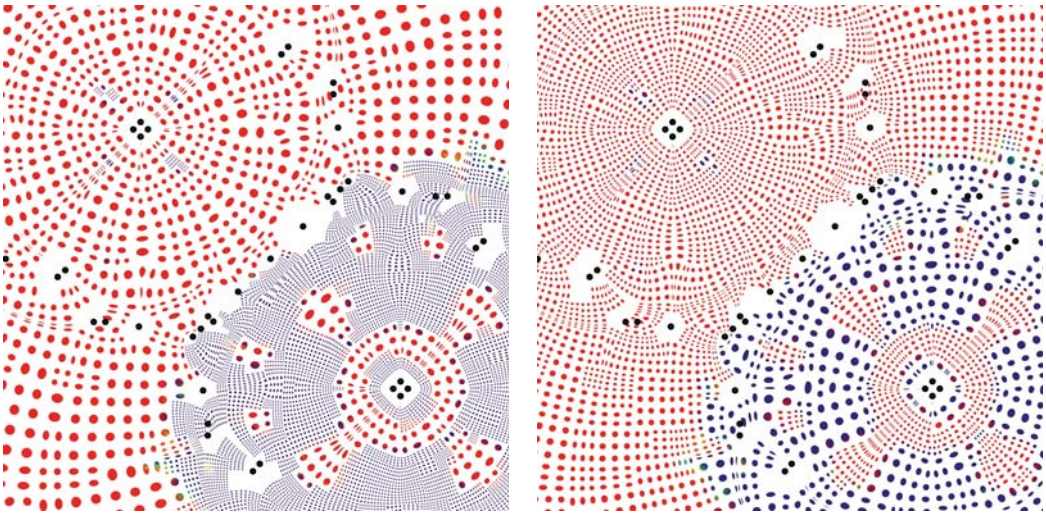
$$n = \lceil averageEdgeLength1/minSideLength \rceil \text{ and} \tag{2.15}$$

$$m = \lceil averageEdgeLength2/minSideLength \rceil \ . \tag{2.16}$$



(a) (b)

Figure 2.16
Major eigenvector field with texture mapping: (a) with unit square texture coordinates and (b) with scaled texture coordinates.



(a) (b)

Figure 2.17
The major eigenvector field is texturized with texture coordinates that are scaled by the eigenvalues: the pattern repetition (a) visualizes the major eigenvalues and (b) the metric of the major eigenvector field.

Here the texture coordinates exceed the values of the interval $[0, 1]$. The pattern is repeated and the cells are tiled by the pattern. This method can display

the underlying tensor lines, without the information of the eigenvalues. Figure 2.16(b) shows the texture mapping with scaled texture coordinates.

3. The (s,t) unit square texture coordinates are multiplied by two scale factors $scaleFactor1$ and $scaleFactor2$ ($s \cdot scaleFactor1, t \cdot scaleFactor2$). This time the scale factors depend on the factors n and m (see above), a user determined threshold for the maximum number of pattern repetitions and the average of the mapped eigenvalues, which are given for each vertex of the polygon:

$$scaleFactor1 = n + \lfloor maxNumPattern - \tilde{\theta}_1 \rfloor \text{ and} \tag{2.17}$$

$$scaleFactor2 = m + \lfloor maxNumPattern - \tilde{\theta}_2 \rfloor \ , \tag{2.18}$$

where $\tilde{\theta}_1$ is the average of the major transformed eigenvalues of the polygon, $\tilde{\theta}_2$ is the average of the minor transformed eigenvalues of the polygon and $maxNumPattern \geq max(\tilde{\theta}_1, \tilde{\theta}_2)$ is the maximum number of pattern repetitions. Again, the texture coordinates exceed the values of the interval $[0,1]$ and the pattern is repeated. This method is used to display the information of the compressive and expansive forces. Thus the physical behavior in the tensor field is visualized. For compressive forces (the transfer function maps the originally negative eigenvalues to small values of the metric) the pattern is scaled down and repeated often. Expansive forces are encoded in larger patterns (see Figure 2.17(a)). The user also has the option of visualizing the metric (see Figure 2.17(b)). In contrast to the visualized physical behavior, small eigenvalues are visualized in larger patterns with less pattern repetitions. This is achieved by computing the scale factors with the following formulas:

$$scaleFactor1 = n + \lfloor \tilde{\theta}_1 \rfloor \text{ and} \tag{2.19}$$

$$scaleFactor2 = m + \lfloor \tilde{\theta}_2 \rfloor \ . \tag{2.20}$$

The texture mapping with these different kinds of texture coordinates exhibits unsteady transitions between the cells. This problem is reasonable since the focus of this work is to visualize the characteristics of the tensor field: the direction of the eigenvector fields, compressive and expansive forces. Nevertheless, this drawback is discussed in the next chapter.

Shaders

The framework allows the user to select a shader. By default both eigenvector fields are displayed; however, the user has the option of fading out the eigenvector fields with a corresponding blend factor. The blend factors range from 0.0 to 1.0. By default they are set to 1.0. These blend factors enable the user to observe the eigenvector fields separately. In the following, details of the implemented shaders are presented:

1. The code of the first GLSL shader – in the following this shader is referred to as the default shader – extends the code B.1 in the appendix by an additional texture, color information for the two eigenvector fields and one blend factor for each eigenvector field. The first input texture is provided for the major eigenvector field; the second input texture is provided for the minor eigenvector field. For each eigenvector field color information is computed: the rgb-color values of the texture pixels are multiplied with the corresponding rgb-color values of the selected color mapping as explained in the next section. The alpha value of the textures is multiplied with the corresponding blend factor. Finally, the color information for the two eigenvector fields is linearly blended. The code B.3 for this shader can be found in Appendix B.1. The default shader is used for texture mapping with the scaled texture coordinates (see Figure 3.5(b)).

2. The second shader is based on the publication presented by Hummel et al. [18]. In the following, the implementation is briefly explained. For a pattern "$P(s,t)$ over the unit square", two "resolution levels l_s and l_t are computed" [18] according to the following formulas:

$$l_s = \log_2 \tau_s \text{ and } l_t = \log_2 \tau_t \ , \tag{2.21}$$

where τ_s or τ_t is "the variation in texture coordinates in image-space at a pixel (i,j)" [18]. They are "determined by the image-space partial derivative of the texture coordinates (s,t) at pixel (i,j)" [18] as

$$\tau_s(i,j) = \sqrt{\left(\frac{\delta s}{\delta i}\right)^2 + \left(\frac{\delta s}{\delta j}\right)^2} \text{ and} \tag{2.22}$$

$$\tau_t(i,j) = \sqrt{\left(\frac{\delta t}{\delta i}\right)^2 + \left(\frac{\delta t}{\delta j}\right)^2} \ . \tag{2.23}$$

"If either of τ_s or τ_t doubles, the pattern frequency in the corresponding direction must be halved to yield the same image-space frequency" [18]. To evaluate the partial derivatives of the texture coordinates, the GLSL build-in functions `dFdx` and `dFdy` are used. Then, the "frequency-adjusted pattern $\hat{P}$ is defined by the evaluation of P" [18] through a texture lookup.

$$\hat{P}_{l_s,l_t}(s,t) := P(s \cdot 2^{-l_s}, t \cdot 2^{-l_t}) \ . \tag{2.24}$$

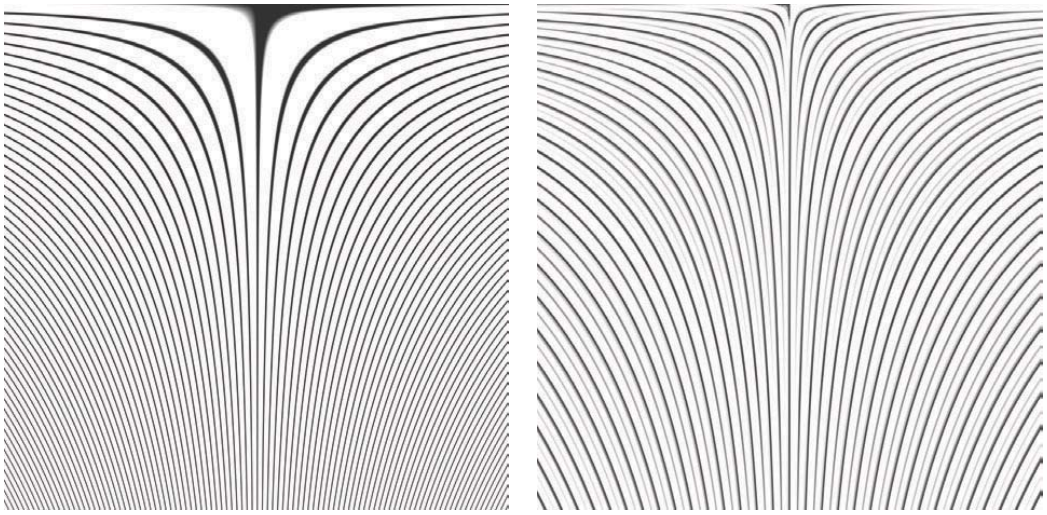
The texture lookup is performed with the GLSL build-in function `texture2D` and returns a rgba-color value. A bilinear interpolation can be applied between neighboring resolution levels, i.e. different rgba-color values

$$\begin{aligned} c(s,t) = (1-\tilde{l_s}) \cdot \left((1-\tilde{l_t}) \cdot \hat{P}_{\lfloor l_s \rfloor \lfloor l_t \rfloor}(s,t) + \tilde{l_t} \cdot \hat{P}_{\lfloor l_s \rfloor \lceil l_t \rceil}(s,t)\right) \\ +\tilde{l_s} \cdot \left((1-\tilde{l_t}) \cdot \hat{P}_{\lceil l_s \rceil \lfloor l_t \rfloor}(s,t) + \tilde{l_t} \cdot \hat{P}_{\lceil l_s \rceil \lceil l_t \rceil}(s,t)\right) \ , \end{aligned} \tag{2.25}$$

where

$$\tilde{l}_s = l_s - \lfloor l_s \rfloor \ , \ \tilde{l}_t = l_t - \lfloor l_t \rfloor$$

"denote the fractional parts of l_s and l_t" [18], with $l_s, l_t \in \mathbb{R}$. For further details see [18]. Figure 2.18 shows the adaptive pattern scaling. The code for the adaptive pattern shader B.4 can be found in Appendix B.1. Similar to the default shader, color information and blending are applied to the texel values. The user can select between two versions of the adaptive pattern shader, with or without bilinear interpolation. The advantages and drawbacks of both versions are discussed in Section 3.2.2. These shaders are used to render the eigenvector field directions. Figure 3.4(b) shows the adaptively scaled pattern with interpolation.



