SKETCH BASED INTERFACE FOR DESIGNING VOLUMETRIC VECTOR FIELDS
ボリュームデータにおけるベクトル場デザインのためのスケッチインタフェース

by

Kenshi Takayama
高山　健志

A Senior Thesis
卒業論文

Submitted to
the Department of Information Science
the Faculty of Science, the University of Tokyo
on February 6, 2007
in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science

Thesis Supervisor: Takeo Igarashi　五十嵐　健夫
Associate Professor of Information Science

ABSTRACT

This paper describes a sketch-based interface for designing vector fields filling 3D models. Vector fields are usually obtained by actual measurements using special equipments or as a result of simulations. However, some applications require manual design of arbitrary vector fields, such as the design of animating water and smoke in computer graphics and running simulation based on some hypotheses in natural science and medicine.

Our system allows the user to specify sample surface vectors by simply drawing strokes on the surface. The user can also cut the model and draw strokes on the cross-section to specify sample vectors inside the model. The system then interpolates all these sample vectors to obtain the final volumetric vector field. We use Laplacian Smoothing algorithm to obtain the optimized solution quickly. The user is also able to browse the resulting vector field efficiently by cutting the model.

Our system supports designers to create desired vector fields quickly and easily. We asked a doctor working on cardiological simulation to try our system for designing muscle fiber orientation inside of a heart as an application, and found that our interface is very useful for practical purposes.

論文要旨

本論文では，3 次元モデル内を充填するベクトル場を自由に作成するための手書きインタフェースを提案する．通常，ベクトル場は高度な計測機器を用いた実測やシミュレーションなどによって得られるものを利用することが多い．しかし，CG 表現において水や煙をデザイナの意図にそって動かす場合や，自然科学・医療分野において特定の仮説に基づきシミュレーションを繰り返し因果関係の傾向を把握する場合などでは，任意のベクトル場を自由に設定することが必要になる．

本システムにおいて，ユーザは 3 次元モデルの表面にストロークを描くことで表面のベクトル場の制約を指定でき，さらにモデルを切断しその断面にストロークを描くことで内部のベクトル場の制約を指定できる．入力が与えられると，システムはまず表面の制約を使って表面のベクトル場を補間し，次いでそのデータとユーザが入力した内部の制約を使って内部のベクトル場を補間する．補間アルゴリズムとしては，最適解が高速に得られる Laplacian Smoothing 法を用いた．また，ユーザはモデルの断面を生成することによって作成したベクトル場を効率的にブラウズすることができる．

本システムを利用することにより，専門家が思い通りのモデルを手早く作成することが可能となる．１つの応用例として心臓シミュレーションの専門家にプロトタイプシステムを試用して心筋の繊維方向を設定する作業を行ってもらい，提案手法が有用であることを確認した．

Acknowledgements

We thank Associate Proffessor Takeo Igarash for his great advice and help. We also thank Dr. Kazuo Nakazawa and Dr. Ryo Haraguchi for their detailed comments and advice based on their experience and deep understanding. We appreciate Dr. Takashi Ashihara's coorporation on our user study and his valuable comments. This work could not be achieved without advice and help from members of Igarashi laboratory, especially Mr. Takashi Ijiri.

Contents

1 Introdunction **1**

1.1 Background . . . 1

1.2 Key Ideas . . . 1

2 Related Work **3**

3 User Interface **5**

4 Algorithm **7**

4.1 Calculation of Boundary Voxels . . . 7

4.2 Laplacian Smoothing . . . 8

4.2.1 Laplacian Matrix on the Surface . . . 9

4.2.2 Laplacian Matrix on the Volume . . . 10

4.3 Setting Constraints . . . 10

4.3.1 On the Surface . . . 10

4.3.2 On the Volume . . . 10

4.4 Adjustment after Smoothing . . . 11

4.5 Calculation of Cross-Section . . . 11

4.5.1 Precomputation . . . 12

4.5.2 Calculation of Voxels on Cross-Section . . . 13

4.5.3 Calculation of Cross-Section Borderlines . . . 14

4.5.4 Calculation of Hidden Surface . . . 15

5 Application Examples **17**

6 User Experience **19**

7 Conclusion **22**

8 Limitations and Future Work **23**
8.1 Tangential Restriction on Boundary 23
8.2 Control on Vector Magnitude 23
8.3 Stroke Modification 23
8.4 Peeling Interface 24
8.5 Mapping to Deformed Model 24

References **25**

List of Figures

3.1 Overview of our system. 6

4.1 Ray casted in the direction of x. Boundary voxels are drawn with red and inside voxels with green. Axis is shown in the left bottom (each red, green and blue line corresponds to x, y and z axis respectively). . 7

4.2 Obtaining cross-section. Borderlines are shown in light blue. 12

4.3 The back-buffer after the precomputation. The model's silhouette is drawn in white. 13

4.4 Cross-section stroke on the back-buffer drawn in red. 14

4.5 Voxels on the cross-section rendered in green points. 14

4.6 Boundary voxels on the cross-section (shown in green) and corresponding polygons (shown in red). 15

4.7 The back-buffer after calculation of hidden surface. One side of the stroke is filled with green. 16

5.1 Velocity fields (left) designed using our system and correspoding particle animations (right) of storm (top), spiral (middle) and stomach (bottom). Particels are rendered as yellow points. 18

6.1 Heart fiber model designed by the doctor using our system. 20

6.2 Sample result of heart simulation based on the model in Figure 6.1. . 20

Chapter 1

Introdunction

1.1 Background

Vector fields are usually obtained by actual measurements using special equipments or as a result of simulations. However, some applications require manual design of arbitrary vector fields, such as the design of animating water and smoke in computer graphics and running simulation based on some hypotheses in natural science and medicine. Few interactive systems are currently available for such purposes, and many designers are forced to use undesirable methods such as assigning vector value to each volume by hand which requires a large amount of time and labor, or calculating vector value automatically by using some approximating functions which is hard to control.

