Computers & Graphics (2018)

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Real-Time Field Aligned Stripe Patterns

Nils Lichtenberg[a,*], Noeska Smit[b], Christian Hansen[c], Kai Lawonn[a]

[a] *Institute for Computational Visualistics, University of Koblenz-Landau, Koblenz, Germany*
[b] *Visualization group, University of Bergen, Bergen, Norway*
[c] *Computer Assisted Surgery group, University of Magdeburg, Magdeburg, Germany*

ARTICLE INFO

Article history:
Received 23 January 2018
Accepted 29 April 2018
Available online 22 May 2018

2000 MSC: 68U05, 76M27, 68W10

Keywords: Parameterization, Visualization, Computer graphics, Computational geometry

ABSTRACT

In this paper, we present a parameterization technique that can be applied to surface meshes in real-time without time-consuming preprocessing steps. The parameterization is suitable for the display of (un-)oriented patterns and texture patches, and to sample a surface in a periodic fashion. The method is inspired by existing work that solves a global optimization problem to generate a continuous stripe pattern on the surface, from which texture coordinates can be derived. We propose a local optimization approach that is suitable for parallel execution on the GPU, which drastically reduces computation time. With this, we achieve on-the-fly texturing of 3D, medium-sized (up to 70k vertices) surface meshes. The algorithm takes a tangent vector field as input and aligns the texture coordinates to it. Our technique achieves real-time parameterization of the surface meshes by employing a parallelizable local search algorithm that converges to a local minimum in a few iterations. The calculation in real-time allows for live parameter updates and determination of varying texture coordinates. Furthermore, the method can handle non-manifold meshes. The technique is useful in various applications, e.g., biomedical visualization and flow visualization. We highlight our method's potential by providing usage scenarios for several applications.

Declarations of interest: none.

1. Introduction

In surface visualization, there is often a need to visualize additional features of the data directly on the surface. If there is only one value that needs to be shown, color mapping is often employed to provide a qualitative impression of the value distribution over the surface. However, for multivariate data, the need can arise to visualize multiple values simultaneously, and simple color mapping will no longer suffice. Multiple views can be presented in such cases, but this requires mental integration for the viewer. Glyph-based or layering techniques are also able to convey multiple quantities, but may lead to clutter and occlusion [3]. To provide the user with an integrated view of multiple features, advanced visualization techniques such as illustrative visualization can be used to encode additional information. For such techniques, however, preprocessing is often required. This has the unfortunate side-effect that those techniques can no longer be employed to display dynamic changes, and there may be cases where preprocessing is undesirable or even impossible. Furthermore, when relying on precalculation, it is not possible to update any parameters involved at run-time.

*Corresponding author: Tel.: +49-261-287-2774
e-mail: nlichtenberg@uni-koblenz.de (Nils Lichtenberg)

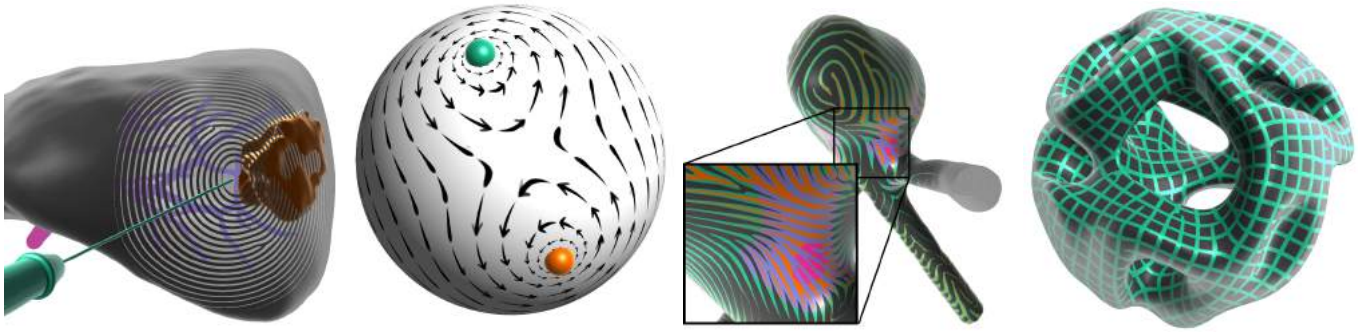

Fig. 1: Our surface parameterization method is able to deal with dynamic changes on-the-fly and can be used in a range of real-time applications. From left to right: (1) selectively discarding fragments achieves a transparency effect, without affecting colors of underlying structures, (2) dynamic flow field visualization, with two spheres representing vortex cores, and arrow textures around them indicating flow direction, (3) visualization of a stress-tensor on aneurysm surface data [1, 2] (when zooming in, the frequency of the pattern increases to allow for more detail), (4) parameterization of the *Hyperball* with a 4-way rotational symmetry field as input.

Therefore, there is a need for a method that is able to provide parameterization of a surface mesh without preprocessing and that can be adjusted on-the-fly.

We propose a technique to parameterize a triangulated surface and generate a global stripe pattern on the surface, based on an underlying tangent vector field. If no vector input is available, principal curvature directions could be computed as a backup strategy. This is also possible in real-time as stated by Griffin et al. [4]. The resulting parameterization can then be used for different visualizations tasks. Existing methods [5, 6, 7] already address this kind of problem. However, these do at most focus on interactivity, while we aim for a real-time visualization, allowing dynamic input properties. Further, our problem formulation is suitable for an optimized reconstruction of the parameterization in the fragment shader. This is beneficial, e.g., if our method is used to generate local texture coordinates. To make our method suitable for real-time applications, we adapt existing approaches and aim for a local solution through local iterative optimization steps. The locality of our approach allows handling of non-manifold surfaces. Also, we can update visualizations and their parameters on-the-fly, for instance driven by dynamic vector fields, or reactive to scene changes resulting from interaction. With this, our main contributions are the following:

- We propose a technique to derive local texture coordinates from tangent vector-fields on a surface mesh, through local iterative optimizations.
- Our technique can be executed in real-time for medium-sized meshes, and thus can be used in visualization of both dynamic meshes, as well as dynamic parameter input.
- We demonstrate the potential of our technique in several usage scenarios from various domains, and compare the performance of our technique both quantitatively and qualitatively to reference methods.

We obtain periodic 1D texture coordinates based on a 1D parameterization aligned to an unoriented vector field. This can be employed for field visualization using a stripe pattern. The parameterization based on two orthogonal vector fields can be used to obtain periodic 2D texture coordinates. These can be used to visualize vector fields or arbitrary scalar properties using different textures or patterns, as we demonstrate in several examples.

2. Related Work

In this section, we examine related work from a technical perspective, as well as from a visualization application perspective.

Surface Parameterization Techniques. Surface parameterization has been intensely researched for a long time [8]. Global parameterization plays an important role in global quad remeshing algorithms in order to find an optimal remeshing across the whole mesh. A survey on this topic is provided by Bommes et al. [9]. Such methods are usually complex to implement, run at most at interactive timings [10] and thus, are not applicable in real-time applications. Jakob et al. [7] proposed a method that relinquishes global optimization, yet is still able to create meshes that align with features on a global scale. This local approach makes their method parallelizable, which makes finding a solution faster by several orders of magnitude. Such techniques define, or get as input, a direction field on the surface, along which the parameterization is aligned. Proper generation of such direction fields is crucial to guarantee mesh quality for these methods. Design of these direction fields has emerged from the above requirements as an additional research area. Details can be found in the state of the art report by Vaxman et al. [11]. Our work uses as input an unoriented vector-field and does not address its further optimization. The methods mentioned so far generate vector fields, or at least require an optimized vector-field as input, and use them in successive steps. The design and visualization of direction fields is often closely coupled to allow for a visual feedback of applied changes [12]. The visualization is often done using line integral convolution (LIC) [13, 14]. However, LIC does only convey the ambiguous orientation of a vector direction $\mathbf{d} \sim -\mathbf{d}$ and cannot be used to display textures. Other methods, like the generation of texture coordinates, utilize vector-valued input to control texture orientation. Then, attention has to be paid to whether the vector field is oriented or non-oriented. Methods that take orientability

into account can be used for a controlled display of orientable textures, but have to take care of visual seams [15, 16, 17], while methods that work on unoriented fields have to rely on symmetric textures [18, 19].

The most important prior art to the work presented here are the position field optimization of the Instant Field Aligned Meshes (IFAM) algorithm by Jakob et al. [7] and the technique for stripe pattern synthesis on surfaces (SPS) by Knöppel et al. [6]. The IFAM algorithm has introduced a local and parallel solution to global parameterization and the patterns that result from applying SPS are globally smooth and applicable for design and texture synthesis tasks. The interpolation scheme by Knöppel allows for a globally continuous pattern away from isolated singular points. Global continuity refers to the property that no jumps in the pattern can be found across the surface (i.e., no seams are visible). More precisely, if a piecewise continuous pattern is given and the pattern is based on a periodic function, the periodicity results in repetitive piecewise continuity across the surface, hence achieving global continuity. In contrast to SPS, our technique finds a locally optimized solution through local iterative optimization steps, which makes it suitable for real-time applications without requiring any precalculation.

Related Visualization Applications. One of the potential application areas for our technique is to employ the generated stripe-patterns as an additional visual encoding channel for multivariate data visualization. Multivariate data is defined in the comprehensive survey by Fuchs and Hauser as information which has an attribute vector for each data item [20]. In the field of multivariate data visualization, Rocha et al. [21] recently proposed a real-time technique to map decals onto surfaces as a new way of representing multivariate data. The sets of images or patterns mapped to the surface are able to represent attributes of the data at the location they are mapped to, and can be used in combination with additional layered visualization elements. In contrast to their approach, we are able to handle dynamic flow patterns in addition to real-time texture coordinate synthesis, since we generate a globally continuous pattern. The work by Schroeder and Keefe [22] specifically caters to time-varying multivariate data visualization by providing an artist with an interface to sketch such visualizations. In their work, they allow artists to sketch illustrative elements that can be used as animated glyphs in a layered 2D visualization. However, their technique is focused on visualization design on a flat 2D surface. In earlier work by Kirby et al. [23], the potential of using illustrative techniques borrowing concepts from painting to visualize multivalued 2D flows was highlighted. Our technique is also able to generate illustrative strokes for flow, but extends to more complex 3D surfaces. Furthermore, we are able to animate these strokes to represent time-varying vector fields. Recent work by Roy et al. [24] use LIC to visualize the sheets of branched covering spaces. However, LIC is not suitable for expressing the unambiguous directionality of vector fields, and thus they require animations to express this aspect.