(a) (b)

Figure 2.18
"Strong anisotropic surface texture coordinate stretching is addressed by an adaptive pattern: (a) regular stripe pattern and (b) adaptive stripe pattern" [18]. Image from [18].

3. The previous shader is extended by scaling the pattern according to the transformed eigenvalues. Instead of Equation (2.22) and Equation (2.23), the frequency of the pattern is computed with the following formulas:

$$\tau_s(i,j) = \sqrt{\left(\frac{\delta(s \cdot \tilde{\lambda}_1)}{\delta i}\right)^2 + \left(\frac{\delta(s \cdot \tilde{\lambda}_1)}{\delta j}\right)^2} \text{ and} \tag{2.26}$$

$$\tau_t(i,j) = \sqrt{\left(\frac{\delta(t \cdot \tilde{\lambda}_2)}{\delta i}\right)^2 + \left(\frac{\delta(t \cdot \tilde{\lambda}_2)}{\delta j}\right)^2} \ , \tag{2.27}$$

where $\tilde{\lambda}_1$ and $\tilde{\lambda}_2$ are the transformed eigenvalues. Again, color information and blending are applied and the user may select between two versions of the adaptive pattern shader, with or the without bilinear interpolation. These shaders are used to display the tensor line directions and the physical behavior (see Figure 3.6).

Color Mapping

The user can select between different color modes. For the color modes 3 – 6, a color map (color table) can be selected and be modified. The range of the mapped values can either be set by the user or be adjusted to the maximum and minimum values of the selected color mode. According to the selected color mode the focus of the observer is directed to different features of the tensor field. The color information is passed to the shader (see Section 2.4.2) and applied according to the following color schemes:

1. no color (all rgb-color values are set to 1.0),
2. color by tensor field (the eigenvector field is displayed in red and the minor eigenvector field is displayed in blue),
3. color by eigenvalues (the magnitude of the eigenvalues is mapped to a rgb-color value),
4. color by transformed eigenvalues (the magnitude of the transformed eigenvalues is mapped to a rgb-color value),
5. color by fractional anisotropy (the magnitude of the fractional anisotropy is mapped to a rgb-color value) and
6. color by shear stress (the magnitude of the shear stress is mapped to a rgb-color value).

For a given color input parameter, the rgb-color value is determined by a color table lookup. Thereby, the input color parameter is restricted to the range of the color table, which can be determined by the user. If the input color parameter is below or above this range, the input parameter is clamped to the minimum and maximum values of the color table, and the corresponding rgb-color value is returned.

2.4.3 Implementation

The reader should be familiar by now with the single steps that are necessary for the texturization. For texture mapping the poly-lines of the segmentation are triangulated. This is performed with the CGAL constrained Delaunay triangulation algorithm. The CGAL constrained Delaunay algorithm triangulates the convex hull of polygons. For non-convex polygons, triangles outside the boundary must be removed. For the triangulation, each triangle is tested to see if it is inside or outside

the boundary. Then, triangles outside the boundary are removed. After the triangulation, the texture coordinates are computed and, if required, scaled according to the selected method. Thereafter, the attributes for the shader are set: namely, the texture coordinates, the transformed eigenvalues and the color values. Finally, the shader is activated to the user selection and the image is rendered. The user has the option to blur the rendered image by the fractional anisotropy or shear stress (see Section 2.4.4).

2.4.4 Blur by Fractional Anisotropy or Shear Stress

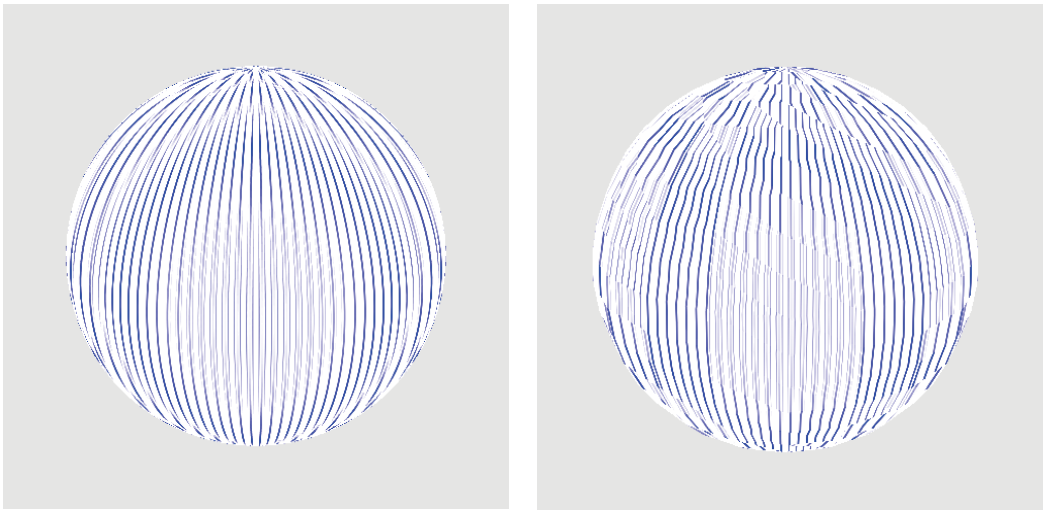
An optional post-processing filter may be applied to the texturization. To do so, the texturized eigenvector field is rendered into a frame buffer object. The frame buffer object stores the rendered image and an additional post-processing step applies the blur according to the fractional anisotropy or shear stress. Three different textures are passed to the GLSL post-shader:

- an image texture (the content of the frame buffer object),
- a Gauss kernel and
- a texture with the normalized inverse fractional anisotropy or shear stress.

The fragments of the image texture are convolved with the Gauss kernel, where the size of the Gauss kernel is a linear combination of a global user determined factor and the texture lookup of the fractional anisotropy or shear stress. For blurring by fractional anisotropy, isotropic regions are convolved with a larger kernel, whereas anisotropic regions are convolved with a smaller kernel. Thus, isotropic regions – regions in the vicinity of degenerate points – are blurred. These are regions where the difference between the eigenvalues is small and the directional behavior of the eigenvector fields can be unstable.

2.4.5 Constraints

The quality of the texturization depends strongly on the OpenGL linear texture coordinate interpolation and the triangulation which is used. Furthermore, the adaptive pattern shader presented by Hummel et al. [18] was designed for highly sampled surfaces. With low polygonal surfaces, the discrete steps of the scaled pattern get visible and the OpenGL linear texture coordinate interpolation is perceptible (see Figure 2.19). Adaptively scaling the pattern with the shader depends on the following factors: the cell sizes, the input pattern frequency, the zoom level, the image size and the transformed eigenvalues. In particular, the discrete steps of the adaptively scaled pattern by the eigenvalues are noticeable in regions where the magnitude of the eigenvalues for an eigenvector field exhibit a high range of variation. The zoom level and the size of the rendered image are crucial factors influencing the result, as computation of the discrete resolution levels depends on the image-space partial derivatives (as explained in Section 2.4.2). The impact of the input pattern frequency and cell size is



(a) (b)

Figure 2.19
Sphere with adaptively scaled stripe pattern: (a) high resolution triangulation and (b) low resolution triangulation.

discussed in Section 3.2.3. For displaying the physical behavior different approaches have been presented. All approaches have the drawback that the minimum pattern frequency and maximum pattern frequency for encoding the physical behavior of the tensor field are restricted to a small range. Furthermore, if the eigenvector field is textured with a directional stripe pattern, unsteady transitions emerge between the cells. This impact, which depends on the algorithm used, can be reduced and is discussed in the next chapter.

Chapter 3

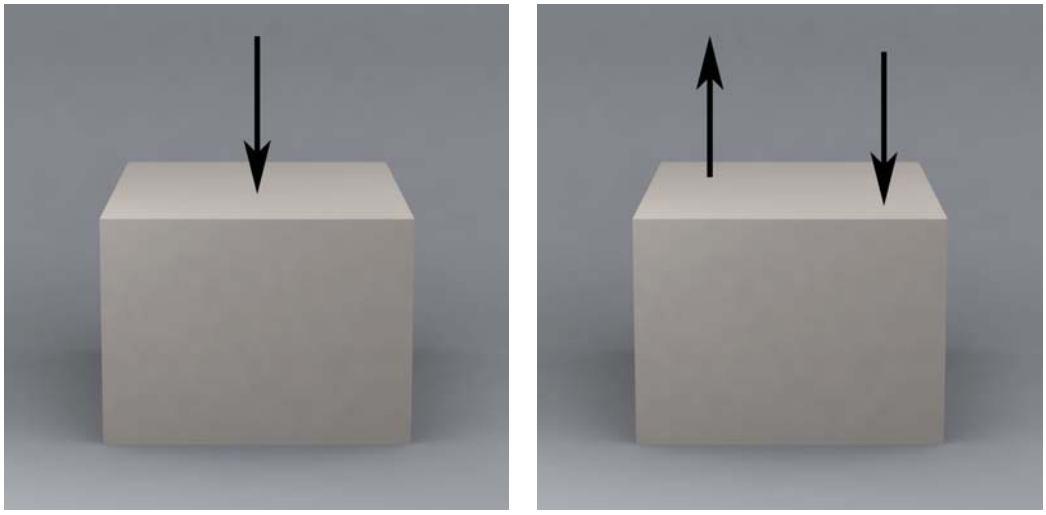
Analysis of Geometry-based and Texture-based Visualization of Tensor Fields

In this chapter, the results of the developed visualization approaches are presented and evaluated by means of two simulated datasets. Firstly, these datasets are introduced. This is followed by the results of the developed visualization approaches. The results are presented in two subsections, one for the geometry-based visualization and one for the texture-based visualization. The texture-based visualization offers the user various possibilities to observe the tensor fields, which focus on different features. These can be, for example, the direction of the major or minor eigenvector field, the magnitude of the eigenvalues and the anisotropy or shear stress. The design of the input texture pattern has a broad impact on the resulting visualization. The input texture pattern design is discussed in an exemplary fashion. The discussion ends with a comparison of the geometry-based and texture-based visualization approaches. The conclusion of this chapter places the work presented in the context of related works, which have been introduced in the first chapter. Finally, suggestions for further improvements are made.