In this paper, we present a novel method for designing volumetric vector fields filling 3D models. Using this method, users can design volumetric vector fields by drawing strokes with common 2D input device such as mouse or pen. The next section describes our key ideas and how they were constructed.

1.2 Key Ideas

Probably the simplest method one might think for designing volumetric vector fields is to specify vector values on some voxels and then interpolate them over the whole 3D space using common interpolation techniques such as radial basis functions. However, such method is unfavorable for our purpose of designing volumetric vector fields filling 3D models, because we want to interpolate with consideration of shape of 3D models. To achieve such shape-considering interpolation, we set an assumption that boundary voxels should always have tangential vectors to surfaces of 3D models. Thanks to this

assumption, we are able to achieve our purpose easily by deviding the problem into two steps: interpolation on surface and interpolation on volume. This approach is also beneficial for users because it allows them to easily specify vector values without creating fair cross-section surface.

One might have a question of whether such an assumption is appropriate or not. However, we see that there are many kinds of volumetric vector fields in the real world which have something like '*contour*' shapes (e.g., storms). Although our system is unsuitable for designing volumetric vector fields which do not have such definite '*contour*' shapes (e.g., explosions), it is very capable of effeiciently designing volumetric vector fields that run along some 3D shapes.

We describe some related works on design of vector fields in Chapter 2. Chapter 3 presents the overview of our prototype system and its user interface. We describe our algorithms in Chapter 4. Chapter 5 shows some results and application examples from our system. Chapter 6 describes a small user test we conducted with a doctor in the area of cardiology. We finally conclude our method in Chapter 7 and discuss limitations of our system and possible directions of future work in Chapter 8.

Chapter 2

Related Work

Various studies have been done on *analysis* and *visualization* of vector fields of various kinds [1]. On the other hand, studies on *design* of vector fields are relatively few. Here we introduce some of the existing methods for designing vector fields.

Salisbury et al. [5] made an efficient tool for designing vector fields on 2D plane for the purpose of rendering 2D image with orientable textures. Its interface is more like that of ordinary 'Paint' applications, and the user can desgin 2D vector fields using operations such as 'draw', 'blur' and 'fill'. It includes 'interpolated fill': fill an area between two curves drawn by the user with interpolation of them. This idea of drawing strokes and interpolating them is very similar to ours.

Praun et al. [4] and Turk [6] used vector fields on surfaces of 3D models to synthesize textures on surfaces. In those papers, they briefly showed simple methods for designing vector fields on surfaces. They let the user specify vector values on some of the vertices of the 3D model and assigned interpolated vector values to the remaining vertices. Praun used Gaussian radial basis functions (distance was defined by Dijkstra's shortest path algorithm on the mesh) for interpolation, while Turk used Mesh Hierarchy method. Their basic idea of designing vector fields on surfaces is very similar to ours. However, we allow the user to specify vector values more intuitively by using sketch-based interface. In addition, we used Laplacian smoothing method for interpolation to achieve interactive rate of performance.

Topology of vector field (i.e., locations of singularities and their connectivities) is often very important for certain applications such as texture synthesis and non-photorealistic rendering, because it has direct influence on visual appearances. Zhang et al. [7] showed a novel method for designing vector fields on 2D planes and 3D surfaces with consideration of topology. Currently our system is yet to support controls of

topology, though.

To our knowledge, there are no studies on design of volumetric vector fields. We propose a method for designing volumetric vector fields using vector fields on surfaces of 3D models. In the next chapter, we present the overview of our system and its user interface.

Chapter 3

User Interface

Our prototype system is shown in Figure 3.1.

The system first lets the user specify a 3D polygon model to be loaded in ".obj" format. The only requirement for this 3D model is that it should have closed surface so that 3D space can be fairly devided into two parts: inside and outside of the model. The system then performes several precomputations including:

- calculation of volume boundary
- calculation of Laplacian matrix on the surface
- calculation of Laplacian matrix on the volume

After these precomputations have finished, the user can now design volumetric vector fields using sketch-based interface.

First, the user can draw strokes on the surface of the model by just dragging over the surface. Each stroke is rendered as a green line with a gradation of brightness representing its direction. When a part of the surface where the user wants to draw strokes is hidden by another part of the surface, the user can take away such interfering part of the surface by cutting operation. The cutting operation works when the user draws a stroke that begins outside of the model's silhouette, runs over it and ends outside of it. Part of the surface on the right side of this stroke disappears and the contour line of the cross-section surface appears in light blue. This operation also allows the user to draw strokes on the cross-section surface in the same way as drawing strokes on the surface. Part of the surface which is cut off by the above operation appears again when the user clicks outside of the model's silhouette.