To the best of our knowledge, ours is the first work to use a globally smooth parameterization for visualization purposes, based on dynamic input data that can be updated in real-time. This concept, w.r.t. to visualization purposes, is inspired by the work by Knöppel et al. [6], who generate a continuous *stripe pattern* on a surface, based on an input vector field. They also present details on the proper visualization of their parameterization results and, e.g., how to obtain texture coordinates from that. Their approach in turn is based on the method to generate a periodic global parameterization (PGP) as described by Ray et al. [5], who focus on re-meshing purposes. The stripe pattern algorithm introduces several changes in order to drastically improve the performance. Jakob et al. [7] were the first to translate the problem addressed by the above mentioned methods to a formulation that allows a local and thus parallel execution of the optimization. However, their CPU implementation is suitable for interactive, but not for real-time performance. Furthermore, the frequency of their periodic pattern is limited by the mesh resolution.

We incorporate ideas and concepts of the above mentioned work and extend these with the goal to come up with an algorithm that allows for parameterization in real-time and is suitable for visualization purposes. We contrast the prior work in the way that we obtain coordinates for orientable textures, how these coordinates can be aligned with the underlying field on a pixel basis and we employ a convergence term for the optimization process. Furthermore, we show a range of application scenarios that can be seen as an inspirational basis for future visualizations tasks, based on dynamic field visualization.

3. Method

To obtain a surface parameterization, we aim to determine a globally smooth stripe pattern on the triangle mesh. The basic idea of the algorithm is to interpret each vertex on the mesh as a sample of an individual wave (i.e, a periodic function). This wave is described by an individual direction, which passes the vertex at a specific phase with a certain frequency. Hence, the input to our algorithm is a vector field that defines the wave directions and a scalar field that defines the wave frequencies. For a globally optimal solution all vertices can be interpreted as samples of the same wave as in Fig. 2. The elongation of each sample on the wave can then be used to generate a periodic stripe pattern.

Notation. For the remainder of this paper, we use the following notation. For a triangulated mesh $\mathcal{M}$, we denote V, E, F as its vertices, edges, and triangles respectively. If $(i, j) \in E$ with $i, j \in V$ then vertex i and j are connected. Similarly, $(i, j, k) \in F$ means that the vertices i, j, k form a triangle. Additionally, we use $\mathcal{N}(i)$ to describe the set of neighbors of vertex i. With $\mathbf{v}_i \in \mathbb{R}^3$, we denote the position of the vertex i in $\mathbb{R}^3$. Furthermore, we define $\mathcal{W}$ as the wave set, which consists of a set of wave directions $\mathcal{D}$, phases $\mathcal{P}$, and frequencies $\mathcal{F}$. Thus, every vertex i is assigned an individual wave $\mathbf{w}_i$ with normalized vector $\mathbf{d}_i \in \mathcal{D}$, which defines the direction of the wave, and a phase $\Phi_i \in \mathcal{P}$ with frequency $f_i \in \mathcal{F}$. Note that we consider the (normalized) direction vector $\mathbf{d}_i \in \mathcal{D}$ at vertex i as an equivalence class with $\mathbf{d}_i \sim -\mathbf{d}_i$. This means that, although this vector has a direction, the solution of a global stripe pattern is independent of this direction and is, therefore, non-oriented.

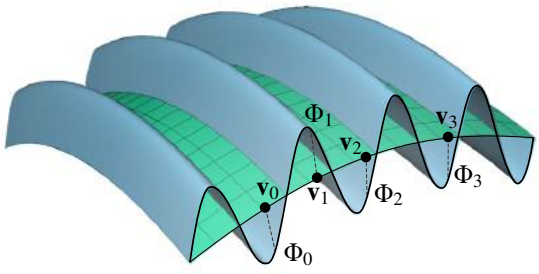

Fig. 2: For an optimal stripe pattern, the vertex phases Φ_i along a surface align with one continuous wave, e.g., the vertex phases sample a single wave along the surface.

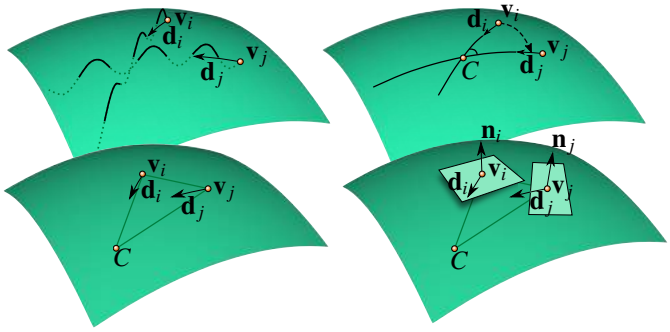

Fig. 3: In the smooth setting, we can find an intersection C of two waves (top left), by the trace of $\mathbf{d}$ along the surface (top right). If the tangent planes of $\mathbf{v}_i$ and $\mathbf{v}_j$ are not the same, we cannot find an intersection of the wave traces in 3D, because $\mathbf{d}$ lies in the respective tangent plane (bottom).

3.1. Basics

The phase shift Φ_{ji}, i.e., the signed difference of the phases from vertex j to i, is computed as follows:

$$\Phi_{ji} = 2\pi \cdot d_{ji} \cdot f_{ji}, \tag{1}$$

where d_{ji} is the aligned distance between $\mathbf{v}_i$ and $\mathbf{v}_j$ and the wave frequency is taken into account by the geometric mean, such that $f_{ij} = f_{ji} = \sqrt{f_i f_j}$. The aligned distance is the distance relative to a common reference point and is used to account for differences in the wave directions $\mathbf{d}_i$ and $\mathbf{d}_j$ (i.e., the divergence of the vector field).

Basically, we cannot directly compute a phase shift (or distance) between i and j, because both represent individual 1D waves along the surface as in Fig. 3 (top, left). If we find an intersection C of the waves (Fig. 3 top, right), we can regard C as the common source of the waves i and j. Thus, the phase shift is computed relative to the intersection point C. In the smooth setting, we would use the geodesic distance to C. If multiple intersection points along the surface exist, we take the closest one. Since we want to process a discrete mesh, a different attempt is made: As shown in Fig. 3 (bottom), the direction $\mathbf{d}$ lies in the respective tangent plane. Consequently, if the tangent planes are not equal, the waves will likely not intersect in 3D space. This is why we create a plane that contains the points $\mathbf{v}_i$, $\mathbf{v}_j$ and $\mathbf{p} = 0.5(\mathbf{v}_i + \mathbf{d}_i + \mathbf{v}_j + \mathbf{d}_j)$ (Fig. 4, left). Then, $\mathbf{d}_i^p$ and $\mathbf{d}_j^p$, which represent the projections of the respective directions into that plane, are used to compute C. As Φ_{ji} represents a signed difference, we have to take the direction of the waves into account:

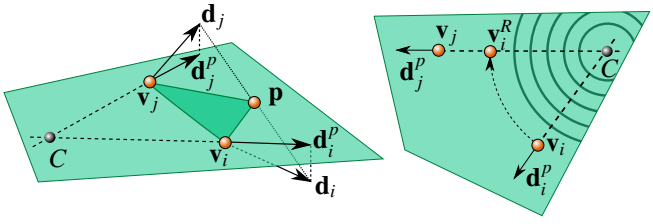

Fig. 4: Computation of the wave intersection C (left). Rotation of $\mathbf{v}_i$ about C for alignment with $\mathbf{v}_j$, where C can be thought of the common source of both waves, indicated by the concentric circle sections (right).

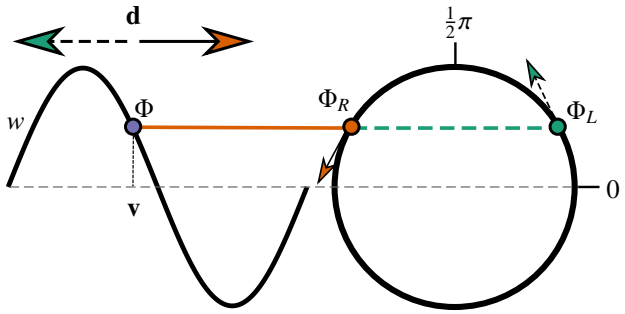

Fig. 5: Interpreting w (left) as a sine-wave has two solutions on the unit circle (right), depending on the direction of $\mathbf{d}$. The elongation at Φ is either increasing or decreasing. In general, if Φ_R represents the phase of the wave traveling to the right, and Φ_L the one traveling to the left, $\pi - \Phi_R = \Phi_L$ holds.

$$d_{ji} = (|C - \mathbf{v}_j| - |C - \mathbf{v}_i|) \cdot \langle C - \mathbf{v}_j, \mathbf{d}_j \rangle, \tag{2}$$

where $\langle ., . \rangle$ denotes the Euclidean dot product. Then, d_{ji} is virtually the distance between v_j and v_i^R (see Fig. 4, right). Eq. 2 cannot handle the non-orientable property of the direction $\mathbf{d} \in \mathcal{D}$ that is depicted in Fig. 5. If $\mathbf{d}_i^p$ and $\mathbf{d}_j^p$ are counter-oriented with respect to their common source C (i.e., one pointing towards and one pointing away from C), we have to take care of the ambiguity of d_{ij}. A simple adjustment to the calculation to solve this problem is described in Section 3.2, Eq.7.