3.1 Datasets

As explained in Chapter 1 material properties in mechanical engineering such as stress and strain can be described by stress tensors. For the evaluation and development of the geometry-based and texture-based visualization approaches presented here, two simulated tensor datasets are used: the one-point load and the two-point load. The one-point load is a solid block given as a cubic volume with a single load applied to it. The two-point load is a solid block given as a cubic volume with two loads of opposing sign applied to it. A Finite Element Method (FEM) generates the two datasets. The FEM approach uses a mesh discretization of the domain, the so-called elements. The 3D stress examples are generated on a discrete domain with $10 \times 10 \times 10$ elements. For

topological extraction and segmentation, slicing the 3D stress tensor field reduces the dimension by one. Figure 3.1 shows an outline of the cubic domain with two different load cases applied.



(a) (b)

Figure 3.1
Two cubic domains with loads applied: (a) one-point load and (b) two-point load.

3.2 Results

This section presents the results of the developed visualization frameworks. Firstly, geometry-based visualization of the topological graph and the segmentation are presented. This is followed by texture-based visualization, which is only applied to the segmentation. In most of the examples, the cells that contain one or more degenerate points are not textured because in those cells the tensor line behavior exhibit irregularities. Degenerate points are marked as black spheres.

3.2.1 Geometry-based Visualization of Segmented Tensor Fields

In Figure 3.2, different examples of the two-point load are displayed: the segmentation-based and topology-based glyph placement. For computing the barycentroids, a pre-processing of the poly-lines was performed with a sampling distance parameter of 1.25 % of the maximal side length of the domain and an angle threshold parameter of 165 °. The glyphs are placed at the pre-computed barycentroids and scaled according to the transformed eigenvalues. For the eigenvalue transformation a logarithmic function was used (see Equation (2.2)) with the following parameters: $a = 1.5$, $c = 1.0$ and $\sigma = 0.1$. The color for all three examples is applied according to the major eigenvalues. The upper left circle shows expansive forces while the lower right circle

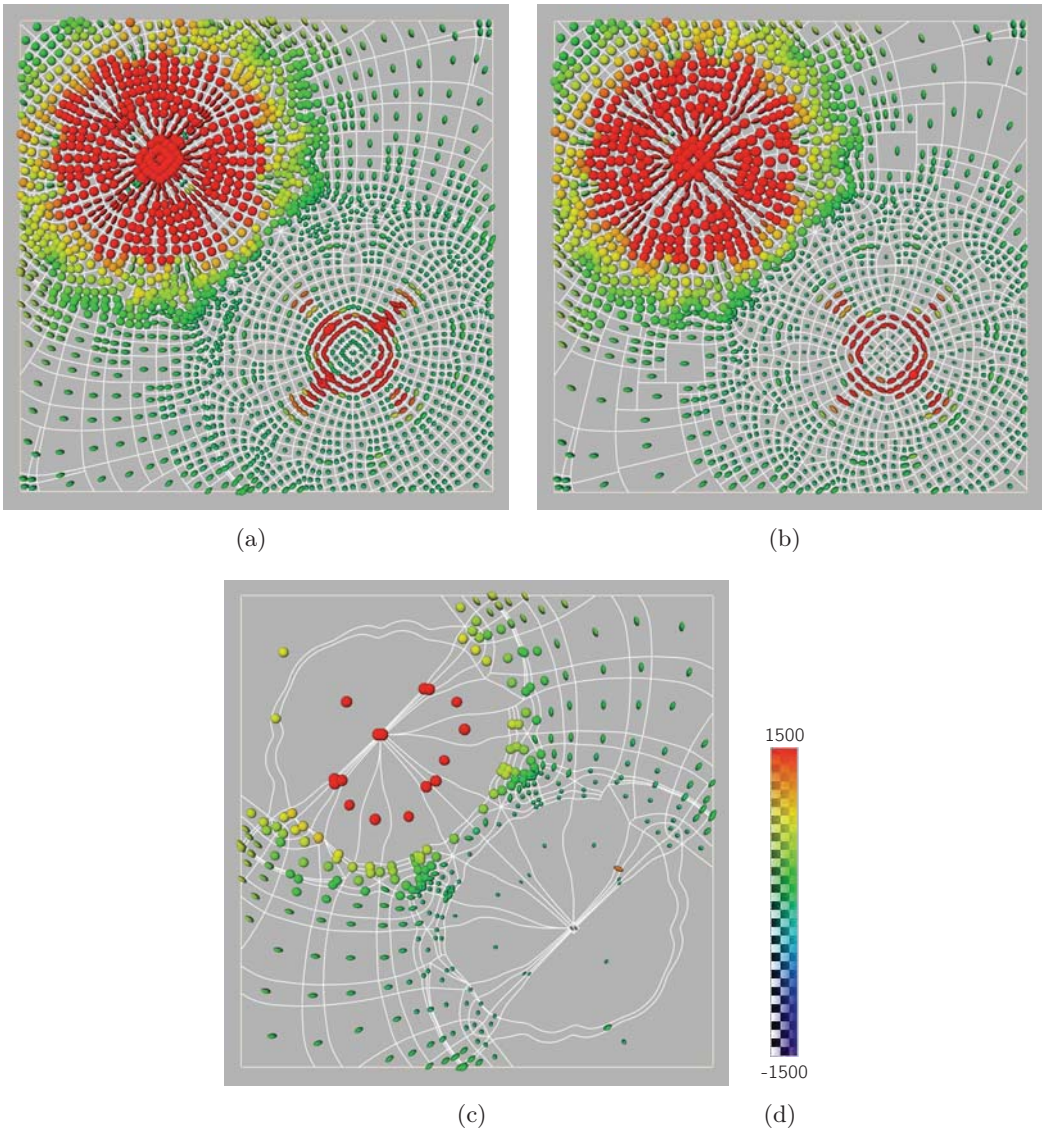


Figure 3.2
Segmentation-based and topology-based glyph placement: (a) segmentation without merged cells, (b) segmentation with merged cells and (c) topological graph. The color information is applied according to the major eigenvalues: (d) color map

shows compressive forces. The cell size is a crucial factor in regard to the perception of the information encoded in the glyphs. This is especially noticeable in the glyph placement based on the topology extraction example (see Figure 3.2(c)). There, the cells exhibit irregular shapes and sizes. The geometric icons overlap if the extracted cells are very small. Conversely, the glyphs are widely spaced if the extracted cells are large.

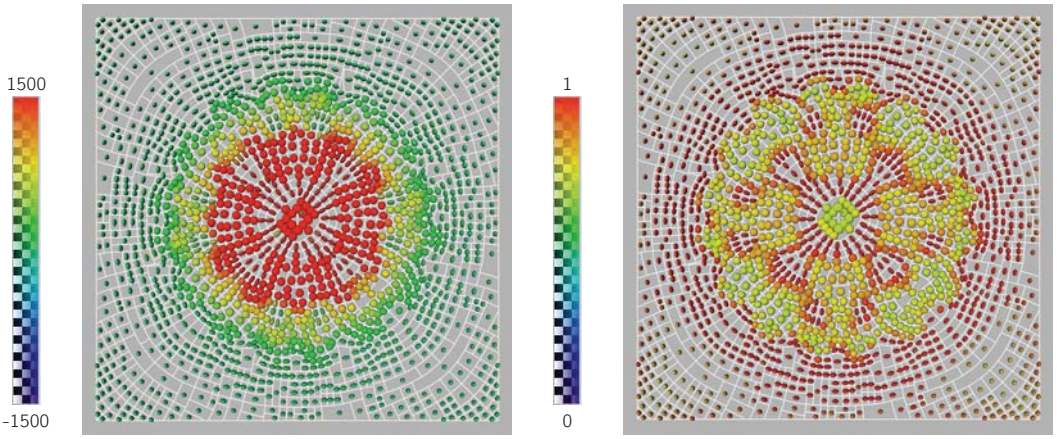


Figure 3.3
One-point load with geometry-based visualization: the color is applied according to (b) the major eigenvalues and (d) the fractional anisotropy. Color map (a) ranging from −1500 to 1500 and (c) ranging from 0.0 to 1.0.

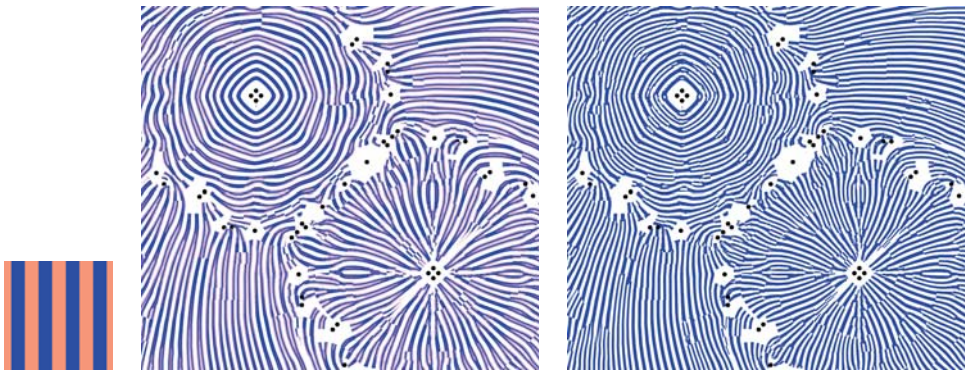
Figure 3.3 depicts two examples of the one-point load. In Figure 3.3(b), the color is applied according to the major eigenvalues. In the second example 3.3(d), the color information corresponds to the difference between the major and minor eigenvalues, as does the shape of the glyphs. Isotropic glyphs are encoded in yellow, which corresponds to spherical geometries. Anisotropic glyphs are encoded in red, which results in well-marked ellipses.