These are the main operations available for the user's design. To support design process, we also provide save/load functions of the input strokes as well as undo/redo

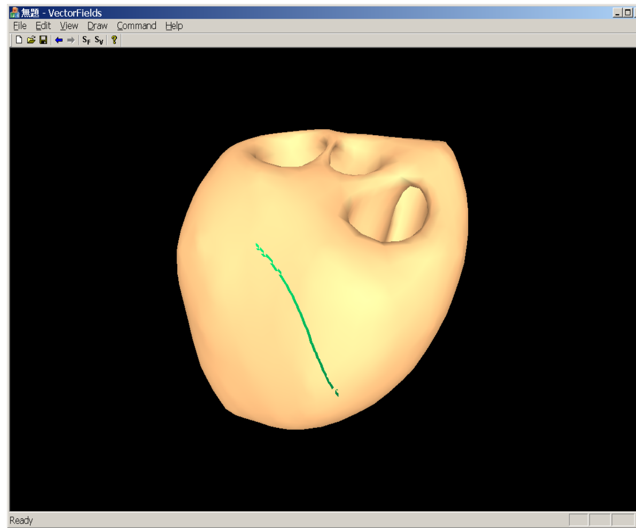

Figure 3.1: Overview of our system.

commands.

When the user has drawn satisfactory amount of strokes, the user then lets the system perform Laplacian smoothing on the surface and on the volume sequentially to obtain the final result of volumetric vector field.

This result and corresponding volume data can be exported to ".txt" files, which will be used from other application softwares.

We implemented our system using Microsoft Visual C++ with MFC platform. We also used OpenGL to render the scene.

Chapter 4

Algorithm

4.1 Calculation of Boundary Voxels

We calculate boundary voxels by ray-casting method. Assume that the model is inside of the box $[0,1] \times [0,1] \times [0,1]$. Rays parallel to the direction of x are casted from every grid-point on plane $x = 0$ heading to $x = 1$. Each ray then computes the position where it first intersects with the model's surface and marks the closest grid-point to this position as being boundary voxel. All the grid-points on the ray are considered as inside voxels until the ray reaches to the next boundary voxel. Each ray iterates this process until it reaches to $x = 1$. We show this process in Figure 4.1.

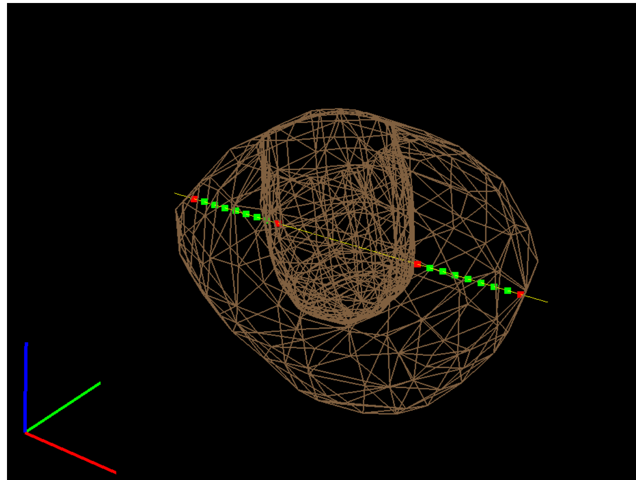

Figure 4.1: Ray casted in the direction of x. Boundary voxels are drawn with red and inside voxels with green. Axis is shown in the left bottom (each red, green and blue line corresponds to x, y and z axis respectively).

After processes about the direction of x are done, we then cast rays in the direction of y and z in the same way. Note that each boundary voxel has information about position on the model, which is required when setting constraints on the volume and calculating borderlines of cross-section. This information includes index i of polygon P_i and two parameters s and t which represent relative position on that polygon as

$$< position >= (1 - s - t)\mathbf{v}_0 + s\mathbf{v}_1 + t\mathbf{v}_2 \tag{4.1}$$

where $\mathbf{v}_0$, $\mathbf{v}_1$ and $\mathbf{v}_2$ are the positions of the first, second and third vertex of P_i respectively. We call this set of information as '*PosInfoOnPolygon*' hereafter in this paper.

4.2 Laplacian Smoothing

Here we give a brief description on Laplacian Smoothing method.
Let $X = \{x_1, \cdots, x_n\}$ as the set of variables and assume that each variable x_i has a linear equation as

$$x_i = \sum_{j \in N_i} w^i_j x_j \quad (i = 1, \cdots, n) \tag{4.2}$$

where N_i is subset of $\{1, \cdots, n\} - \{i\}$ and w^i_j is the corresponding weight. We then set constraints on some variables as

$$x_{k_i} = b_i \quad (i = 1, \cdots, m) \tag{4.3}$$

where k_i is the index of variable in the i-th constraint and m is the number of constraints. Now the aim of Laplacian Smoothing is to find the values of remaining unconstrained variables that best fit the equation (4.2) under the constraints of (4.3).