If the phases are consistent on the mesh, then the equation $\Phi_i = \Phi_{ji} + \Phi_j$ would hold for every edge $(i, j) \in E$. In general, this cannot be guaranteed, thus we aim to find the phase set $\mathcal{P}$ that minimizes the following energy:

$$\mathcal{E}(\mathcal{P}) = \sum_{(i,j)\in E} w_{ji} |\Phi_{ji} + \Phi_j - \Phi_i|^2, \tag{3}$$

where w_{ji} is a weight which can be chosen arbitrarily for each edge. We used $w_{ji} = |\langle \mathbf{d}_i, \mathbf{d}_j \rangle|$.

Knöppel et al. [6] formulated the same minimization problem (regarding the energy formulation) and described how they can achieve a globally optimal solution. In this paper, we present a local algorithm, tailored to the GPU. Furthermore, we do not compute the energy based on the phase-shift directly. As previously done by Marcias et al. [25], it is based on the difference of the phases after transforming the phases into 2D Cartesian coordinates on the unit circle:

$$\mathcal{E}_i = \sum_{j \in \mathcal{N}(i)} w_{ji} \| \frac{1}{2}(\varphi_{ji} - \varphi_i) \|^2, \tag{4}$$

where $\varphi_i = (\cos(\Phi_i), \sin(\Phi_i))$ and $\varphi_{ji} = (\cos(\Phi_j + \Phi_{ji}), \sin(\Phi_j + \Phi_{ji}))$. This energy is equivalent to the one in [6], utilizing a single variable for each vertex and each edge. Our global energy is defined as:

$$\mathcal{E}_V = \sum_{i \in V} \mathcal{E}_i. \tag{5}$$

The rationale for using Cartesian coordinates is the handling of mismatching phases, similar to an outlier treatment, and will be described in more detail in Sec. 3.2. The relation between radial and Euclidean distances is depicted in Fig. 6 (left). With our approach, we achieve similar results to the ones presented by Knöppel et al. [6] by finding a local solution Φ_i such that $\mathcal{E}_i$ is minimal. We will later show (in Section 3.6) that only small changes to the implementation are necessary, such that we can process unit cross-fields.

3.2. Local Optimization

Our optimization strategy iteratively adjusts Φ_i to minimize the global energy $\mathcal{E}_V$. First, we initialize Φ_i with a pseudo-random value, assigning a value obtained by using the vertex ID as input for the One-at-a-Time hash, that can be found in the online version of [26]. The minimization is done in parallel for each vertex i and iteration k in two main steps:

1. For every (oriented) edge $(j, i) \in E$, a target phase $\Psi^k_{(j,i)}$ is computed. (Sec. 3.2.1)

2. Φ^{k+1}_i is determined such that $\mathcal{E}^{k+1}_V \leq \mathcal{E}^k_V$. (Sec. 3.2.2)

In step 2 we explicitly do not ask for every vertex that $\mathcal{E}^{k+1}_i \leq \mathcal{E}^k_i$ holds, but we ensure that $\mathcal{E}^{k+1}_V \leq \mathcal{E}^k_V$ holds. This is inspired by *Simulated Annealing* (SA), which randomly allows locally negative updates during optimization [27]. More details about the convergence are given in Sec. 3.3.

3.2.1. Target Phase (Step 1)

The goal in this step is to determine Φ^{k+1}_i for vertex i. As a preliminary value, we set

$$\hat{\Psi}^k_{(j,i)} = \Phi^k_{ji} + \Phi^k_j, \tag{6}$$

as we would expect $\Phi_i = \Phi_{ji} + \Phi_j$ for a perfect stripe pattern. At this point, we have to take the wave directions $\mathbf{d}$ into consideration. If the waves of the vertex i and j are counter-oriented with respect to their common source (Fig. 4), errors occur due to the directionality of the wave function. In particular, $\hat{\Psi}^k_{(j,i)}$ does not correctly reflect the target phase that is necessary to match two such waves. In this case we have to adjust $\hat{\Psi}^k_{(j,i)}$ (see Fig. 4 right and Fig. 5):

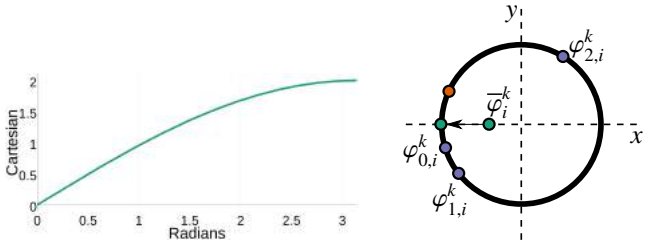

Fig. 6: Comparison of Euclidean and radial distances of two points on the unit circle (left). $\overline{\varphi}^k_i$ is the average target phase based on the Cartesian coordinates on the unit circle ($\varphi^k_{j,i} = (\cos(\Psi^k_{(j,i)}), \sin(\Psi^k_{(j,i)}))$). We can observe that the Cartesian mean phase (green dots) differs from the radial mean phase (mean phase along the arc, orange dot) (right).

$$\Psi^k_{(j,i)} = \begin{cases} \pi - \hat{\Psi}^k_{(j,i)}, & \langle \mathbf{d}_i, \mathbf{d}_j \rangle \cdot \langle C - \mathbf{v}_i, C - \mathbf{v}_j \rangle < 0 \\ \hat{\Psi}^k_{(j,i)}, & \text{otherwise} \end{cases} \tag{7}$$

With this adjustment $\Psi^k_{(j,i)}$ matches the waves of vertex i and j across the edge (j, i) by their elongation rather than by their phase, and thus takes counter-oriented directions into account.

3.2.2. Phase Alignment (Step 2)

In step two, we compute the mean target phase $\overline{\varphi}^k_i$ of vertex i in Cartesian coordinates, which is based on the target phases $\Psi^k_{(j,i)}$ along the (oriented) edges $(j, i) \in E$:

$$\overline{\varphi}^k_i = \sum_{j \in \mathcal{N}(i)} w_{ji} \varphi^k_{j,i}, \quad \text{with } \varphi^k_{j,i} = (\cos(\Psi^k_{(j,i)}), \sin(\Psi^k_{(j,i)})) \tag{8}$$

with $w_{ji} = |\langle \mathbf{d}_i, \mathbf{d}_j \rangle|$. Since in our local approach each vertex should fit as best as it can to its neighborhood with respect to the wave direction, we favor parallel wave directions and prune orthogonal or diverging directions. We average the Cartesian coordinates of the target phases on the unit circle, because this reduces the influence of phases that are far away from the average. This has two reasons:

1. Normalization of the 2D coordinate $\overline{\varphi}^k_i$ within the unit circle does not affect the result of atan2($\overline{\varphi}^k_i$) (Fig. 6, right). Thus, if a phase $\varphi^k_{j,i}$ has drawn $\overline{\varphi}^k_i$ towards the center of the unit circle, that effect is partly compensated by the normalization of $\overline{\varphi}^k_i$. Consequently, such disagreeing phases have less impact on the result, which is an important property of this representation, as it helps connected vertices to share a common phase more quickly.

2. When considering the distance of two points on the unit circle, the relation between the Euclidean and the radial distance is not linear (see Fig. 6, left). In the Euclidean representation distances greater than $\sqrt{2}$ are distinctly compressed, compared the radial counterpart. This reduces the contribution of phases far from $\overline{\varphi}^k_i$, (i.e., outliers get pruned). We compared the presented averaging method with averaging the phases directly in radial space and found that the Cartesian averaging resulted a more uniform pattern, with less bifurcations and lower residue energy $\mathcal{E}_\mathbf{v}$.

To estimate Φ_i^{k+1}, we compute $\overline{\varphi}_i^k$ in Cartesian space. We obtain the following offset to the current phase:

$$\Delta\overline{\varphi}_i^k = \overline{\varphi}_i^k - \varphi_i^k \tag{9}$$

with $\varphi_i^k = (\cos(\Phi_i^k), \sin(\Phi_i^k))$. Finally, we obtain Φ_i^{k+1} by:

$$\Phi_i^{k+1} = \text{atan2}(\varphi_i^k + m_i^k + \Delta\overline{\varphi}_i^k), \tag{10}$$

where m_i^k is a momentum. m_i^{k+1} is updated for each iteration as

$$m_i^{k+1} = s_m \Delta\overline{\varphi}_i^k \tag{11}$$

with $s_m = 0.2$ being the momentum strength. For the first iteration, we set $m_i^0 = (0, 0)$. The momentum can help to keep vertex phases flexible, to avoid a premature local convergence.

3.3. Convergence

To achieve convergence, we ensure that for every iteration $\mathcal{E}_V^{k+1} \leq \mathcal{E}_V^k$ holds. We compute the vertex energy $\mathcal{E}_V^k$ (Eq. 5), after each iteration k, to keep track of the optimization progress. Then, the relative energy improvement

$$\Delta\mathcal{E}_V^{k+1} = \frac{\mathcal{E}_V^k - \mathcal{E}_V^{k+1}}{\mathcal{E}_V^k} \tag{12}$$

is computed for every step. Note that, if a vertex reduces its local energy, the global energy is also decreased, because $|\Phi_{ji}| = |\Phi_{ij}|$ (see Eq. 1). The above assumption holds, as we ensure that updates of neighboring vertices do not interfere, by applying a graph-coloring to the mesh. Details on the graph-coloring are given in Sec. 4. Convergence is detected when $\Delta\mathcal{E}_V^k$ drops below a threshold ϵ_G. The convergence term allows the algorithm to stop as soon as a certain optimization quality is reached, but also requires to keep track of the convergence progress. This is also useful during the hierarchical optimization that is described in the next section.

3.4. Hierarchical Optimization

Our algorithm can optionally employ the hierarchical structure presented in [7]. In this section we recap the purpose of this hierarchy for completeness. In the local approach, the phase of a vertex is only optimized with respect to its direct neighbors. In order to find a solution that is globally satisfactory, the information of a vertex has to propagate across the mesh, which is problematic for high resolution meshes. This propagation of information can be speeded up by running the optimization in multiple resolutions of the mesh. A coarse resolution will quickly converge, while a fine resolution will take small features of the mesh into account. Thus, we optionally utilize the multi resolution hierarchy depicted in [7], to obtain a set of meshes that approximately halves the number of vertices with each coarser hierarchy level. We can start our optimization in a coarse level and propagate the results to the next finer level until the original mesh is reached. We switch levels as soon as the current level has converged as described in the previous section. However, it has to be mentioned that the input wave directions and frequencies in the finest level are consecutively smoothed (area weighted average of merged vertices) in coarser levels. If the smoothed properties differ greatly from the original input, the results of coarse hierarchy levels will not properly represent the final result. Hence, the hierarchy is less feasible if the input properties are not smooth.