3.2.2 Texture-based Visualization of Segmented Tensor Fields

For the texturization, input patterns for the major and minor eigenvector field must be provided. In most of the visualization examples directional input texture patterns are used, where an input pattern with vertical stripes corresponds to the major eigenvector field and an input pattern with horizontal stripes corresponds to the minor eigenvector field. The declaration of the input pattern is restricted to a single eigenvector field if the input pattern is specified in the figures.

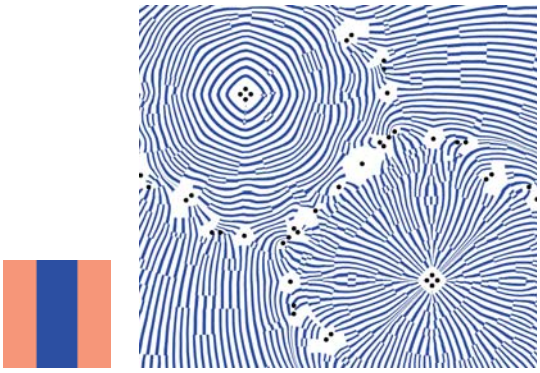
Directional Information of the Eigenvector Field

Three different visualizations of the direction of the minor eigenvector field are shown in Figure 3.4. As an input texture, different stripe patterns with an alpha channel were used. Pattern repetition is performed by scaling the texture coordinates as explained in Section 2.4.2 (see Figure 3.5(b)); the side length threshold is set to 1.25 % of the maximal side length of the domain. The averaging of opposing edges and the rounding of the fractions (see Equations (2.15) and (2.16)) results in unsteady



(a) (b) (c)

Figure 3.4
Comparison of different tensor line shaders I: (a) input pattern and (b) adaptive pattern shader with interpolated resolution levels and (c) adaptive pattern shader without interpolated resolution levels. Both shaders are used with the unit square texture coordinates.

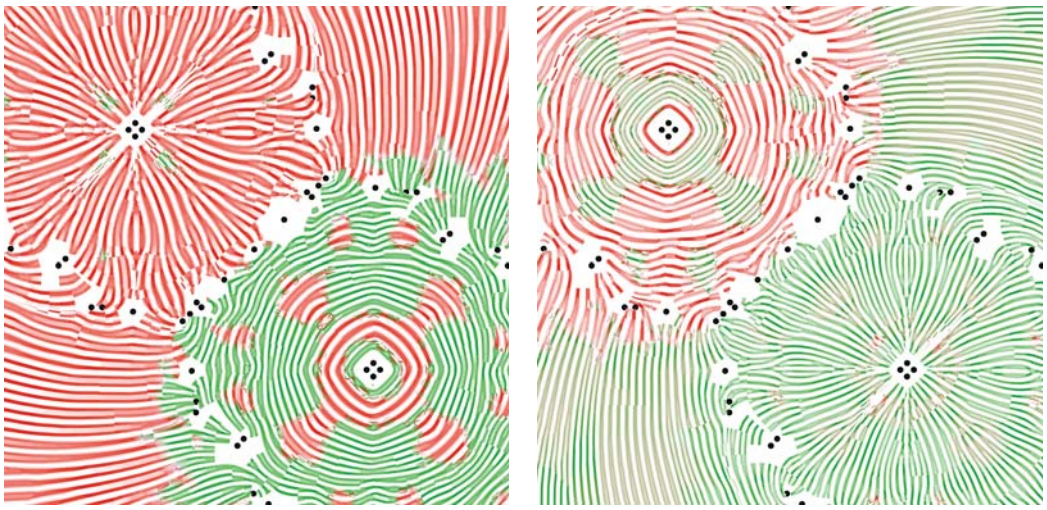


(a) (b)

Figure 3.5
Comparison of different tensor line shaders II: (a) input pattern and (b) texture mapping rendered with scaled texture coordinates and "the default shader".

transitions between some cells; however, this impact strongly depends on the user determined minimum side length threshold. Figure 3.4(b) shows the adaptively scaled pattern with interpolation between the resolution levels, as presented by Hummel et al. [18], and Figure 3.4(c) shows the adaptively scaled pattern without interpolation. Without interpolation the rendered image is "sharper"; however, unsteady transition between different resolution levels emerge. With interpolation some of these unsteady transitions can be compensated. The user decides which algorithm is appropriate for a particular application.

Directional and Physical Behavior of the Eigenvector Field



(a) (b)

Figure 3.6
Major and minor eigenvector field with textures applied: (a) displays the major eigenvector field and (b) displays the minor eigenvector field. The pattern is adaptively scaled by the corresponding eigenvalues and the color visualizes the metric.

Figure 3.6 depicts a modified version of the adaptively scaled pattern (see Shader B.5 in Appendix B). The pattern is scaled by an additional factor: the transformed eigenvalues. The eigenvalues are transformed with a transfer function based on the hyperbolic tangent (see Equation (2.4)) with $c = 0.1$, $a = 3.0$ and $\sigma = 0.5$. The transformed eigenvalues range from 2.5 to 3.5. The transformed eigenvalues are used to scale the input texture pattern according to Equation (2.26) and Equation (2.27). Transformed eigenvalues smaller than 3.0 refer to compressive forces. This results in a higher frequency pattern, which is displayed in green. Transformed eigenvalues larger than 3.0 refer to expansive forces. This results in a lower frequency pattern, which is displayed in red.

Experimental Result

In the following, an experimental results is discussed. This texturization approach visualizes effectively the stress in the volume; however, the symmetric input pattern – a circle – is distorted by the shape of the cell, which is not connected to the internal forces acting in the tensor field. The texture coordinates are scaled by the transformed eigenvalues, they exceed the range from 0.0 to 1.0 and the pattern is repeated. The texture mapping is performed with the default shader. Figure 3.7 shows two superpositions of both eigenvector fields. The same input texture pattern – a white circle with an alpha channel – is used for both eigenvector fields. Color is applied

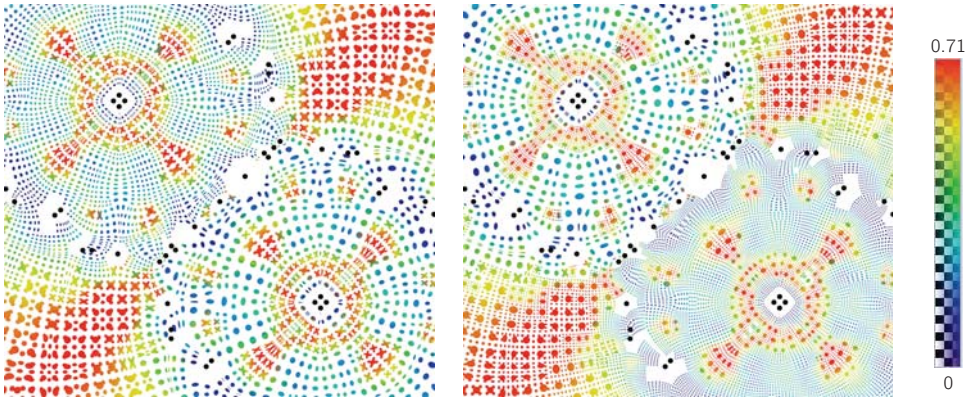


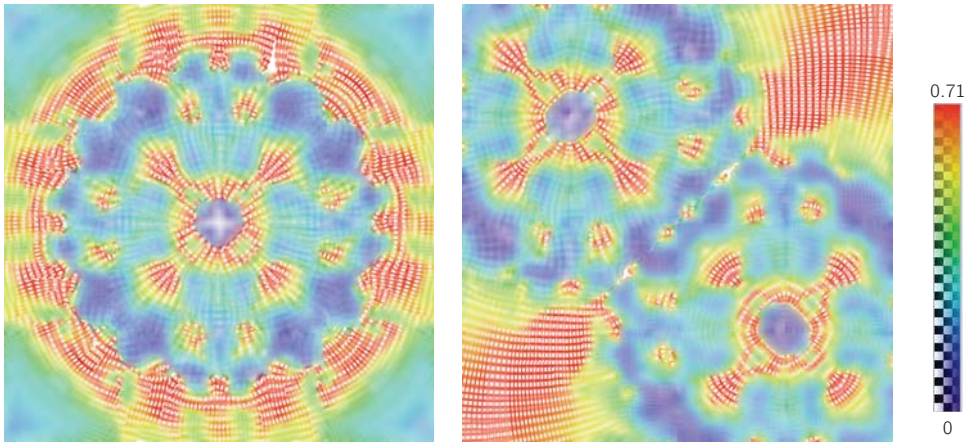
Figure 3.7
The pattern scaling is encoded in the texture coordinates: (a) the pattern is scaled down and repeated often for large values of the transformed eigenvalues and (b) large patterns refer to large values of the transformed eigenvalues. Color is applied according to the fractional anisotropy: (c) color map.

according to the fractional anisotropy. Red indicates anisotropic regions. In these regions, patterns with different sizes and margins for the major and minor eigenvector field are superimposed. Blue indicates isotropic regions. In these regions, patterns for the major eigenvector field coincide patterns for the minor eigenvector field. In both examples the hyperbolic tangent is used as a transfer function. In Figure 3.7(a), the parameters for the transfer function are set to the following values: $c = 0.1$, $\sigma = 0.75$ and $a = 1.0$. In Figure 3.7(b), the parameters for the transfer function are set to the following values: $c = 0.1$, $\sigma = 1.5$, $a = 0.0$ and the user determined threshold for the maximum number of pattern repetitions is set to four (see Equation (2.19) and Equation (2.20)).