Above equations (4.2) and (4.3) can be rewritten using vectors and matrices as

$$\mathbf{Lx} = \mathbf{0} \tag{4.4}$$

$$\mathbf{Cx} = \mathbf{b} \tag{4.5}$$

where $\mathbf{x} = (x_1, \cdots, x_n)^{\mathbf{T}}$ and $\mathbf{b} = (b_1, \cdots, b_m)^{\mathbf{T}}$ are vectors, and $\mathbf{L} = (l_{ij})$ and $\mathbf{C} = (c_{ij})$ are $n \times n$ and $m \times n$ matrices respectively defined by

$$l_{ij} = \begin{cases} -1 & (i = j) \\ w^i_j & (j \in N_i) \\ 0 & otherwise \end{cases}$$

$$c_{ij} = \begin{cases} 1 & (j = k_i) \\ 0 & otherwise \end{cases}$$

These two equations (4.4) and (4.5) can be unified into one single equation as

$$\begin{pmatrix} \mathbf{L} \\ \mathbf{C} \end{pmatrix} \mathbf{x} = \begin{pmatrix} \mathbf{0} \\ \mathbf{b} \end{pmatrix} \tag{4.6}$$

This $(n+m) \times n$ matrix $\mathbf{A} = \begin{pmatrix} \mathbf{L} \\ \mathbf{C} \end{pmatrix}$ is not square and this means it is impossible to find a solution which satisfies all the equations. Instead we obtain an optimized one by solving another equation

$$\mathbf{A}^{\mathbf{T}}\mathbf{A}\mathbf{x} = \mathbf{A}^{\mathbf{T}} \begin{pmatrix} \mathbf{0} \\ \mathbf{b} \end{pmatrix} \tag{4.7}$$

Matrix $\mathbf{A}^{\mathbf{T}}\mathbf{A}$ and vector $\mathbf{A}^{\mathbf{T}} \begin{pmatrix} \mathbf{0} \\ \mathbf{b} \end{pmatrix}$ can be extracted as

$$\begin{aligned} \mathbf{A}^{\mathbf{T}}\mathbf{A} &= (\mathbf{L}^{\mathbf{T}}\ \mathbf{C}^{\mathbf{T}}) \begin{pmatrix} \mathbf{L} \\ \mathbf{C} \end{pmatrix} \\ &= \mathbf{L}^{\mathbf{T}}\mathbf{L} + \mathbf{C}^{\mathbf{T}}\mathbf{C} \\ \mathbf{A}^{\mathbf{T}} \begin{pmatrix} \mathbf{0} \\ \mathbf{b} \end{pmatrix} &= (\mathbf{L}^{\mathbf{T}}\ \mathbf{C}^{\mathbf{T}}) \begin{pmatrix} \mathbf{0} \\ \mathbf{b} \end{pmatrix} \\ &= \mathbf{C}^{\mathbf{T}}\mathbf{b} \end{aligned}$$

Note that the k_i-th diagonal element in $\mathbf{C}^{\mathbf{T}}\mathbf{C}$ is 1 and the k_i-th element in $\mathbf{C}^{\mathbf{T}}\mathbf{b}$ is b_i (for $i = 1, \cdots, m$) and all the other elements in $\mathbf{C}^{\mathbf{T}}\mathbf{C}$ and $\mathbf{C}^{\mathbf{T}}\mathbf{b}$ are 0.
In our case, Laplacian matrix $\mathbf{L}$ remains unchanged and constraint matrix $\mathbf{C}$ varies during interaction. It is thus smart to precompute the Laplacian matrix and add the updated constraint matrix to it. Our Laplacian matrix is sparse enough to apply a specific algorithm for sparse matrix. We used UMFPACK, a library for solving sparse linear systems, in our implementation. We describe in the following subsection how we define Laplacian matrix on the two fields: surface and volume.

4.2.1 Laplacian Matrix on the Surface

Vector field on the surface is defined on vertices of the model. We let N_i the set of indices of vertices around the i-th vertex and define weights as

$$w_j^i = \frac{1}{|N_i|}$$

This means that the value of i-th vertex is simply the average of the vertices around.

4.2.2 Laplacian Matrix on the Volume

Vector field on the volume is defined on the voxels. However, since we don't want to deal with all the voxels outside of the model, we first index the inside voxels and create a mapping between this index and the actual position. Each value of voxel should be the average of values of voxels touching with its six sides: top, bottom, right, left, front and back. We define N_i and weights similarly to the ones described in the previous subsection.

4.3 Setting Constraints

Here we describe how we set constraints for Laplacian smoothing.

4.3.1 On the Surface

We first initialize all the vector values of the vertices to $\mathbf{0}$ before we convert strokes on the surface into constraints on the surface.

Since each stroke on the surface is represented as sequential segments of *PosInfoOnPolygon*, we explain how a single segment of *PosInfoOnPolygon* p_0 and p_1 is converted to constraints. We first convert p_0 and p_1 to actual position $\mathbf{v}_0$ and $\mathbf{v}_1$ by (4.1) , and obtain the differential vector $\mathbf{w} = \mathbf{v}_1 - \mathbf{v}_0$. We then simply add this vector $\mathbf{w}$ to all the three vertices that consist the polygon of p_0. After we added all the differential vectors of each segment in the strokes, we finally normalize all the constrained non-zero vectors on the surface. The method we used here is very rough and possibly there may be other smarter methods, though.