3.5. Texture Coordinates

If we want to render a periodic texture or pattern with the values given in $\mathcal{W}$, we have to compute a per-pixel phase Φ_P in the fragment shader. Each fragment's phase is determined by the vertex phases Φ_i, Φ_j, Φ_k, where $(i, j, k) \in F$ are the vertices of the triangle generating the fragment. We cannot simply interpolate the vertex phases linearly, due to the periodicity of the wave function [6]. This is important if the actual phase shift between two vertices along a wave is larger than 2π. In these cases we have to incorporate the frequency f. Next, we describe our interpolation method, which takes f into account.

Per-Fragment Phase. To compute the fragment's phase Φ_P, we need information about the wave direction $\mathbf{d}_P$ at the three-dimensional coordinate of fragment P.

$$\mathbf{d}_P = \sum_{m\in\{i,j,k\}} s_m \cdot b_m \cdot \mathbf{d}_m, \tag{13}$$

where b_m is the barycentric weight of each vertex to the fragment and s_m takes care of the direction of $\mathbf{d}_m$. If the wave of a vertex m in a triangle is counter-oriented relative to the other two, we set $s_m = -1$, otherwise we set $s_m = 1$.

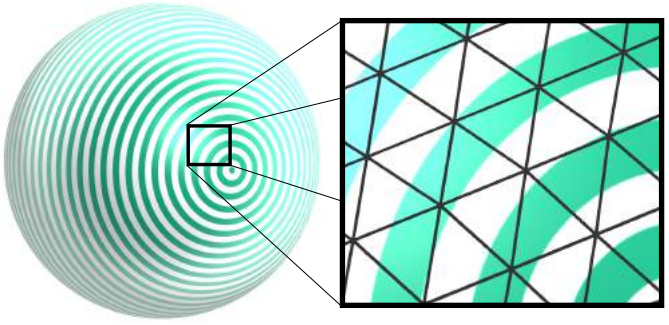

Fig. 7: Wave pattern on a sphere. The close-up shows how our interpolation method is capable of aligning each individual fragment with the wave pattern. The direction of the pattern visibly changes even inside individual triangles, reflecting the vector field divergence.

This is required to avoid null vectors and to consider the equivalence $\mathbf{d}_i \sim -\mathbf{d}_i$. The adjustment through s_m is crucial for the proper display of textures (see Fig. 5). Note that this can only be applied to triangles that can be oriented in a general direction. If this is not the case, we only make sure that the pattern is continuous across the triangle edges. This is achieved by using a barycentric subdivision of the triangle as described in [6].

With the 3D-coordinate of the fragment and $\mathbf{d}_P$ we can obtain a fragment phase $\bar{\varphi}^P$, similar to Eq. 8. In this case w_{ji} is replaced by the barycentric weight, the vertex i is replaced by the fragment, and the neighboring vertices j are replaced by the three vertices, that generate the fragment. Finally, we compute the fragment phase:

$$\Phi_P = \text{atan2}(\bar{\varphi}^P). \tag{14}$$

The computation of individual fragment phases yields a pixel perfect alignment of the wave pattern, as shown in Fig. 7 and the bending arrow texture in Fig. 1 (center, left). This is an advantage over a linear interpolation as Fig. 17 (left) shows. The IFAM source code uses linear interpolation to reconstruct the fragment parameters. This leads to visible linear segments in curved regions of the parameterization and limits the pattern frequency by the mesh resolution. The interpolation method used in [6] can handle the pattern frequency independent of the mesh resolution, but still generates linear segments inside triangles. Note that our fragment interpolation is only feasible because it is equivalent to the interpolation of vertices during the optimization process. Differing interpolation approaches would lead to distortion of the pattern.

2D Coordinates. Computing two wave patterns U, V, based on orthogonal vector fields allows to display 2D textures. For this, we map $\Phi_P^U, \Phi_P^V \in [-\pi, \pi]$ to a range of $[0, 1]$ and use this value for texture access. Note that we need to take $\mathbf{d}$ into account again. Even if the peaks and troughs of counter-oriented waves match after our optimization, $\mathbf{d}$ still affects the texture access. Through the application of s_m in Eq. 13, two neighboring triangles can be counter-oriented with respect to their texture access. This happens if vertices on an edge (i, j) are counter-oriented, and for one triangle i is flipped, while for the other triangle j is flipped. Then, the texture coordinates along the edge are flipped as well. To obtain matching texture coordinates at sites of counter-oriented waves, we simply add $\pi/2$ to each fragment phase and then compute the texture coordinates. This matching only applies to textures that are symmetric along both axes (i.e. with a 180° symmetry). As in Fig. 5, the direction in which we sample a texture depends on the wave direction. This does not account for textures that are symmetric along both axes.

3.6. Cross-Fields

As described in the previous paragraph, we can use two orthogonal vector fields to generate 2D coordinates for, e.g., texture access. Here, we shortly describe how the implementation can be optionally changed to handle two orthonormal vector fields as a cross-field with rotational symmetry. In our problem formulation the wave direction $\mathbf{d}$ was treated as an equivalence class $\mathbf{d} \sim -\mathbf{d}$, hence two orthonormal vector fields resemble a unit cross-field in the notation by Vaxman et al. [11]. We can extend the implementation to manage unit cross-fields by optimizing 2 linked stripe patterns (see Fig. 1, right). In this case, each vertex $\mathbf{v}_i$ references 2 directions $\mathbf{d}_{i,r}$, with corresponding phases $\Phi_{i,r}, r \in \{0, 1\}$. The directions $\mathbf{d}_{i,r}$ are given by rotations of an input direction $\mathbf{d}_i$ about the normal of $\mathbf{v}_i$ by $r/2\pi$. During optimization, the individual waves are not isolated. Instead, the mean target phase $\overline{\varphi}_{i,r}$ for a given $\Phi_{i,r}$ (Eq. 8) is computed after removing the ambiguity (i.e., the rotational symmetry) of the cross-field: For each neighbor $j \in \mathcal{N}(i)$ we obtain $\Psi_{(j,i)}$ based on $\Phi_{j,m}$ and $\mathbf{d}_{j,m}$. Here, $m \in \{0, 1\}$ is chosen such that $|\langle \mathbf{d}_{i,r}, \mathbf{d}_{j,m} \rangle|$ is maximized. This has the effect, that $\overline{\varphi}_{i,r}$ is computed by taking the neighboring waves into account, that minimize the orientational difference to the direction $\mathbf{d}_{i,r}$. Note that this modification is optional. For the display of textures, two separate orthogonal vector fields are better suited, because they allow textures of 180° symmetry. A cross-field for example (that represents a 90° symmetry) is suitable for textures of 90° symmetry.

4. Implementation

We have defined our optimization method as a highly parallelizable problem. In this section, we describe our implementation of the algorithm on the GPU.

Generally, we would like to compute as many vertices as possible in parallel. Care has to be taken in terms of memory consistency, because the update for each vertex per iteration depends on the state of the neighboring vertices. By applying a graph coloring C to the mesh, we can find disjoint subsets $c \in C$ of the mesh, such that there is no edge between any nodes in c, i.e., neighbored vertices do not share the same color. We can then iterate over the subsets $c \in C$ and update all nodes belonging to c in parallel. In this case we do not need to take care of race conditions or memory consistency. The pipeline consists of four main steps:

1. Initialization
2. Graph Computation
3. Phase Evaluation
4. Pattern Extraction

Step 1. We initialize the optimization by setting vertex phases to an initial pseudo-random value and calculate per vertex properties, if necessary. Per vertex properties might for instance consist of individual frequencies, or modifications to the underlying vector field. For example, the frequency might change based on the distance of a vertex to a dynamic reference point or the vector field might change with respect to time resolved vector field data. This step is performed in a vertex-shader. We do not apply further modifications to the data in order to achieve a most appropriate visualization of the input.

Step 2. According to Jakob et al. [7] and with the neighborhood information obtained from the mesh topology, we compute a graph-coloring such that two neighboring vertices never share the same color. For this we implemented the Jones-Plassman-Luby algorithm described in the work by Naumov et al. [28]. To enable a vertex to access information to its neighbors, we store the vertex IDs of all neighbors per vertex. All data is stored in Shader Storage Buffer Objects (SSBO), so that arbitrary reads are possible.

Step 3. We invoke a render pass, consisting of a vertex-shader, for each color of the graph and only process vertices of that active color. Like this, an active vertex can read data from its neighbors and write its updated data, without interfering memory access with its neighbors. Such a division into disjunctive sets has also been used in the work by Choong et al. [27] within

Fig. 8: Branch properties: Area around undefined phase (left), angular deviation of the parameter gradient and the vector field, where a full red location represents a deviation of 90°(center), and angular deviation larger than 5°(right).

the context of a parallel SA implementation. At first, the mean target phase $\overline{\varphi}_i^k$ is computed as in Eq. 8. Along with $\overline{\varphi}_i^k$ we can compute the vertex energy $\mathcal{E}_i^k$, based on the state of the previous iteration. $\mathcal{E}_i^k$ is summed up over all vertices i using an atomic float counter. Thus, the convergence is checked against the energy of the last but one iteration and we can immediately obtain and apply Φ_i^{k+1} as in Eq. 10.

Step 4. After the optimization has stopped, a final render-call determines the per fragment phase Φ_P as in Eq. 14 and extracts a pattern or texture coordinates to either display a stripe pattern or a texture. The GPU storage size which our SSBOs require, depends on how many vector fields we want to process simultaneously and on the maximum number of neighbors per vertex.