Anisotropy and Shear Stress

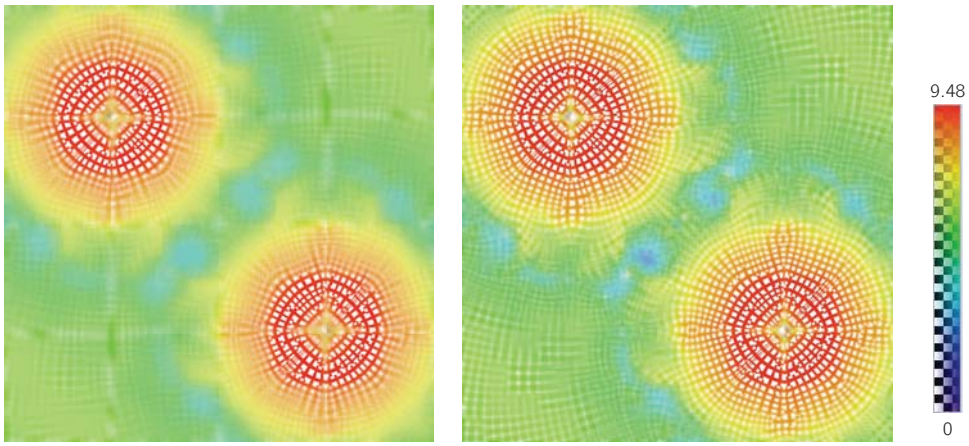
Figure 3.8 shows post-processing filtering by a superposition of both eigenvector fields. The major and minor eigenvector fields are shaded with an eigenvalue scaled pattern (Shader B.5 in Appendix B) and colored by the fractional anisotropy. The resulting image is blurred by the fractional anisotropy. The blur emphasizes regions where the magnitude of the eigenvalues is almost the same. These isotropic regions do not exhibit a distinctive directional behavior and the stripe pattern frequency is approximately the same. Unblurred, anisotropic regions (colored in red) indicate a difference between the magnitude of the eigenvalues. This difference is observable in a different pattern frequency of the major and minor eigenvector field.

Regions with a high shear stress are of special importance in engineering. These are regions where the material tends to fail. Figure 3.9 shows a superposition of both tensor fields, which are blurred colored by the shear stress magnitude. Regions with a



(a) (b) (c)

Figure 3.8
Superposition of both eigenvector fields blurred by fractional anisotropy: (a) displays the one-point load and (b) displays the two-point load. Color is applied according to the fractional anisotropy: (c) color map.



(a) (b) (c)

Figure 3.9
Superposition of both eigenvector fields blurred by shear stress: (a) displays the one-point load and (b) displays the two-point load. Color is applied according to the shear stress: (c) color map.

low shear stress are blurred, as they are of less interest. In doing so, the focus of the observer is directed to the regions of interest, i.e. regions with high shear stress, and information of less interest is de-emphasized. The mapped texture is scaled according to side length of the cells, shaded with the default shader and displays the directional behavior of the eigenvector fields.

3.2.3 Texture Design and Input Pattern Frequency

As noted previously, the input pattern frequency is critical in regard to the resulting visualization. For the default shader with scaled texture coordinates, a pattern with low frequency should be chosen, i.e. a pattern with one stripe. Otherwise, if the input pattern frequency is too high, aliasing artifacts arise. For scaling of the pattern with the shader (where the unit square texture coordinates are used), a dense input pattern should be chosen; however, this input pattern frequency depends on the zoom level and image size. If the pattern frequency is too low for the zoom level only a small cutout of the texture is mapped onto the cell. As a result, the original pattern is no longer discernible anymore. In the following section, texture design and input pattern frequency are discussed.

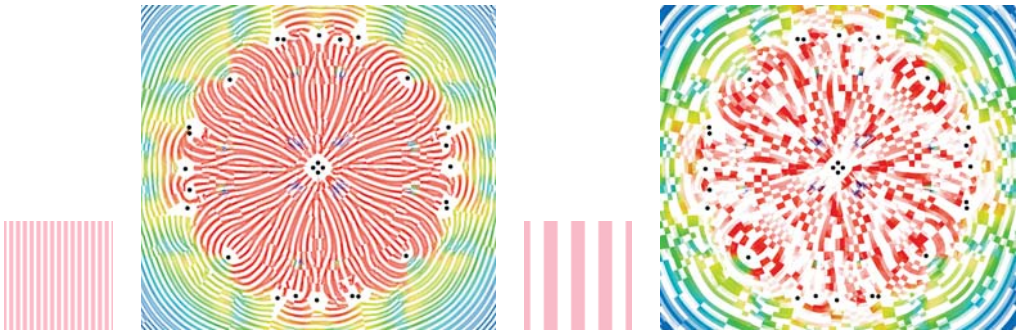
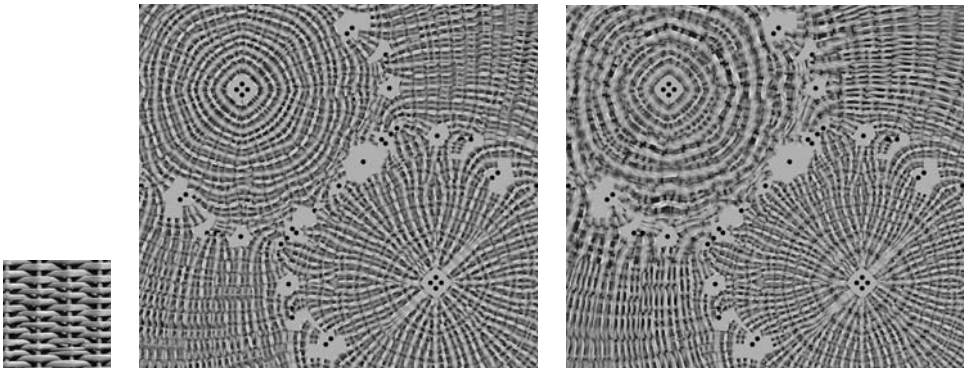


Figure 3.10
One-point load rendered with two different input pattern: (a) high frequency input pattern used (b) for the visualization and (c) low frequency input pattern used (d) for the visualization. The alpha channel of the input pattern is marked in a lighter red. Color is applied according to the transformed major eigenvalues.

Figure 3.10 shows the impact of input pattern frequency (see Shader B.5). Both images are rendered with an adaptively scaled pattern, the same image size and the same zoom level, while the input pattern is different. In the first example, an input pattern with a high frequency is used. The input pattern is aligned according to the tensor lines of the major eigenvector field; the original stripe pattern is well perceptible. The second example is rendered with a low frequency pattern. Particularly in the center of the image, where the cells are small, only a small cutout of the input texture is used. There, the texturization acts more like a rendering of expressive brushstrokes with random placement. The directions of the tensor lines are hardly perceptible. Therefore, the physical characteristics of the tensor field are only reflected in the colorization.

For the majority of the examples presented in this work, the major and minor eigenvector fields are separately discussed. In Figure 3.11 a bidirectional input texture pattern is used. The bidirectional weave pattern is distorted and aligned according to



(a) (b) (c)

Figure 3.11
Two-point load rendered with a bidirectional input pattern: (a) bidirectional input pattern, (a) mapped into tensor field (the pattern is scaled according to the zoom level) and (c) mapped into the tensor field (the pattern is scaled by the transformed eigenvalues and the zoom level).

the tensor field. The directional behavior of both eigenvector fields can be displayed without a superposition of the minor and major eigenvector field. The frequency of the input pattern is different in the s and t directions. For this input pattern, the direction with the higher frequency dominates when the user observes the rendering.

Another proposal for texture mapping was to design a directional input noise pattern to overcome the problem of unsteady transitions between the cells of the segmentation and the different resolution levels of the adaptively scaled pattern. The problem of unsteady transitions is only reduced and at a closer look they are still perceptible. Figure 3.12 displays the results of the rendering. If only one eigenvector field is displayed, the direction of the eigenvector field is effectively visualized. The directional behavior of the tensor field is no longer readily visible when both eigenvector fields are superimposed and the rendered image appears very noisy.

3.3 Discussion

Although the physical world is three-dimensional, two-dimensional tensor field visualizations are still of interest. They might be used to explore the data by slicing. The two-dimensional tensor field visualization methods presented here are able to visualize internal forces acting in the tensor field and slice analysis is a step on the way to understand three-dimensional visualization. The visualizations give an impression of the stress distribution, its directions and strength; however, they demand a degree of experience of the observer. In the following the geometry-based and texture-based approaches are compared by means of an example. In addition, the advantages and drawbacks of each method are discussed.