4.3.2 On the Volume

After we run Laplacian smoothing on the surface to obtain vector field on the surface, we now set constraints on the volume. As is stated in Section 1.2, we use vector fields on the surface as well as strokes on the volume as constraints on the volume.

Fisrt we explain how the vector field on the surface is converted to constraints on the volume. Since each boundary voxel has *PosInfoOnPolygon* p (see Section 4.1), we can easily obtain its constrained value by linear interpolation as

$$< value >= (1 - s - t)\mathbf{w}_0 + s\mathbf{w}_1 + t\mathbf{w}_2$$

where $\mathbf{w}_0$, $\mathbf{w}_1$ and $\mathbf{w}_2$ are the vector values of the first, second and third vertex in the polygon of p respectively.

We now move on to convert strokes on the volume to constraints on the volume. Since each stroke on the volume is represented as sequential segments of 3D coordinates, we explain how a single segment of 3D coordinates $\mathbf{p}_0$ and $\mathbf{p}_1$ is converted to constraints. We take the simplest method that the differential vector $\mathbf{w} = \mathbf{p}_1 - \mathbf{p}_0$ is assigned to the voxel which is closest to $\mathbf{p}_0$.

4.4 Adjustment after Smoothing

Because we want vectors on the surface to be tangential to the surface, we remove their non-tangential components by using normals of the surface after smoothing on the surface. In addition, we currently do not consider the magnitude of vectors so we normalize all the vectors both on the surface and on the volume after each smoothing.

4.5 Calculation of Cross-Section

Our purpose of cross-sectioning is only the following two: to draw strokes on the cross-section and to draw strokes on some part of the surface which is hidden by another part of the surface. We thus do not calculate remeshed polygons to show the exact shape of the cross-section. Instead we only show borderlines of cross-section surface and hide part of polygons which should be cut off. Figure 4.2 shows how the cross-sectioning works. We implemented these two functions using pixel-based algorithm where the system processes the back-buffer which is not visible on the screen. Details are described in the following subsections.

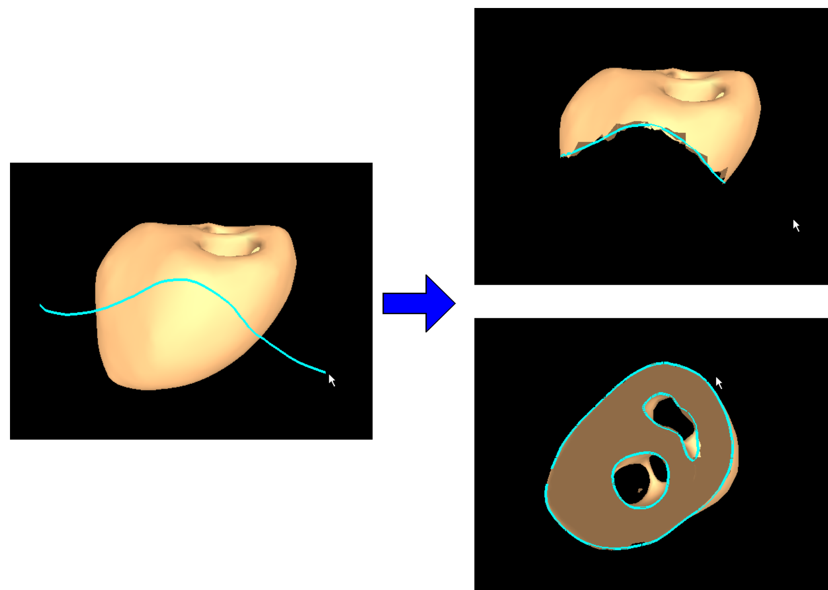

Figure 4.2: Obtaining cross-section. Borderlines are shown in light blue.

4.5.1 Precomputation

Assume that the back-buffer is cleared with black. We perform precomputation every time when the view changes (i.e., rotation and scale). This process includes drawing silhouette of the model with white and computing coordinates of all the vertices projected onto the screen. These are required when calculating polygons to be hidden. Figure 4.3 shows how the back-buffer appears after this process.

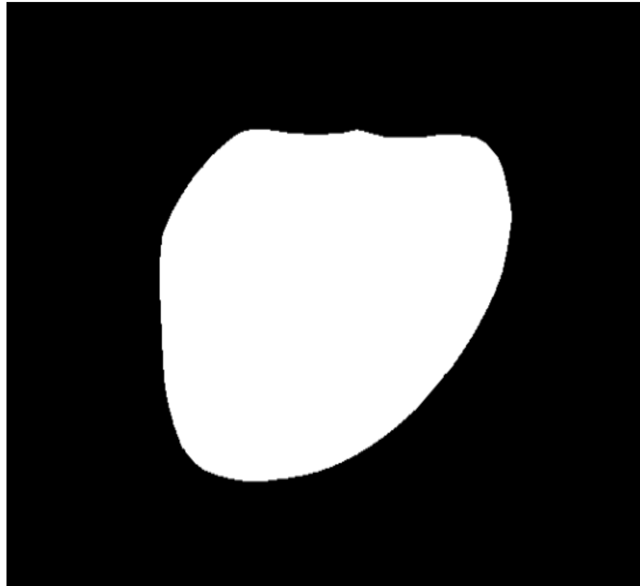

Figure 4.3: The back-buffer after the precomputation. The model's silhouette is drawn in white.