Our current implementation reserves memory for $4 + 7V + N$ components per vertex for V vector fields, if the maximum number of vertex neighbors is N. This contains: the world coordinate (3), the number of neighbors (1), the neighbor IDs (N), the direction $\mathbf{d}$ ($4V$), the phase Φ ($1V$), the frequency f ($1V$) and the energy $\mathcal{E}$ ($1V$). In this very straight-forward implementation, the color-graph allows us to neglect any memory barriers, other than separate render calls. A positive side-effect is that colors that are processed later, can already use the updated information of neighboring vertices. This effectively speeds up the propagation of information across the mesh.

Further, while a graph subset $c \in C$ is processed, we can be sure that the neighbors of the nodes in c do not change. Thus, a positive update of the vertices in c cannot be undone by parallel execution of their neighbors. However, the graph-coloring reduces the grade of parallelism, but if the GPU is still working at capacity, even if only vertices of single colors are computed, we expect the overhead to be at a minimum. Another optimization we can utilize depends on the quality of the graph-coloring. We utilize a naive parallel graph coloring approach and found that the last third of assigned colors covers only about 1.5% of the graph. With that in mind, we process the vertices of the last third of colors only every second iteration, to save computation time. We found that the influence on the result's quality is negligible in that case.

Branches. During the optimization, our algorithm automatically inserts branches to the stripe pattern. Such locations are also known as singularities of a positional symmetry field as in [7]. For completeness, we provide a short recap of this topic. A branch adds or removes a stripe segment to ensure an isometric spacing of the stripes, adapting to the surface morphology or to vector field divergence. These branches occur at locations where our interpolation (Eq. 8) results in an undefined phase (i.e., in the center of the unit circle in Fig. 6 (right)) for a given triangle of the input mesh. Such a location is marked in red in Fig. 8 (left). We can find triangles that contain such a singularity by using the properties $\mathbf{w}_i$ of each of the triangle's vertices, and estimating the presence of such an undefined phase. Generally speaking, the less branches a parameterization contains, the higher its quality. The minimum number of branches depends on the mesh morphology and pattern frequency. It can be observed that the input vector field cannot be correctly depicted by the stripe pattern in branch regions. In the optimal case, the gradient of the parameterization (i.e., $\nabla\Phi$) is parallel to the vector field. The angular deviation of the gradient and the vector field are shown in Fig. 8 (center and right). Regions where the parameterization does not properly agree with the underlying vector field are limited to branch regions. Away from branch regions, the stripe pattern is well suited to represent the vector field in a precise manner.

Dynamic Input. The implementation can easily handle dynamic input (e.g. changing frequency, vector field direction or vertex position). If one of these properties changes from frame n to frame $n + 1$, the output $\mathcal{P}$ at n can be used as input for $n + 1$. Very few iterations of our optimization are necessary, to adjust Φ with respect to the dynamic input change. The number of iterations required to achieve a visually smooth update depends on the ratio of the frame rate and the rate at which the dynamic property changes. Rapid property changes require a higher number of update iterations. We found that updating $\mathcal{P}$ with 1-3 iterations per frame yields good visual results, while a higher number adjusted $\mathcal{P}$ too fast, resulting in jittery movements of the visualized pattern.

5. Results

In this section, we present several usage scenarios in which we applied our technique to surface meshes from both real and artificially generated datasets. Additionally, we provide a qualitative and quantitative performance comparison to the work of Knöppel et al. [6] and Jakob et al. [7].

5.1. Usage scenarios

Our technique can be employed in a range of scenarios in which information needs to be visualized on surface meshes on-the-fly, that is based on scalar or tangent vector input. It can also be utilized in the course of multi-variate data visualization to some extent, because the patterns generated by our method can be employed as an information channel in addition to color or glyphs, for instance. Here, we present several concrete usage scenarios for our technique based on biomedical and vector field visualization.

Surface Sampling. Our method can be employed for structured surface sampling. After the optimization process, we can find locations on the surface with $\Phi = n \cdot 2\pi$, $n \in \{0, 1\}$, which can be thought of as the peak of a cosine wave. If two orthogonal stripe patterns are given, the intersection of their wave representations' peak locations sample the surface in a regular pattern.

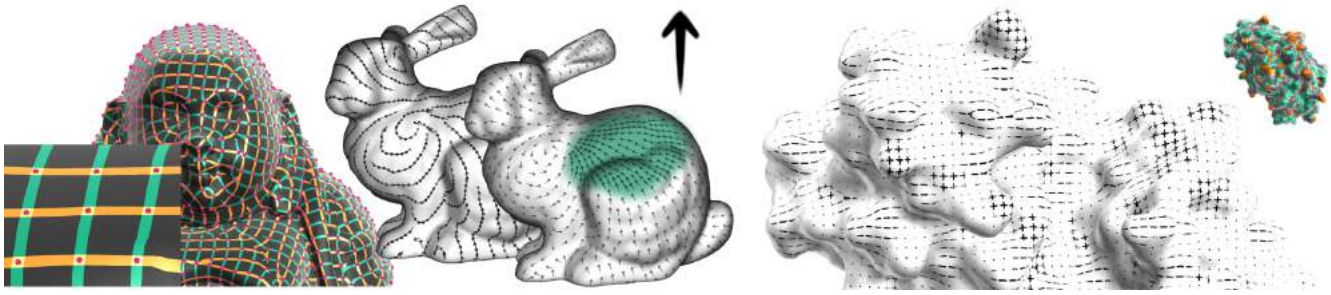

Fig. 9: Magenta dots mark detected sample points (left). Anisotropic (center, left) and importance (center) sampling combined with arrow decals (using decal maps [21]) placed by our sampling technique visualize a vector field. The sample density is increased in the highlighted area. Electrostatics of a molecule (right): Comparison of a color map and user defined texture patches that represent positive (orange, +), negative (green, -) and neutral (gray, ·) charges.

The search of these samples can be implemented in the geometry shader, since for each triangle, the peak locations can be estimated based on their vertices' phases Φ, directions $\mathbf{d}$ and frequencies f. In Fig. 9 (left) the green and orange stripes show locations around wave peaks of orthogonal vector fields. The intersections of these peaks sample the surface in a regular pattern, with respect to the wave directions and frequencies. An anisotropic sampling has been achieved by simply using different frequencies for two orthogonal stripe patterns (Fig. 9, center, left). Similar to the Poisson-sampling proposed by Corsini et al. [29], we can introduce an importance sampling, by increasing the frequency in more important regions (Fig. 9, center). More sampling-based visualizations are shown in Fig. 10. For the bunny on the left, we generate tangent vector field-aligned quads, based on the sampling. These quads are then rendered with an artistic hatching texture, allowing us to draw across the original mesh boundary. In Fig. 10 (right) a vector field is visualized using arrow glyphs. An additional scalar field is used to modulate the pattern frequency, which can be employed to draw the glyphs in varying size. In this context, we think of applications in tensor field visualization, where the glyph size and spacing is controlled by the eigenvalues of the tensor. The adaptive sampling is then able to resemble a sort of glyph packing [30].

Illustrative Biomedical Visualization. Our technique can be employed for illustrative biomedical visualization applications. As shown in the work by Ritter et al. [31], illustrative vascular visualization methods can be used to enhance spatial perception for complex vascular structures. By using texture to encode shape and topology, the color channel is left free to encode additional information. Lawonn et al. [32] followed up on the work by Ritter et al. by developing a visualization technique for 3D vascular models in the liver, in which they used hatching styles to encode distances [31]. While their approach required preprocessing in the form of streamline calculation based on curvature, our current technique can be employed for similar purposes without any offline calculation. Curvature information, for instance, can be computed in real-time as well [4].

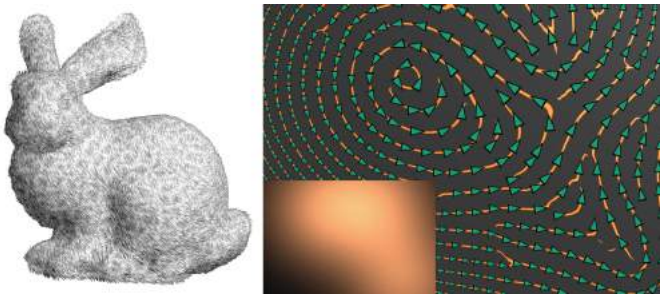

Fig. 10: Artistic hatching achieved by rendering surface aligned quads with a stroke texture (left). Visualization of a vector and scalar field (right, the scalar field is depicted in the inset).

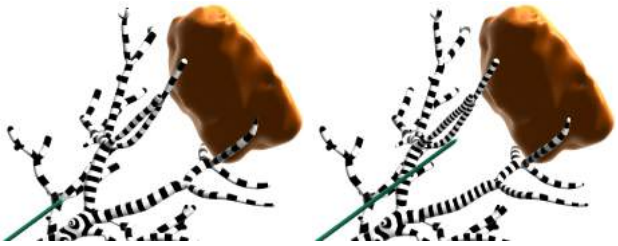

Fig. 11: Liver vasculature is visualized using our technique, with a tumor displayed in brown and a surgical instrument indicated in green. The hatching frequency dynamically updates to encode the distance of the vessel to the needle, when the needle tip is further away (left), or closer to a vessel (right).

Since our technique is able to dynamically adjust the hatching frequency and stroke width, we can adjust our visualization on-the-fly to take novel information into account, such as changes to the scene resulting from interaction. In a surgical guidance context, hatching stroke frequency for instance could be based on the current distance to the camera [31] or one of the surgical instruments employed during an operation, as can be seen in Fig. 11. The surfaces in this figure were reconstructed from a clinical CT dataset. By varying the hatching frequency based on instrument proximity, as the instrument gets closer to the vessel, the stripe size adjusts proportionally, such that the line width is approximately 1/10 of the distance to the needle. The hatching style and color are then still free to encode other information, for instance to use pseudo-chroma-depth to enhance depth perception. Furthermore, an interactive focus-and-context visualization can be generated using dynamic cutaways via a *binary transparency*. This effect is similar to the *screendoor focus* in [33], though our pattern follows the mesh surface, preserving geometrical features. This can be used to provide a view on nested structures, for instance the vasculature and tumor, which reside inside the liver, without altering the color perception of the structures within (see Fig. 1, left).