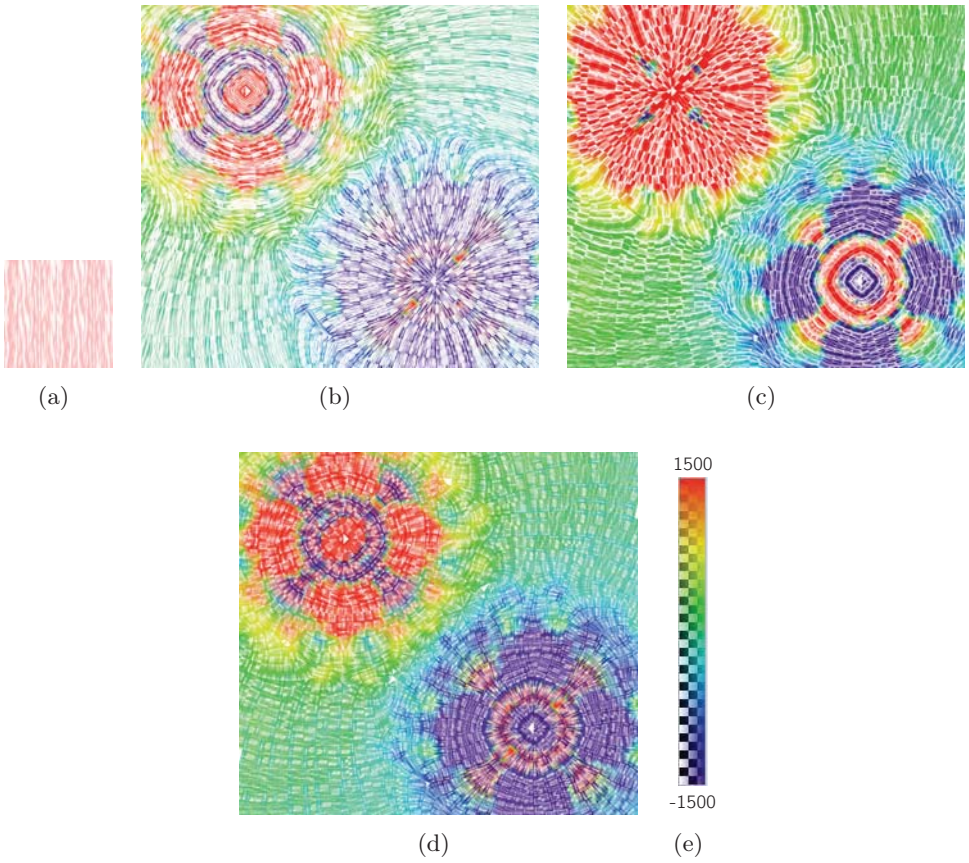


Figure 3.12
Two-point load rendered with a directional input noise pattern: (a) input pattern, (b) minor eigenvector field, (c) major eigenvector field and (d) superposition of both eigenvector fields. Color is applied according to the eigenvalues: (d) color map.

3.3.1 Mutual Agreement and Differences of the Visualization Methods

The geometry-based visualization and several results of the texture-based visualization have been presented to the reader. To summarize the visualization methods developed in this thesis, Figure 3.13 shows a comparison of the developed visualizations. Texture-based visualization is able to render the directions of the eigenvector fields and the eigenvalues. Furthermore, the anisotropy is implicitly encoded in the rendering by a different pattern density. The texture-based visualization gives a good global overview of the directions of the tensor lines; however, as discussed in Section 2.4.5, pattern scaling by the eigenvalues is restricted to only few discrete resolution levels. In the geometry-based approach, the glyphs are scaled according to the eigenvalues. The magnitude of these eigenvalues is reflected in the size and shape of the glyphs. Color can represent the magnitude of the major or minor eigenvalues. In contrast to texture-based visualization, the direction of the eigenvector field is not perceptible

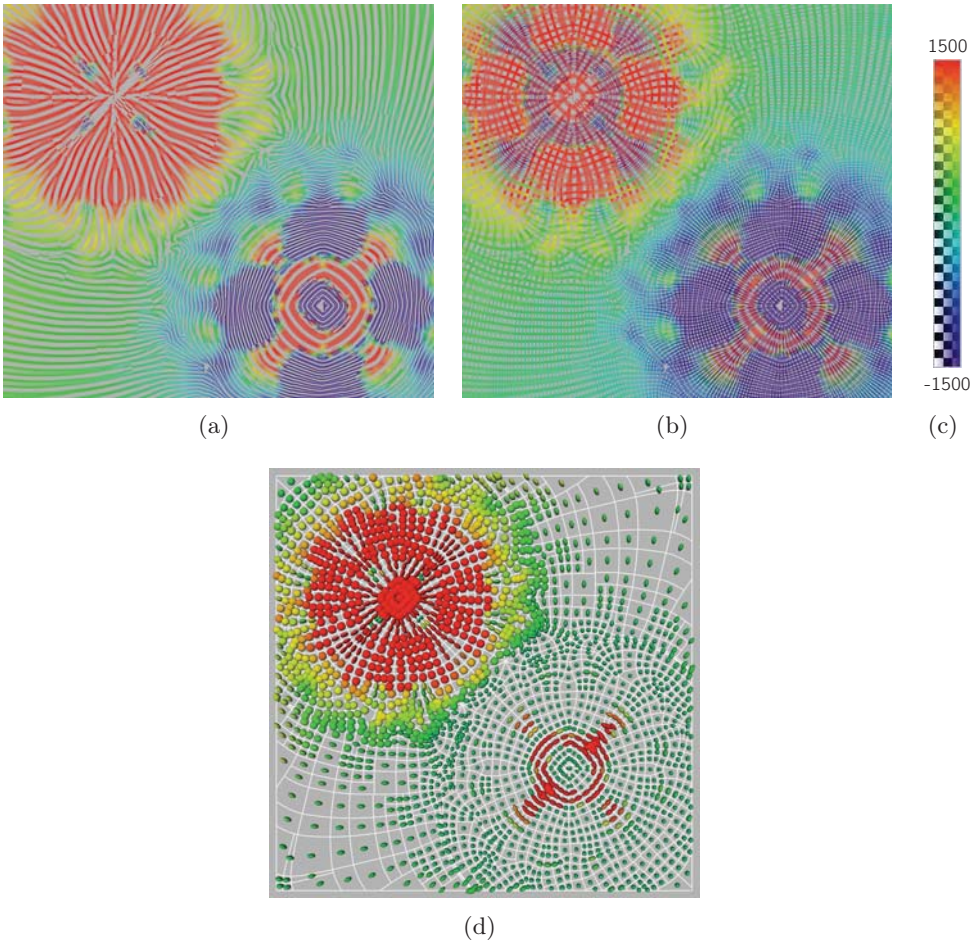


Figure 3.13
Tensor field visualization: texture-based (a) major eigenvector field, (b) superposition of both eigenvector fields and (d) geometry-based visualization. Color is applied according to the eigenvalues: (c) color map.

due to the discrete nature of the glyphs, but the glyphs are continuously scaled by the transformed eigenvalues.

3.3.2 Evaluation of the Visualization Methods

As mentioned before, the quality of the topological-based and segmentation-based glyph placement depends strongly on the quality of the tiling. Furthermore, regions with dense sampling and large glyphs are emphasized. For computing the barycentroid according to the algorithm presented by Rustamov et al. [22] an eigendecomposition and gradient descent must be computed. These are expensive computations regarding the runtime, which strongly depend on the sampling of the topological graph or segmentation; however, for placing the glyphs in the segmentation a sparse sampling

is sufficient (see Section 3.2.1). Computing the centers of convex cells according to Equation (2.14) optimizes the performance.

Geometry-based visualization must always address the trade-off between glyph size and distance between the glyphs. In contrast, texture-based visualization methods are able to give a piecewise continuous overview and the appearance of the rendering can be easily modified by the input texture. As presented in the section "Results", the texture-based visualization method developed here offers the user various possibilities for manipulating the rendering.

The texture must be mapped onto the cells in such a way that the input texture pattern is aligned parallel to the bounding tensor lines of the cells. This texture mapping results in unsteady transitions between the cells, and this problem may be reduced but not eliminated; when the user zooms in these transitions are perceptible. If the scaled texture coordinates are used with the default shader, the shape of the cells distorts the texture. In particular, this holds if the texture coordinates are scaled by the transformed eigenvalues. Anisotropic distortion of the cells may be addressed by adaptively scaling the pattern. Furthermore, the range for encoding the physical behavior of the tensor field in the image-space pattern frequency is restricted to a small interval.

Despite the drawbacks discussed, the texture-based visualization is able to display a reasonable overview of the direction of the eigenvector fields and to visualize different characteristics of the tensor field. Furthermore, the observer can interactively fade in and out the eigenvector field of interest and change the color map. In contrast to other texture-based visualization methods, which only take into account the local directional behavior, the global directional behavior is addressed due to the parameterization provided by the segmentation.

3.4 Conclusion

The purpose of this thesis was to develop a geometry-based and texture-based visualization framework for segmented tensor fields. The given topology-based segmentation provides the input for the visualization. For the geometry-based approach one local exponent is placed into the center of the each extracted cell. Since the cells are non-convex, this is achieved by computing the barycentroid based on an interior distance measurement. The information that is encoded in this discrete visualization approach depends strongly on the sampling, i.e. the quality of the segmentation. Dependent on the scaling and coloring of the glyphs, various properties of the tensor field may be emphasized.

For the piecewise continuous texture-based visualization approach, the texture coordinates are computed in such a way that the input texture pattern is aligned according to the direction of the eigenvector fields. The user may explore both eigenvector fields at a time or fade in and out one eigenvector field, which is of special interest. The compressive and expansive forces of the tensor field are encoded in the pattern density. In addition, the user has more options to explore the tensor field:

to select a color mode or blur the tensor field by the fractional anisotropy or by the shear stress.

The information encoded in the texture-based visualization approach is similar to HyperLIC [28] or the physical based rendering [16] introduced in Section 1.4.1. HyperLIC visualizes the direction of one eigenvector field and the anisotropy. The physical based visualization provides information on the directions of both eigenvector fields and the expansive and compressive forces. The anisotropy is implicitly encoded in the line density of the texture. This information is also encoded in the newly developed framework; however, the minimum pattern frequency and maximum pattern frequency for encoding the physical behavior of the tensor field are restricted to a small range. Furthermore, in the vicinity of degenerate points and at the borders of the domain the directional behavior is not unambiguously defined. In most of the examples these cells are not textured.

To summarize, the framework developed here joins a segmentation-based visualization approach to the well-known existing visualization concepts for tensor fields. Compared to existing tensor field visualization methods the user has more options to change the appearance of the rendering. These options are for example different shaders, different input texture patterns, different color modes and de-emphasizing regions of less interest by blur.

3.5 Future Work

In Section 2.3.4, the constraints on computing the barycentroids were discussed. In our framework a slight noise in the potential was observable if the poly-line of the cell boundary exhibits irregularities in the sampling. Either the sampling of the boundaries may be improved or a different method for the gradient descent should be used, which is not sensitive to outliers in the potential.