4.5.2 Calculation of Voxels on Cross-Section

After the cross-section stroke is drawn, the system first calculates whether each voxel is on the cross-section or not. This information is required when visualizing vector field on the volume and drawing strokes on the volume as well as calculating borderlines of the cross-section surface. The cross-section stroke is drawn on the back-buffer in red with a certain width, and then each voxel is projected on the back-buffer to examine if it is on the stroke. Figure 4.4 shows how the back-buffer appears after this process and Figure 4.5 shows voxels on the cross-section.

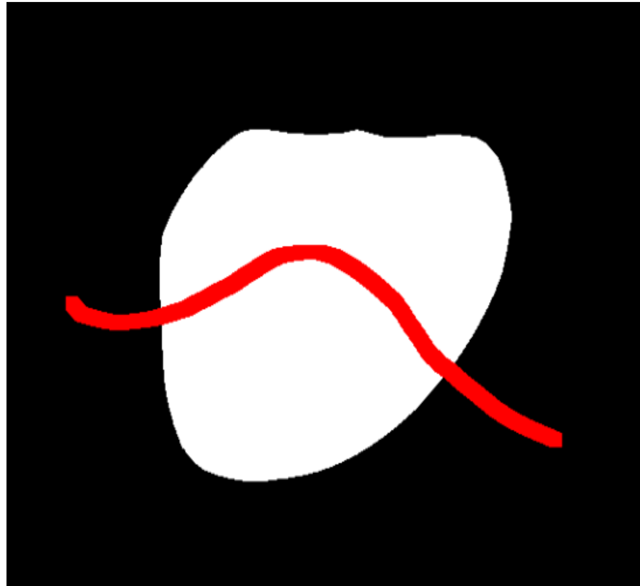

Figure 4.4: Cross-section stroke on the back-buffer drawn in red.

Figure 4.5: Voxels on the cross-section rendered in green points.

4.5.3 Calculation of Cross-Section Borderlines

We need to calculate all the intersections between polygons of the model and segments in the cross-section stroke. However, most of the polygons do not intersect with the stroke. We thus reduce the wasted calculation by eliminating polygons which are far from the stroke using following techniques. We first gather bounary voxels from voxels on the cross-section obtained in the previous subsection, and make a list of polygons that *PosInfoOnPolygon* of these boundary voxels belong to. We show such voxels and polygons in Figure 4.6.

Figure 4.6: Boundary voxels on the cross-section (shown in green) and corresponding polygons (shown in red).

Now we calculate the borderlines of cross-section. First, we assign a flag of 'un-visited' to all the polygons of the model. We then perform following process for all polygons in the list:

1. If the flag is 'visited', do nothing.
2. Set flag of 'visited' to this polygon.
3. Calculate its intersection with all the segments of the cross-section stroke.
4. If any intersections are detected, call this function recursively for the three neighboring polygons.

4.5.4 Calculation of Hidden Surface

We calculate hidden part of the surface by examining on which side of the cross-section stroke is each vertex projected. If a vertex is projected onto the right side of the stroke, all the polygons that include this vertex are considered as hidden part of the surfcae. We thus need to distinguish the two areas of silhouette partitioned by the stroke. This is achieved by filling one side of the stroke with green by using ordinary seed-fill algorithm. Figure 4.7 shows the back-buffer after this process. We also need to decide which color (white or green) is on the right side of the stroke. We solved this problem by using approximation of relation between a certain point in one side and its closest segment in the stroke.

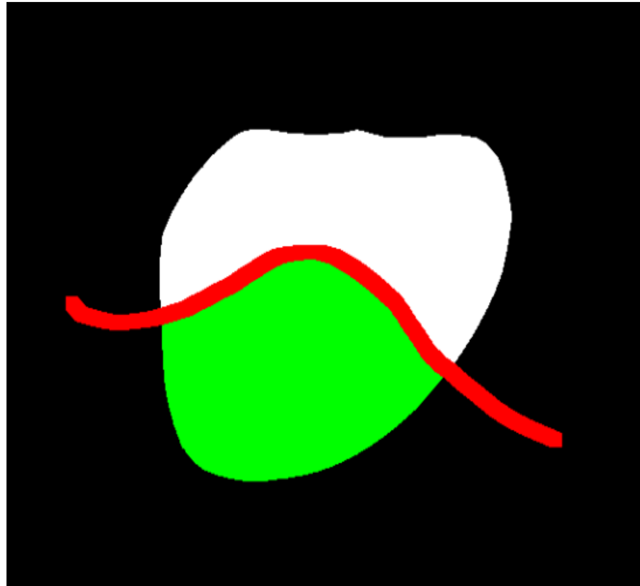

Figure 4.7: The back-buffer after calculation of hidden surface. One side of the stroke is filled with green.

Chapter 5

Application Examples

We designed some sample models of volumetric vector fields using our system and applied them for particle animations as shown in Figure 5.1. We could design these models in a few minutes and they provides attractive appearances, which suggests the effectiveness of our system for particle animations.