Several techniques address the generation of hatching

strokes, based on dynamic properties, such as focus-and-context driven line generation [34] or apparent ridges [35]. Our technique is able to generate strokes of adjustable width and spacing and allows dynamic stroke directions without precomputation. An example, along with a comparison to existing techniques by Lawonn [36] and high quality hatching by Zander [37] is given in Fig. 12. The example shows that the parameterization can be used to draw locally varying hatching strokes, or to render a dashed silhouette, to obscure less important parts of a mesh.

When introducing color to the stripe pattern, we can display vector and scalar fields in a combined view, as done in Fig. 1 (center, right). Here, we show the first eigenvector of a stress-tensor on the vessel surface of an aneurysm. The first eigenvector is represented by the stripe pattern. The first two eigenvalues are shown in green and orange. Purple and magenta highlight areas, where the respective eigenvalue exceeds a user-defined threshold. The space between the strokes depends on another user-defined threshold and provides information about the global relation of the tensor magnitudes. I.e., thin and thick strokes represent low and high eigenvalues, respectively. The pattern frequency can be dynamically adjusted, e.g., based on the target object's distance to the camera. As the user zooms in, the frequency increases. Our algorithm's ability to dynamically update the pattern yields a smooth transition while zooming. A similar visualization of surface stress, that requires precomputation, can be found in Meuschke et al. [38]. However, single stream-lines that are employed to depict the tensor data in their work, may overlap and therefore impair the perception. The idea of visualizing tensor fields with orientable patterns is

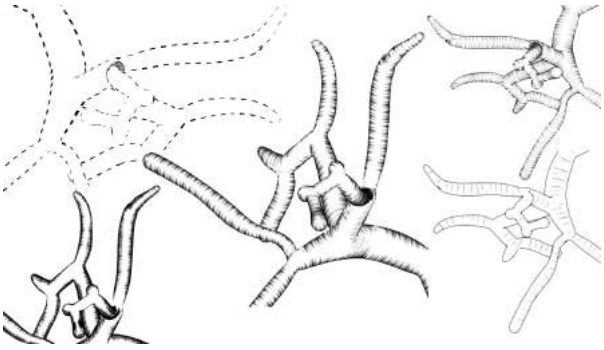

Fig. 12: Illustrative visualization of vascular structures of the liver: Locally varying hatching strokes (center), cross hatching (left, bottom) and dashed silhouette (left, top). Comparative visualizations using the ConFis method by Lawonn [36] (right, top) and high quality hatching by Zander [37] (right, bottom).

also implemented in the work by Auer et al. [39].

Visualizations of biological information can also benefit from illustrative visualization techniques [40, 41]. For example, molecular visualization often aims to abstract context information, or has to deal with occlusion [42], or time-varying simulation data. In Fig. 9 (right) we visualize electrostatic properties of a molecule. While the color map is a common way to do so, we can use our periodic texture coordinates for a space-filling mapping of texture patches to the surface. This way, the color channel is left free for representation of other properties. In general, there are many cases in which biomedical multivariate or dynamic data needs to be visualized on meshes where preprocessing is not possible and/or not desirable.

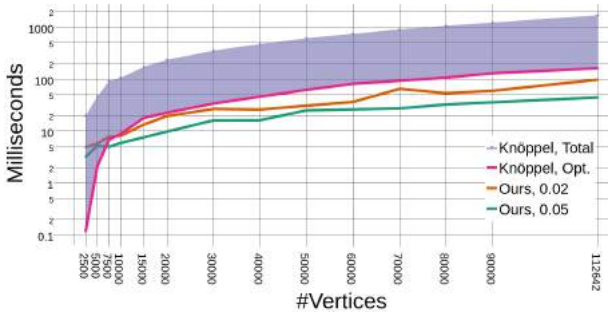

Fig. 13: Timings for two orthogonal parameterizations of our approach compared to SPS, applied to the *Horse* mesh at different resolutions. The lower bound of the filled line displays the timings for one optimization iteration in [6] Alg. 6. The upper bound represents the total time their algorithm takes for both, building up their matrix representation ([6] Alg. 4) and computing one optimization iteration. Numbers are given for our optimization for $\epsilon_G = 0.05$ and $\epsilon_G = 0.02$ in log scale. The visual results are shown in Fig. 15.

Vector Field Visualization. Besides handling cases in which we visualize dynamically changing scene information at run-time, such as updates based on instrument location, we are also able to handle dynamically changing vector fields on the surface itself. This is useful for instance in flow visualization, specifically when visualizing unsteady flow, i.e., flow which is changing over time. In the flow visualization survey by McLoughlin et al. [43], it was stated that unsteady flow is more challenging to visualize, and animation is a natural way of representing this time-dependent flow. In Fig. 1 (center, left), we provide an example of visualizing an artificially generated time-dependent vector field on a surface. The vortex cores, represented by the orange and blue spheres in the figure, move over time and influence the vector field on the surface. Close to the vortex cores, we visualize the vector field with an animated arrow texture, while further away we animate the field with a less salient texture to emphasize the flow around the vortices. The flow can be animated, by shifting the texture access based on a periodic time parameter. The speed of the flow is then represented by the pattern frequency. For example, texture patches in regions of high frequency move slower, because they cover a smaller region in 3D space during a constant time period. Due to the globally continuous pattern that is generated every timestep, we are able to update changes to the flow on-the-fly. We further visualize a synthetic vector field on the bunny model in Fig. 9 (center, left), utilizing the Decal-Maps method by Rocha et al. [21]. The decal positions are defined by our sampling method. The more regular distribution of our samples might be an advantage when sample positions are used for flow visualization. With our method, the sampling itself is able to resemble the flow direction, because the samples are found along wave peaks. It has to be noted that our technique - since it is an optimization - can only represent an approximation of the underlying vector field.

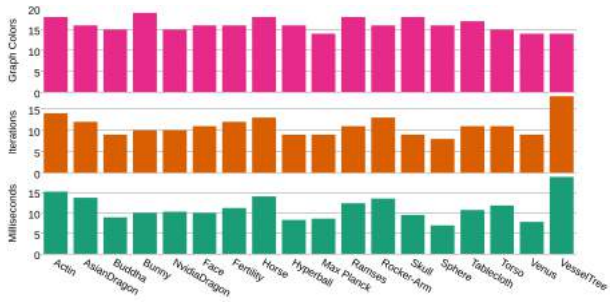

Fig. 14: Timings of our approach for different models with 30k vertices and 60k faces each. The number of iterations correlates strongly with the runtime, indicating that GPU work-load is distributed similarly across different meshes. The number of colors in our color-graph does not correlate with the timings.

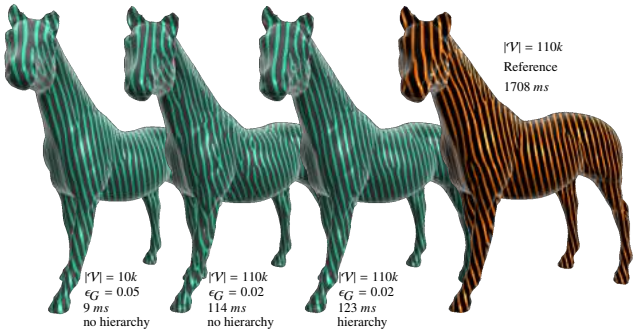

Fig. 15: Comparison of our method and SPS (Reference, after one iteration). Note that the timings were measured while computing two orthogonal patterns, but we show only one pattern to avoid visual clutter.

5.2. Performance

To assess the performance of our algorithm, we generated 2D-stripe patterns for several meshes of various sizes, ranging from 2.5k to 110k vertices per mesh. The performance tests were executed on a desktop computer environment with a 4.00 GHz i7-6400 processor, a GTX-1070 GPU and 16GB RAM. Since the problem that we solve is very similar to that targeted by SPS [6] and IFAM [7], we provide a quantitative comparison of our technique with their methods. To make the comparison with Knöppel et al. [6], we compiled the original code provided by the authors, together with the SuiteSparse and CHOLMOD packages from the Ubuntu package-manager. Both algorithms get a vector field as input, that we obtain by projecting the same global vector into the tangent plane of each vertex. Depending

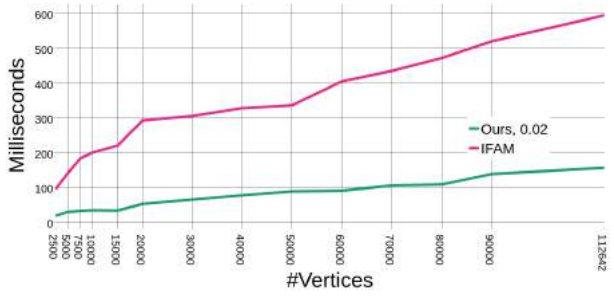

Fig. 16: Timings of our algorithm (with $\epsilon_G = 0.02$) compared to the position field optimization by Jakob et al. [7] using a cross-field on the *horse* mesh at different mesh resolutions.

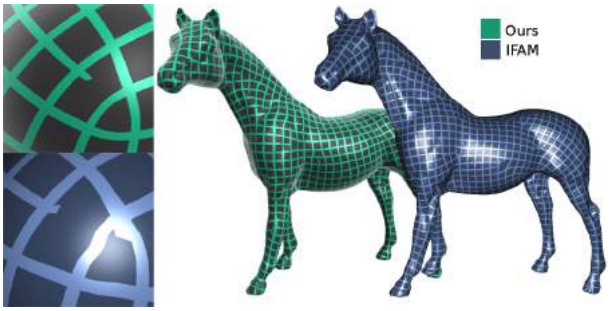

Fig. 17: Visualization of similar parameterization results of the same sphere model (left). The *horse* mesh (225k faces) parameterized with a cross-field by our method and the method by Jakob et al. [7] (right). Note that the cross-fields are not the same, since these are based on a random seeding.

on the optimization parameter ϵ_G (see Sec. 3), we can optimize for speed, with lower quality pattern generation results, or optimize for quality, at the cost of computation speed. The stripe pattern algorithm can be tuned in a similar way. Their computation relies on an energy-matrix build-up, and an iterative optimization of this matrix ([6], Alg.4 and Alg.6). The number of optimization iterations in their publicly available implementation is 20. We found that the results after 4 iterations was visually difficult to distinguish from the results after 20 iterations. Nevertheless, we chose to conduct our comparison with their results after only one optimization iteration, to obtain the lowest possible timings and to account for probable real-time ability. For the comparison with the IFAM algorithm, we use the code that was published by Jakob et al. [7]. Here, both algorithms find a solution based on a cross-field input using 10 hierarchy levels.