As mentioned in Section 2.4.5, adaptive pattern scaling is vulnerable to the input pattern frequency, the zoom level and the image size. Since the image size is determined by the settings of the software module this aspect can be neglected. In our framework, several input textures with a determined pattern frequency have been used. One possible improvement would be to replace this input texture by a procedural texture, where the pattern frequency depends on the zoom level. Initially, the whole tensor field is displayed. Hence, a procedural input texture with a higher frequency would be necessary. When the user zooms in the texture should be replaced by a texture with a lower frequency.

Furthermore, edges of the extracted cells follow the tensor lines. These tensor lines are in general not straight. The texture is mapped onto the triangulated cells in a piecewise linear fashion due to the OpenGL linear texture coordinate interpolation. This is observable at high zoom levels. An adaptive supersampling and refinement of the triangulation of the extracted cells, which depend on the zoom level, would improve this artifact.

For the texture mapping, the cells are triangulated with the CGAL constrained

Delaunay triangulation. This triangulation is unable to cope with self-intersecting cells. In the vicinity of degenerate points, triangular shaped cells occur that taper. These polygons may have self-intersections. In the framework discussed here, these self-intersections could be removed by pre-processing. A triangulation algorithm that can cope with self-intersecting arbitrary shaped polygons would make the framework more robust.

Appendix A

Pseudo-Code

A.1 Pre-processing

Algorithm 1 addEqualDistanceToEdge($samplingDistance$)

for $i = 1 \rightarrow n$ **do**
  if $i = n$ **then**
    $v_{i+1} := v_0$
  end if
  if $i = 1$ **then**
    $v_{i-1} := v_n$
  end if
  $vector := v_{i+1} - v_i$
  $distance := vector.length()$
  if distance $>$ samplingDistance **then**
    $added := 1$
    $normVector := vector.normalize()$
    $numVertices := floor(distance/samplingDistance)$
    $samplingLength := distance/numVertices + 1$
    while $added \leq numVertices$ **do**
      $newPoint := v_i + added * samplingLength * normVector$
      $addToPolyline(newPoint)$
      $added := added + 1$
    end while
  end if
  $i := i + 1$
end for

Algorithm 2 pruneEdge($angleThreshold$, $samplingDistance$)

for $i = 1 \rightarrow n$ **do**
  if $i = n$ **then**
    $v_{i+1} := v_0$
  end if
  if $i = 1$ **then**
    $v_{prev} := v_n$
  end if
  $vector1 := v_{i+1} - v_i$
  $vector2 := v_{prev} - v_i$
  $angle := getAngle(vector1, vector2)$
  $distance := length(v_{prev}, v_{i+1})$
  if $angle \geq angleThreshold$ **and** $distance < samplingDistance$ **then**
    $removeFromPolyLine(v_i)$
  else
    $v_{prev} := v_i$
  end if
  $i := i + 1$
end for

A.2 Topology-based and Segmentation-based Glyph Placement

Algorithm 3 computeBarycentricWeights(p)

for $i = 1 \rightarrow n$ **do**
  if $i = n$ **then**
    $v_{i+1} := v_0$
  end if
  if $i = 1$ **then**
    $v_{i-1} := v_n$
  end if
  $Ai := getSignedArea(v_i, p, v_{i+1})$
  $AiM := getSignedArea(v_{i-1}, p, v_i)$
  $Bi := getSignedArea(v_{i-1}, p, v_{i+1})$
  $ri := length(p - v_i)$
  $riM := length(p - v_{i-1})$
  $riP := length(p - v_{i+1})$
  $wi := (riM * Ai - ri * Bi + riP * AiM)/(AiM * Ai)$
  $sumW := sumW + wi$
  $i := i + 1$
end for
for $i = 1 \rightarrow n$ **do**
  $wi := wi/sumW$
end for

Algorithm 4 computeCenterOfArea($vertices$)

$n := vertices.size()$
if $n > 2$ **then**
  if $polygonIsConvex(vertices)$ **then**
    $centerOfArea := computeCenterOfGravity(vertices)$
  else
    $initPoint := findInitialPoint(vertices)$
    $laplacian := computeStencilLaplacian(vertices)$
    $voronoiArea := computeVoronoiArea(vertices)$
    $gram := computeEmbeddedGramMarix(laplacian, voronoiArea)$
    $centerOfArea := minimizePotential(initPoint, gram, vertices, tolerance,$
      $initialExploreStep, finalExploreStep, maxUpdates)$
  end if
end if

Appendix B

Shaders

B.1 Texturization of Segmented Tensor Fields

```
1  // Vertex shader
2  void main()
3  {
4    gl_FrontColor = gl_Color;
5    gl_TexCoord[0] = gl_MultiTexCoord0;
6    gl_Position = ftransform();
7  }
8
9  // Fragment shader
10 uniform sampler2D texture;
11
12 void main()
13 {
14     gl_FragColor = texture2D(texture , gl_TexCoord[0].st);
15 }
```

Listing B.1
Plain shader for texture mapping

```
1  // Vertex shader
2  in vec4 texCoord1;
3  in vec4 texCoord2;
4  in vec4 color;
5  in vec4 color2;
6
7  varying vec4 varyTexCoord1;
8  varying vec4 varyTexCoord2;
9
10 varying vec4 varyColor1;
11 varying vec4 varyColor2;
12
13 void main()
14 {
15     varyTexCoord1 = texCoord1;
16     varyTexCoord2 = texCoord2;
17
```

```
18      varyColor1 = color;
19      varyColor2 = color2;
20
21      gl_FrontColor = vec4(1.0);
22      gl_Position = ftransform();
23  }
```

Listing B.2
Vertex shader used in the texture-based visualization approach

```
1  // Fragment shader
2  uniform sampler2D texture1;
3  uniform sampler2D texture2;
4
5  varying vec4 varyTexCoord1;
6  varying vec4 varyTexCoord2;
7
8  varying vec4 varyColor1;
9  varying vec4 varyColor2;
10
11 uniform float blend1;
12 uniform float blend2;
13
14 void main()
15 {
16   vec4 texel01 = texture2D(texture1, varyTexCoord1.st);
17   vec4 texel02 = texture2D(texture2, varyTexCoord2.st);
18
19   vec4 c1 = vec4(texel01.rgb * varyColor1.rgb, texel01.a * blend1);
20   vec4 c2 = vec4(texel02.rgb * varyColor2.rgb, texel02.a * blend2);
21
22   gl_FragColor = mix(c1, c2, texel02.a * blend2);
23 }
```

Listing B.3
Fragment shader of the default shader used in the texture-based visualization approach

```
1  // Fragment shader
2  uniform sampler2D texture1;
3  uniform sampler2D texture2;
4
5  varying vec4 varyTexCoord1;
6  varying vec4 varyTexCoord2;
7
8  varying vec4 varyColor1;
9  varying vec4 varyColor2;
10
11 uniform float blend1;
12 uniform float blend2;
13
14 uniform vec2 texSize;
15
16 vec2 texLookUp01;
17 vec2 texLookUp02;
18 vec2 texLookUp03;
19 vec2 texLookUp04;
20
21 vec4 getTexelValue(vec4 texCoord, sampler2D texture)
```

```
22 {
23   vec2 texLookUp = texCoord.st;
24
25   float dsx = dFdx(texCoord.s * texSize.x);
26   float dsy = dFdy(texCoord.s * texSize.y);
27
28   float dtx = dFdx(texCoord.t * texSize.x);
29   float dty = dFdy(texCoord.t * texSize.y);
30
31   float tau_s = sqrt(dsx * dsx + dsy * dsy);
32   float tau_t = sqrt(dtx * dtx + dty * dty);
33
34   float ls = log2(tau_s);
35   float lt = log2(tau_t);
36
37   texLookUp01  = vec2(texLookUp.s * exp2(-floor(ls)),
38           texLookUp.t * exp2(-floor(lt)));
39   texLookUp02  = vec2(texLookUp.s * exp2(-floor(ls)),
40           texLookUp.t * exp2(-ceil(lt)));
41   texLookUp03  = vec2(texLookUp.s * exp2(-ceil(ls)),
42           texLookUp.t * exp2(-floor(lt)));
43   texLookUp04  = vec2(texLookUp.s * exp2(-ceil(ls)),
44           texLookUp.t * exp2(-ceil(lt)));
45
46   float ls_frac = ls - floor(ls);
47   float lt_frac = lt - floor(lt);
48
49   vec4 texel = (1.0 - ls_frac)
50             * ((1.0 - lt_frac) * texture2D(texture, texLookUp01)
51             + lt_frac * texture2D(texture, texLookUp02))
52         + ls_frac * ((1.0 - lt_frac) * texture2D(texture, texLookUp03)
53             + lt_frac * texture2D(texture, texLookUp04));
54
55   return texel;
56 }
57
58 void main ()
59 {
60   vec4 texel1 = getTexelValue(varyTexCoord1, texture1);
61   vec4 texel2 = getTexelValue(varyTexCoord2, texture2);
62
63   vec4 c1 = vec4(texel1.rgb * varyColor1.rgb, texel1.a * blend1);
64   vec4 c2 = vec4(texel2.rgb * varyColor2.rgb, texel2.a * blend2);
65
66   gl_FragColor = mix(c1, c2, texel2.a * blend2);
67 }
```