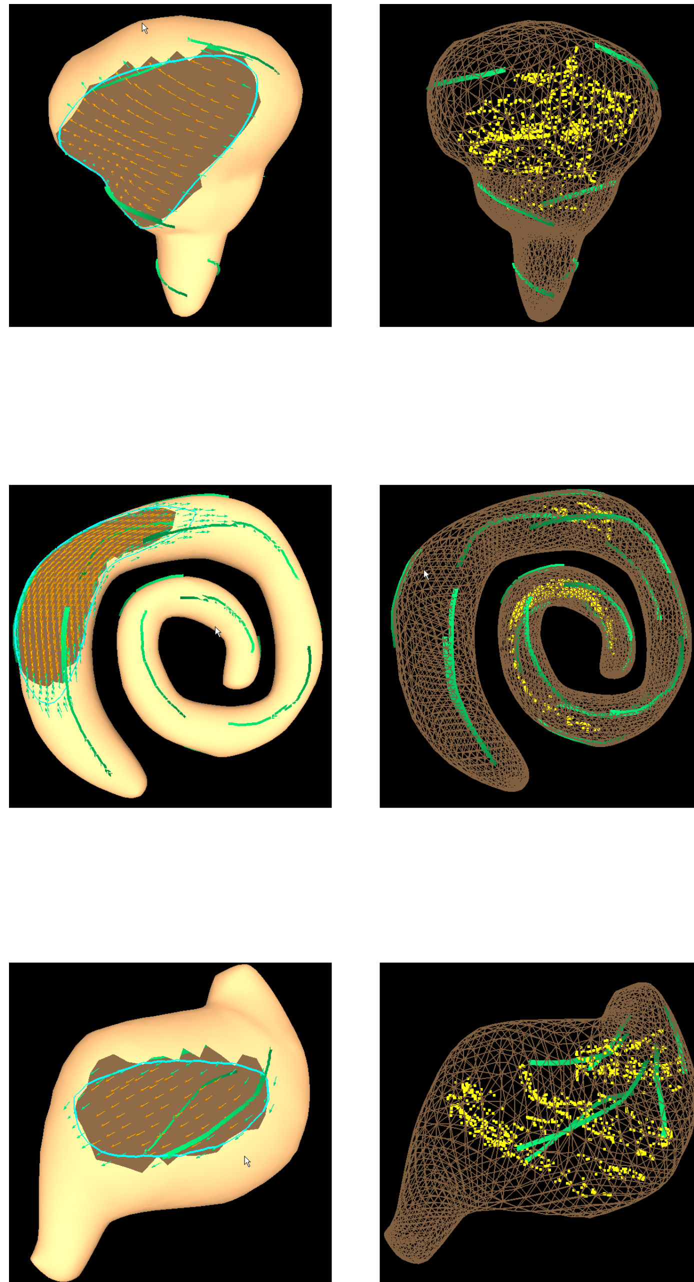

Figure 5.1: Velocity fields (left) designed using our system and correspoding particle animations (right) of storm (top), spiral (middle) and stomach (bottom). Particels are rendered as yellow points.

Chapter 6

User Experience

We conducted a small user test with a doctor in the area of cardiology.

In this area, there is an aproach to abnormal cardiac rhythm by using electrical simulation of heart. Variations of heart fiber orientation represented as volumetric vector field is required in such simulation to explore direct influences of fiber orientation on stimulus propagation. It has been very hard, however, to obtain such fiber orientation because there has been no effective tools for dealing with volumetric vector fields. Therefore, our system has potential power to accelerate research in this area.

We asked the doctor to design a sample model of heart fiber orientation by using our system. The test was conducted with an ordinary laptop PC (equipped with 2.1 GHz processor and 2.0 GB memory) and a mouse. We provided a commercially available 3D polygon model of heart for him to design fiber orientation on it. We first explained the usage of our system, and he was able to get used to it in about 10 minutes. He then started to design a sample model of heart fiber orientation and finished it in about 8 minutes. The model he designed is shown in Figure 6.1. Figure 6.2 shows the sample result of simulation based on this heart fiber model.

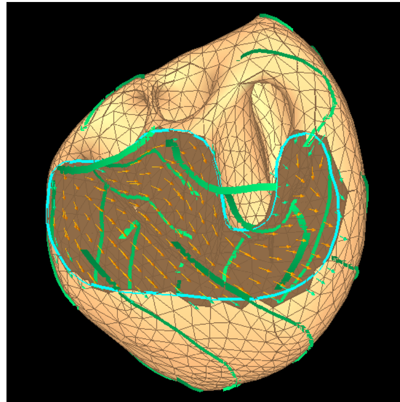

Figure 6.1: Heart fiber model designed by the doctor using our system.

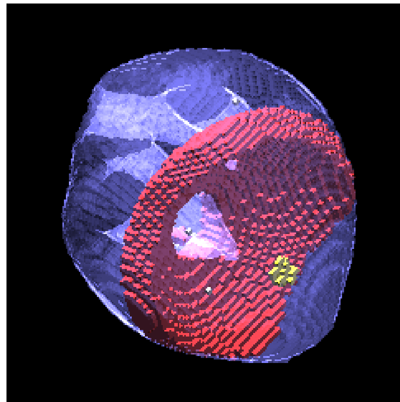

Figure 6.2: Sample result of heart simulation based on the model in Figure 6.1.

We then interviewed him and obtained following feedbacks.