Quantitative Perspective. The plot in Fig. 13 shows that our method computes stripe patterns faster than the globally optimal stripe pattern, for sufficiently large meshes. Very small meshes do not benefit enough from the parallelism of our approach. The presented timings were achieved without using the hierarchy levels mentioned in Sec. 3.4. We did this, because the hierarchy is not generally feasible. In cases where we update the parameterization frame-by-frame and use the result of the previous frame as input, we cannot take advantage of the hierarchy. Re-using results in coarser hierarchy levels for consecutive frames would not yield a frame-coherent visualization at the finest level, since the propagation to the coarser levels would smooth the input. We compare the timings our algorithm takes to converge, with the total time of the stripe pattern method, which includes building up an energy-matrix representation and optimizing that energy over one iteration. We do so, because their energy-matrix is computed based on the mesh morphology and the target frequency or orientation per vertex. Thus, for morphology changing meshes, or dynamic input, the timings displayed in Fig. 13 represent the minimum timings per frame. Even if the energy-matrix build-up would not be completely recomputed for each frame, our algorithm still completes faster, than one optimization iteration of the reference algorithm. Convergence of our algorithm depends on the threshold ϵ_G. To account for this, we show timings for two configurations.

For dynamic input, our approach is superior to the stripe pattern algorithm. When updating an existing parameterization with dynamic input, 1-3 iterations of our algorithm are sufficient and thus only a fraction of the timings given in Figures 13 and 14 are required. The plot in Fig. 16 compares our method with the parallel CPU optimization by Jakob et al. [7]. It shows that our GPU approach scales better with the number of vertices and is able to complete the parameterization significantly faster. From Fig. 14 we can observe that our algorithm's performance is drastically mesh-dependent.

Qualitative Perspective. If computation speed is less crucial, ϵ_G can be decreased. As processing time increases, the quality of the output pattern increases as well. However, we can observe in Fig. 15 (center, left) that the quality is nonetheless limited for high resolution meshes. In comparison we show the same model (Fig. 15, center, right), after the optimization using 10 hierarchy levels. The processing time has slightly increased, but the visual outcome is close to the optimal reference (Fig. 15, right). Also, in this example, the number of branches for the hierarchical optimization has been reduced by about 33% compared to the non-hierarchical one. Similar quality is achieved for a lower resolution of the mesh, without using the hierarchy (Fig. 15, left). A visual comparison of to the method by Jakob et al. [7] can be found in Fig. 17.

Robustness. Our method is robust against noise and incomplete meshes (see Fig. 18) and can as well be applied to non-manifold meshes. As long as $\mathcal{D}$, $\mathcal{F}$ and the neighborhood $\mathcal{N}$ are defined, the algorithm is independent of any topological restrictions. It would even run on point clouds, but we leave this for future work.

Limitations. The most significant limitation of the proposed algorithm is that it does not scale well with mesh resolution. We can address this problem with the hierarchical optimization, which allows us to parameterize large meshes at the cost of building up the hierarchy. However, the hierarchy is only applicable if the input parameters on the original mesh are already smooth. Furthermore, if we apply a frame-by-frame update of the parameterization to adapt to dynamic input, we also have to renounce the hierarchy. In such a case it might still be feasible to use the hierarchy for an initial parameterization, which can then be modified in consecutive frames. Processing dynamic input only on the level of the original mesh is crucial to maintain a frame-coherent appearance of the result. Regarding the results of our quantitative evaluation, we can state that the proposed algorithm is capable of processing meshes of up $70k$ vertices with approximately 30 fps. Beyond that, interactive rates are still possible but the parameterization quality is significantly impaired if not using the hierarchy (see Fig. 15). However, the stated mesh size is appropriate for the proposed visualization tasks and especially for applications in the medical domain. Further, it might be desirable that the stripe pattern aligns with sharp features of the mesh. This is currently not supported but could be addressed by incorporating the extrinsic energy formulation in [7].

We assume that our implementation, which mainly resides in the vertex-shader, is straightforward on the GPU and sufficiently proves the real-time capability of our method. With the presented structure, there is no need for further synchronization of memory access. Contrarily, the CPU is likely to be a bottleneck here, since the GPU and CPU have to communicate for each render-call. During the optimization iterations, the number of calls amounts to the number of iterations times the number of colors in the color-graph. An implementation using CUDA, OpenCL, or compute shaders, which offer more flexibility and manually defined memory barriers, might improve the computation times for our algorithm.

6. Conclusion and Future Work

We present a technique for the parameterization of surfaces that can be applied to surface meshes in real-time without time-consuming preprocessing steps. Our method generates a stripe pattern on arbitrary morphology on-the-fly. The work incorporates several ideas of existing work [5, 6, 7]. We adopt the representation of the periodicity in our parameterization through waves (i.e., the sine and cosine of the parameter space as in PGP), since their formulation is most intuitive in our opinion. The PGP algorithm parameterizes triangles and reconstructs per-vertex parameters from them. Hence, each vertex is initially processed n times, with n being the valence of the vertex. In our method, vertices are parameterized directly, with respect to their neighbors, making it more suitable for parallel execution. This adopts the energy from SPS [6], defining one parameter for each vertex and each edge in a mesh (instead of parameters per vertex per triangle as in PGP). Knöppel et al. have introduced a sub-triangle interpolation scheme, to allow a pattern frequency higher than the mesh resolution. We use an extended interpolation method that is able to resemble the changes of the parameterization (especially with respect to vector field divergence, see Fig. 7) within a triangle on a fragment basis. The interpolation method takes the vector field directions into account to compute the distance between two points on the surface. This approach is incorporated into our optimization process, to make the parameterization compatible with the interpolation method. A by-product of this action is, that we can obtain periodic patterns on the surface, that have an arbitrarily higher resolution than the mesh. This stands in contrast to IFAM, where the pattern resolution has to be lower than the mesh resolution. We further use a convergence term for our optimization process, whereas IFAM uses a fixed number of optimization iterations.

The performance evaluation and comparison to the reference method by Knöppel et al. [6] and Jakob et al. [7], reveal that aesthetically pleasing and accurate results can be generated under real-time conditions. We bring the topic of periodic parameterization to the context of data visualization, as shown by multiple examples that address different tasks. Besides animated textures on static surfaces, our technique is also capable of handling morphological changes in the surface mesh, and can be used for animated meshes. The computation for the *Ramses* model shown in Fig. 19, containing $826k$ vertices, took 400 *ms* using 10 hierarchy levels. Hierarchies which support dynamic

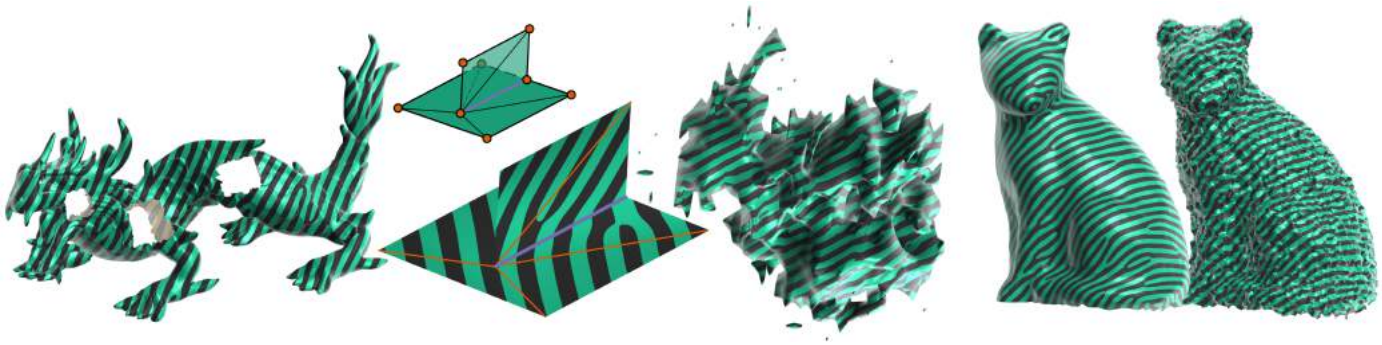

Fig. 18: Our method can process incomplete (left) and non-manifold (center, left) meshes. The low quality tumor segmentation with unconnected parts (center, right) can also be handled, as well a noisy meshes (right).

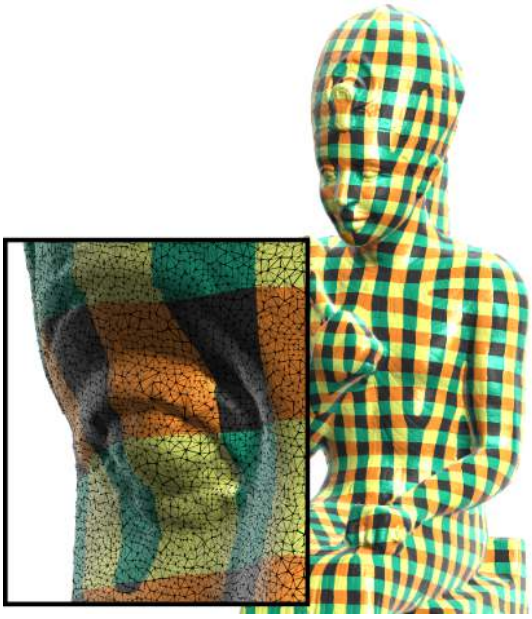

Fig. 19: The *Ramses* model with 826*k* vertices, parameterized in 400 *ms*.

changes of the input have been proposed by Schertler [44]. Thus, future work should address the utility of the hierarchy for dynamic input in the context presented here, as to overcome the current limitations of this work.