Listing B.4
Fragment shader of adaptive pattern shader with linear interpolation of the discrete resolution levels

```
1 // Fragment shader
2 uniform sampler2D texture1;
3 uniform sampler2D texture2;
4
5 varying vec4 varyTexCoord1;
6 varying vec4 varyTexCoord2;
7
8 varying vec4 varyColor1;
9 varying vec4 varyColor2;
```

```
10
11 varying vec2 varyEvals;
12
13 uniform float blend1;
14 uniform float blend2;
15
16 uniform vec2 texSize;
17
18 vec2 texLookUp01;
19 vec2 texLookUp02;
20 vec2 texLookUp03;
21 vec2 texLookUp04;
22
23 vec4 getTexelValue(vec4 texCoord, sampler2D texture)
24 {
25   vec2 texLookUp = texCoord.st;
26
27   float mEV1 = varyEvals.s;
28   float mEV2 = varyEvals.t;
29
30   float dsx = dFdx(texCoord.s * texSize.x * mEV1);
31   float dsy = dFdy(texCoord.s * texSize.y * mEV1);
32
33   float dtx = dFdx(texCoord.t * texSize.x * mEV2);
34   float dty = dFdy(texCoord.t * texSize.y * mEV2);
35
36   float tau_s = sqrt(dsx * dsx + dsy * dsy);
37   float tau_t = sqrt(dtx * dtx + dty * dty);
38
39   float ls = log2(tau_s);
40   float lt = log2(tau_t);
41
42   texLookUp01  = vec2(texLookUp.s * exp2(-floor(ls)),
43           texLookUp.t * exp2(-floor(lt)));
44   texLookUp02  = vec2(texLookUp.s * exp2(-floor(ls)),
45           texLookUp.t * exp2(-ceil(lt)));
46   texLookUp03  = vec2(texLookUp.s * exp2(-ceil(ls)),
47           texLookUp.t * exp2(-floor(lt)));
48   texLookUp04  = vec2(texLookUp.s * exp2(-ceil(ls)),
49           texLookUp.t * exp2(-ceil(lt)));
50
51   float ls_frac = ls - floor(ls);
52   float lt_frac = lt - floor(lt);
53
54   vec4 texel = (1.0 - ls_frac)
55             * ((1.0 - lt_frac) * texture2D(texture, texLookUp01)
56             + lt_frac * texture2D(texture, texLookUp02))
57         + ls_frac * ((1.0 - lt_frac) * texture2D(texture, texLookUp03)
58             + lt_frac * texture2D(texture, texLookUp04));
59
60   return texel;
61 }
62
63 void main ()
64 {
65   vec4 texel1 = getTexelValue(varyTexCoord1, texture1);
66   vec4 texel2 = getTexelValue(varyTexCoord2, texture2);
67
68   vec4 c1 = vec4(texel1.rgb * varyColor1.rgb, texel1.a * blend1);
69   vec4 c2 = vec4(texel2.rgb * varyColor2.rgb, texel2.a * blend2);
70
71   gl_FragColor = mix(c1, c2, texel2.a * blend2);
```

```
72 }
```

Listing B.5
Fragment shader that scales the pattern by the transformed eigenvalues (with linear interpolation)

```
1  //Fragment shader
2  uniform sampler2D image;  //content of the OpenGL frame buffer
3  uniform sampler2D gaussTexture;
4  uniform sampler2D faTexture;
5
6  uniform vec2 texSize;
7  uniform float filterScale;
8  uniform float faFactor;
9
10 vec4 convoluteWithGaussKernel()
11 {
12     vec4 sum = vec4(0.0,0.0,0.0,0.0);
13     float gaussValue;
14     float gaussSum;
15     float x, y;
16     float gaussStepX, gaussStepY;
17
18     vec3 currentFa = vec3(texture2D(faTexture, gl_TexCoord[0].st));
19     float k = currentFa.x * faFactor;
20     float fs = filterScale;
21     vec2 offset = vec2(1.0 / texSize.x, 1.0 / texSize.y);
22
23     for (y = k; y ≥ -k; y--)
24     {
25         for (x = -k; x ≤ k; x++)
26         {
27             // the center element is at (0.5, 0.5) of the Gauss texture
28             gaussStepX = (1.0 / k * 2.0) * x;
29             gaussStepY = (1.0 / k * 2.0) * y;
30
31             gaussValue = (texture2D(gaussTexture,
32                     vec2(gaussStepX, gaussStepY))).r;
33             gaussSum += gaussValue;
34
35             vec4 texel = texture2D(image ,
36                     gl_TexCoord[0].st + vec2(x,y) * fs * offset);
37             sum += gaussValue * texel;
38         }
39     }
40
41     return sum/gaussSum;
42 }
43
44 void main()
45 {
46     vec4 texel = convoluteWithGaussKernel();
47
48     gl_FragColor = texel;
49 }
```

Listing B.6
Fragment shader for post-processing

References

[1] R. Abraham, J. E. Marsden, and R. Ratiu. *Manifolds, Tensor Analysis and Applications*. Springer-Verlag New York, Inc., USA, 2nd edition, 1988.

[2] A. Aldroubi and P. Basser. "Reconstruction of Vector and Tensor Fields From Sampled Discrete Data", 1999.

[3] C. Auer, J. Sreevalsan-Nair, V. Zobel, and I. Hotz. "2D Tensor Field Segmentation". In *Scientific Visualization: Interactions, Features, Metaphors*, volume 2 of *Dagstuhl Follow-Ups*. 2009.

[4] A. H. Barr. "Superquadrics and Angle-preserving Transformations". *IEEE Comput. Graph. Appl.*, 1:11–23, 1981.

[5] P. J. Basser and C. Pierpaoli. "Microstructural and Physiological Features of Tissues Elucidated by Quantitative-Diffusion-Tensor MRI". *Journal of Magnetic Resonance, Series B*, 111(3):209–219, 1996.

[6] W. Benger and H.-C. Hege. "Tensor Splats". In *Visualization and Data Analysis*, pages 151–162, 2004.

[7] D. Breßler. "Texturbasierte Methoden zur Visualisierung von Tensorfeldern", 2010. Bachelorarbeit.

[8] B. Cabral and L. C. Leedom. "Imaging Vector Fields Using Line Integral Convolution". In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 263–270, New York, NY, USA, 1993. ACM.

[9] R. R. Coifman, S. Lafon, A. B. Lee, M. Maggioni, F. Warner, and S. Zucker. "Geometric Diffusions as a Tool for Harmonic Analysis and Structure Definition of Data: Diffusion maps". In *Proceedings of the National Academy of Sciences*, pages 7426–7431, 2005.

[10] D. A. Danielson. *Vectors and Tensors in Engineering and Physics*. Addison-Wesley Publishing Company, Boston, MA, USA, 2nd edition, 1997.

[11] F. de Goes, S. Goldenstein, and L. Velho. "A Hierarchical Segmentation of Articulated Bodies". In *Proceedings of the Symposium on Geometry Processing*, SGP '08, pages 1349–1356, Aire-la-Ville, Switzerland, 2008. Eurographics Association.

[12] W. C. de Leeuw and J. J. van Wijk. "A Probe for Local Flow Field Visualization". In *Proceedings of the 4th conference on Visualization '93*, VIS '93, pages 39–45, Washington, DC, USA, 1993. IEEE Computer Society.

[13] T. Delmarcelle and L. Hesselink. "The Topology of Symmetric, Second-order Tensor Fields". In *VIS '94: Proceedings of the conference on Visualization '94*, pages 140–147, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[14] H. Hagen and C. Garth. "An Introduction to Tensors". In *Visualization and Processing of Tensor Fields*, Mathematics and Visualization, pages 3–13. Springer Berlin Heidelberg, 2006.

[15] K. Hormann and M. S. Floater. "Mean Value Coordinates for Arbitrary Planar Polygons". *ACM Transactions on Graphics*, 25:1424–1441, 2006.

[16] I. Hotz, L. Feng, H. Hagen, B. Hamann, B. Jeremic, and K. Joy. "Physically Based Methods for Tensor Field Visualization". In *Proceedings of IEEE Visualization 2004*, pages 123–130. IEEE Computer Society Press, 2004.

[17] I. Hotz, J. Sreevalsan-Nair, H. Hagen, and B. Hamann. "Tensor Field Reconstruction Based on Eigenvector and Eigenvalue Interpolation". In *Scientific Visualization: Advanced Concepts*, Dagstuhl Follow-Ups, pages 110–123. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[18] M. Hummel, C. Garth, B. Hamann, H. Hagen, and K. I. Joy. "IRIS: Illustrative Rendering for Integral Surfaces". *IEEE Transactions on Visualization and Computer Graphics*, 16:1319–1328, 2010.

[19] J. Kessenich, D. Baldwin, and R. Rost. "The OpenGL Shading Language Version 4.10", 2010. http://www.opengl.org/documentation/glsl/.

[20] R. Marroquim and A. Maximo. "Introduction to GPU Programming with GLSL". In *Proceedings of the 2009 Tutorials of the XXII Brazilian Symposium on Computer Graphics and Image Processing*, SIBGRAPI-TUTORIALS '09, pages 3–16, Washington, DC, USA, 2009. IEEE Computer Society.

[21] M. Moakher and P. Batchelor. "Symmetric Positive-definite Matrices: From Geometry to Applications and Visualization". In *Visualization and Processing of Tensor Fields*, pages 285–298. Springer Berlin Heidelberg, 2006.

[22] R. M. Rustamov, Y. Lipman, and T. Funkhouser. "Interior Distance Using Barycentric Coordinates". In *SGP '09: Proceedings of the Symposium on Geometry Processing*, pages 1279–1288, Aire-la-Ville, Switzerland, 2009. Eurographics Association.

[23] J. Sreevalsan-Nair, C. Auer, B. Hamann, and I. Hotz. "Eigenvector-based Interpolation and Segmentation of 2D Tensor Fields". In *Topological Methods in Visualization. Theory, Algorithms, and Applications (TopoInVis 2009)*, 2010.

[24] D. Stalling and H.-C. Hege. "Fast and Resolution Independent Line Integral Convolution". In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 249–256, New York, NY, USA, 1995. ACM.

[25] D. Stalling, M. Westerhoff, and H.-C. Hege. "Amira: a Highly Interactive System for Visual Data Analysis". In *The Visualization Handbook*, pages 749–767. Elsevier, 2005.

[26] A. C. Telea. *Data Visualization.* A K Peters, Wellesley, MA, USA, 1st edition, 2007.

[27] X. Tricoche, G. Scheuermann, H. Hagen, and S. Clauss. "Vector and Tensor Field Topology Simplification, Tracking, and Visualization". In *PhD. thesis, Schriftenreihe Fachbereich Informatik (3), Universität*, pages 107–116, 2002.

[28] X. Zheng and A. Pang. "HyperLIC". In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 33–, Washington, DC, USA, 2003. IEEE Computer Society.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

_____ Berlin, den 1. August 2011