First, he evaluated our system as outstanding for his research area, because it allows users direct design of 3D fiber structures without lowering dimensions to 2D. He was pleased with his results from our system, as it finely showed typical twisted structure of heart fiber. He stressed that our system is definitely faster than existing methods, even if it may take considerable time for calculation as the number of grid increases.

On the other hand, he noted some unpleasant points in our system and showed some possible directions of further development if we aim to specialize our system for design of heart fiber models.

He often got troubles of losing what part of the heart he was looking at while changing views, because there are little landmarks to guide on the surface of heart.

Therefore, he desired a function that enables him to come back to some preset view-points.

As we had expected, he wanted to modify strokes he had drawn because he often had troubles that some endpoints of his strokes unexpectedly bounced back. This taught us a lesson that stroking with mouse is not an easy operation for an unpracticed user, and remided us the importance of stroke modification. He also said that he could have designed well if pen and tablet were available.

He then noted that it will be great if he could design heart fiber model with sample images of actual medical data mapped onto the surface of heart. This will allow him far more real design by tracing sample images. In general, he pointed out, tracing operation is important from the medical point of view because it means human filtering of noisy medical data.

In heart fiber modeling, he noted, smooth interpolation is not always welcome because there often exists areas where fiber orientation changes discontinuously. He suggested that allowing users to specify such areas will be useful.

He taught us that there is a place called 'cardiac apex' on the surface of heart which looks like a vortex. Since its location is very important in medical sense, he suggested that it will be great if our system allows the user to control topology of vector fields on the surface of heart.

Finally he pointed out that our method of visualizing volumetric vector fields by cross-sectioning is not well suitable for heart fiber model, because structure of actual heart fiber consists of number of folding layers of heart muscle and many researchers associate heart fiber models with layers, not with cross-section surfaces. Thus he suggested a 'peeling' interface, with which a designer can see varing rate by continuously peeling layers. He also noted that such visualization technique will enhance further intuitive design as well.

Chapter 7

Conclusion

In this paper we presented a novel method for designing volumetric vector fields filling 3D models using sketch-based interface, and described detailed algorithms for implementation. We took an approach of two-step Laplacian smoothing which achieved an easy and interactive design. We also conducted a small user test with a doctor in the area of cardiology and received some valuable feedbacks.

As is pointed out in Chapter 6, our system has some clear improvable points. So we will discuss the limitations of our system and show possible directions of future work in the next chapter.

Chapter 8

Limitations and Future Work

8.1 Tangential Restriction on Boundary

We used an approach of using vector field on surface as constraints for volumetric vector field in order to allow user to design easily and intuitively. But this restriction that volumetric vector field should always have tangential value at the boundary may cause difficulty in some cases, such as designing volumetric vector field for animations of particles spreading out of certain region. This problem may be solved by allowing user to specify areas on surface where should not be considered as constraints for volumetric vector field.

8.2 Control on Vector Magnitude

Vector magnitude is very important in some applications such as simulations and animations, which cannot be handled in our current system. There are many possible interfaces for this purpose including use of color or width of strokes, and we may need to explore the best choice through corporative prototyping approach.

8.3 Stroke Modification

When a user wants to create a variety of slightly different volumetric vector fields of one single model, function of stroke modification will greatly reduce the amount of the user's labor. The user will be able to design another volumetric vector field in short time by modifying strokes he or she has once drawn. This function may possibly be achieved by various methods for curve editing such as Igarashi's method [2].

8.4 Peeling Interface

As is stated in section **??**, 'peeling' interface will be very suitable to deal with models which have layered structure (i.e., vectors are always tangential to the surface of each layer). And it will be useful as a tool not only for designing but also for visualizing volumetric vector field, as the user will be able to intuitively understand continuous change rates of volumetric vector field depending on depth. To achieve this interaction is not a trivial problem, but several studies such as [3] might be a good clue for the solution.

8.5 Mapping to Deformed Model

Mapping volumetric vector field of certain model to another slightly deformed model would be useful in some cases such as ...??? In our method, strokes on surface are associated with polygons of the model and they can be finely mapped onto the deformed model with no effort. However, strokes inside of the model are not associated with any positional information of the model, so the problem of mapping them onto another is not a trivial question.

References

[1] Helwig Hauser, Robert S. Laramee, and Helmut Doleisch. State-of-the-art report 2002 in flow visualization.

[2] Takeo Igarashi, Tomer Moscovich, and John F. Hughes. As-rigid-as-possible shape manipulation. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1134–1141, New York, NY, USA, 2005. ACM Press.

[3] Shigeru Owada, Frank Nielsen, Makoto Okabe, and Takeo Igarashi. Volumetric illustration: designing 3d models with internal textures. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 322–328, New York, NY, USA, 2004. ACM Press.

[4] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 465–470, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[5] Michael P. Salisbury, Michael T. Wong, John F. Hughes, and David H. Salesin. Orientable textures for image-based pen-and-ink illustration. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 401–406, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[6] Greg Turk. Texture synthesis on surfaces. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 347–354, New York, NY, USA, 2001. ACM Press.

[7] Eugene Zhang, Konstantin Mischaikow, and Greg Turk. Vector field design on surfaces. *ACM Trans. Graph.*, 25(4):1294–1326, 2006.