Further, we would like to analyze the behavior of our approach on tessellation changing meshes, which could be applied to tasks that use several levels-of-detail.

However, with respect to the current results, we consider our method a powerful approach that provides convincing visual results. It has the potential to provide an important basis for future visualization applications.

Acknowledgements. Funding: This work was supported by the DFG: LA 3855/1-1 and HA 7819/1-1, and the Bergen Research Foundation [grant number 811255].

References

[1] Berg, P, Roloff, C, Beuing, O, Voss, S, Sugiyama, SI, Aristokleous, N, et al. The computational fluid dynamics rupture challenge 2013 - phase ii: variability of hemodynamic simulations in two intracranial aneurysms. Journal of biomechanical engineering 2015;137(12):121008.

[2] Janiga, G, Berg, P, Sugiyama, S, Kono, K, Steinman, D. The computational fluid dynamics rupture challenge 2013phase i: prediction of rupture status in intracranial aneurysms. American Journal of Neuroradiology 2015;36(3):530–536.

[3] Kehrer, J, Hauser, H. Visualization and visual analysis of multifaceted scientific data: A survey. IEEE Transactions on Visualization and Computer Graphics 2013;19(3):495–513.

[4] Griffin, W, Wang, Y, Berrios, D, Olano, M. GPU curvature estimation on deformable meshes. In: Symposium on Interactive 3D Graphics and Games. ACM; 2011, p. 159–166.

[5] Ray, N, Li, WC, Lévy, B, Sheffer, A, Alliez, P. Periodic global parameterization. ACM Transactions on Graphics 2006;25(4):1460–1485.

[6] Knöppel, F, Crane, K, Pinkall, U, Schröder, P. Stripe patterns on surfaces. ACM Transactions on Graphics 2015;34(4):39:1–39:11.

[7] Jakob, W, Tarini, M, Panozzo, D, Sorkine-Hornung, O. Instant field-aligned meshes. ACM Transactions on Graphics (Proceedings of SIGGRAPH ASIA) 2015;34(6).

[8] Floater, MS, Hormann, K. Surface parameterization: a tutorial and survey. In: Dodgson, NA, Floater, MS, Sabin, MA, editors. Advances in Multiresolution for Geometric Modelling. Springer. ISBN 978-3-540-26808-6; 2005, p. 157–186.

[9] Bommes, D, Lévy, B, Pietroni, N, Puppo, E, Silva, C, Tarini, M, et al. Quad-mesh generation and processing: A survey. In: Computer Graphics Forum; vol. 32. Wiley Online Library; 2013, p. 51–76.

[10] Ebke, HC, Schmidt, P, Campen, M, Kobbelt, L. Interactively controlled quad remeshing of high resolution 3d models. ACM Transactions on Graphics (TOG) 2016;35(6):218.

[11] Vaxman, A, Campen, M, Diamanti, O, Panozzo, D, Bommes, D, Hildebrandt, K, et al. Directional field synthesis, design, and processing. In: Computer Graphics Forum; vol. 35. Wiley Online Library; 2016, p. 545–572.

[12] Lai, YK, Jin, M, Xie, X, He, Y, Palacios, J, Zhang, E, et al. Metric-driven rosy field design and remeshing. IEEE Transactions on Visualization and Computer Graphics 2010;16(1):95–108.

[13] Palacios, J, Zhang, E. Interactive visualization of rotational symmetry fields on surfaces. IEEE transactions on visualization and computer graphics 2011;17(7):947–955.

[14] Chen, G, Kwatra, V, Wei, LY, Hansen, CD, Zhang, E. Design of 2d time-varying vector fields. IEEE Transactions on Visualization and Computer Graphics 2012;18(10):1717–1730.

[15] Praun, E, Finkelstein, A, Hoppe, H. Lapped textures. Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques SIGGRAPH 00 2000;(1):465–470.

[16] Lefebvre, S, Hoppe, H. Appearance-space texture synthesis. ACM Transactions on Graphics 2006;25(3):541.

[17] Singh, RVP, Namboodiri, AM. Efficient texture mapping by homogeneous patch discovery. Proceedings of the Eighth Indian Conference on Computer Vision, Graphics and Image Processing - ICVGIP '12 2012;:1–8.

[18] Turk, G. Texture synthesis on surfaces. In: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques. SIGGRAPH '01; ACM; 2001, p. 347–354.

[19] Myles, A, Pietroni, N, Zorin, D. Robust field-aligned global parametrization . ACM Transactions on Graphics (TOG) 2014;33:1–14.
[20] Fuchs, R, Hauser, H. Visualization of multi-variate scientific data. Computer Graphics Forum 2009;28(6):1670–1690.
[21] Rocha, A, Alim, U, Silva, JD, Sousa, MC. Decal-maps: Real-time layering of decals on surfaces for multivariate visualization. IEEE Transactions on Visualization and Computer Graphics 2017;(1):821–830.
[22] Schroeder, D, Keefe, DF. Visualization-by-sketching: An artist's interface for creating multivariate time-varying data visualizations. IEEE Transactions on Visualization and Computer Graphics 2016;22(1):877–885.
[23] Kirby, RM, Marmanis, H, Laidlaw, DH. Visualizing multivalued data from 2D incompressible flows using concepts from painting. In: Proceedings of Visualization'99. IEEE; 1999, p. 333–540.
[24] Roy, L, Kumar, P, Golbabaei, S, Zhang, Y, Zhang, E. Interactive Design and Visualization of Branched Covering Spaces. IEEE Transactions on Visualization and Computer Graphics 2017;2626(c).
[25] Marcias, G, Pietroni, N, Panozzo, D, Puppo, E, Sorkine-Hornung, O. Animation-aware quadrangulation. Computer Graphics Forum (proceedings of EUROGRAPHICS/ACM SIGGRAPH Symposium on Geometry Processing) 2013;32(5):167–175.
[26] Jenkins, B. Algorithm alley: Hash functions. `http://www.drdobbs.com/database/algorithm-alley/184410284`; 1997. Accessed: 2017-03-31.
[27] Choong, A, Beidas, R, Zhu, J. Parallelizing simulated annealing-based placement using GPGPU. Proceedings - 2010 International Conference on Field Programmable Logic and Applications, FPL 2010 2010;:31–34.
[28] Naumov, M, Castonguay, P, Cohen, J. Parallel graph coloring with applications to the incomplete-lu factorization on the gpu. Tech. Rep.; NVIDIA, Tech. Rep; 2015.
[29] Corsini, M, Cignoni, P, Scopigno, R. Efficient and flexible sampling with blue noise properties of triangular meshes. IEEE Transactions on Visualization and Computer Graphics 2012;18(6):914–924.
[30] Kindlmann, G, Westin, CF. Diffusion tensor visualization with glyph packing. IEEE Transactions on Visualization and Computer Graphics 2006;12(5).
[31] Ritter, F, Hansen, C, Dicken, V, Konrad, O, Preim, B, Peitgen, HO. Real-time illustration of vascular structures. IEEE Transactions on Visualization and Computer Graphics 2006;12(5):877–884.
[32] Lawonn, K, Luz, M, Preim, B, Hansen, C. Illustrative visualization of vascular models for static 2D representations. In: International Conference on Medical Image Computing and Computer-Assisted Intervention. Springer; 2015, p. 399–406.
[33] De Moura Pinto, F, Freitas, CM. Importance-aware composition for illustrative volume rendering. Proceedings - 23rd SIBGRAPI Conference on Graphics, Patterns and Images 2010;:134–141.
[34] Lichtenberg, N, Smit, N, Hansen, C, Lawonn, K. Sline: Seamless line illustration for interactive biomedical visualization. In: Proceedings of the Eurographics Workshop on Visual Computing for Biology and Medicine (VCBM). The Eurographics Association; 2016,.
[35] Judd, T, Durand, F, Adelson, E. Apparent ridges for line drawing. ACM Transactions on Graphics 2007;26(3):19.
[36] Lawonn, K, Moench, T, Preim, B. Streamlines for illustrative real-time rendering. Computer Graphics Forum 2013;32(3):321–330.
[37] Zander, J, Isenberg, T, Schlechtweg, S, Strothotte, T. High quality hatching. In: Computer Graphics Forum; vol. 23. Wiley Online Library; 2004, p. 421–430.
[38] Meuschke, M, Vo, S, Beuing, O, Preim, B, Lawonn, K. Glyph-based comparative stress tensor visualization in cerebral aneurysms. Computer Graphics Forum 2017;36(3):99–108.
[39] Auer, C, Stripf, C, Kratz, A, Hotz, I. Glyph- and texture-based visualization of segmented tensor fields. In: GRAPP/IVAPP. 2011, p. 670–677.
[40] Van Der Zwan, M, Lueks, W, Bekker, H, Isenberg, T. Illustrative molecular visualization with continuous abstraction. Computer Graphics Forum 2011;30(3):683–690.
[41] Lawonn, K, Krone, M, Ertl, T, Preim, B. Line integral convolution for real-time illustration of molecular surface shape and salient regions. Computer Graphics Forum 2014;33(3):181–190.
[42] Kozlikova, B, Krone, M, Lindow, N, Falk, M, Baaden, M, Baum, D, et al. Visualization of biomolecular structures: State of the art. In: Eurographics Conference on Visualization (EuroVis)-STARs. The Eurographics Association; 2015, p. 061–081.
[43] McLoughlin, T, Laramee, RS, Peikert, R, Post, FH, Chen, M. Over two decades of integration-based, geometric flow visualization. Computer Graphics Forum 2010;29(6):1807–1829.
[44] Schertler, N, Tarini, M, Jakob, W, Kazhdan, M, Gumhold, S, Panozzo, D. Field-aligned online surface reconstruction. ACM Trans Graph 2017;36(4):77:1–77:13.