LAPPED SOLID TEXTURES: FILLING A MODEL WITH ANISOTROPIC TEXTURES

Lapped Solid Textures: 異方性ソリッドテクスチャを用いた中身を持つ３次元物体のモデリング

by

Kenshi Takayama
高山　健志

A Master Thesis
修士論文

Submitted to
the Graduate School of the University of Tokyo
on February 3, 2009
in Partial Fulfillment of the Requirements
for the Degree of Master of Information Science and Technology
in Computer Science

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

ABSTRACT

We present a method for representing solid objects with spatially-varying oriented textures by repeatedly pasting solid texture exemplars. The underlying concept is to extend the 2D texture patch-pasting approach of lapped textures to 3D solids using a tetrahedral mesh and 3D texture patches. The system places texture patches according to the user-defined volumetric tensor field over the mesh to represent oriented textures. We extend the original technique to handle nonhomogeneous textures for creating solid models whose textural patterns change gradually along a depth field. We identify several texture types considering the amount of anisotropy and spatial variation and discuss methods for obtaining solid texture exemplars based on these texture types, including our novel techniques for synthesizing nonhomogeneous solid textures and for manually designing solid textures with a painting interface. We provide a tailored user interface for constructing tensor fields over the mesh based on these texture types. With our simple framework, large-scale realistic solid textured models can be created easily with little memory and computational cost. We demonstrate the effectiveness of our approach with several examples including trees, fruits, and vegetables.

論文要旨

我々は，サイズの小さなソリッドテクスチャをサンプルとして 3 次元形状内に繰り返し貼り付けることで，サイズの大きな中身の詰まった 3 次元モデルを作成することができる手法を提案する．基本的なアイディアは，既存のサーフェス上でのパッチベースのアルゴリズムを，2 次元テクスチャを 3 次元ソリッドテクスチャに，三角形メッシュを四面体メッシュにそれぞれ置き換えることで 3 次元版に拡張するというものである．システムは 3 次元テクスチャのパッチを，ユーザが指定した 3 次元テンソル場に沿って繰り返し貼り付けていくことで，中身の詰まった 3 次元モデルを表現する．また提案手法では，空間的遷移のあるテクスチャをモデル内に定義されたデプス場に合わせて並べることが可能である．我々はソリッドテクスチャのサンプルを，その異方性の度合いおよび空間的遷移の度合いに基づいていくつかのタイプに分類し，これに基づいたテクスチャサンプルの合成方法についても議論する．その中で我々は，空間的に遷移するテクスチャの合成手法およびペイントインタフェースによるテクスチャの直接的なデザイン手法を新しく提案する．我々はモデル上にテンソル場をデザインするためのインタフェースをテクスチャのタイプごとに適切に設計することで，直感的なテクスチャ配置の指定を可能にした．提案手法を用いて作成した樹木・果物・野菜などモデルの例を通して，少ない計算コストと記憶容量でサイズの大きな中身の詰まった 3 次元モデルを作成するのに提案手法が有効であることを示す．

Acknowledgements

My deepest gratitude goes to my supervisor Takeo Igarashi for always encouraging me to pursue research, and providing numerous essential comments and suggestions to improve my unpolished demos, writings, and talks.

I would like to thank Kazuo Nakazawa and Ryo Haraguchi at National Cardiovascular Center Research Institute, and Takashi Ashihara at Shiga University of Medical Science, for providing me a chance to do a research in the field of medicine, and supporting me enthusiastically to write a paper for a medical journal. The initial motivation of this thesis came from the discussions with them.

Makoto Okabe, one of the coauthors of my SIGGRAPH paper, first gave me the hint of extending lapped textures to 3D solids. My project would not have even started without his insightful words. He also helped me with his excellent implementation of solid texture synthesis despite his own SIGGRAPH submission and Ph.D. thesis. Takashi Ijiri, another coauthor, helped me with his nice 3D modeling of trees and other fruits. His continuous positive comments on my project made me believe that it deserves publication at SIGGRAPH. Shigeru Owada at Sony CSL gave me his helpful feedback on my project in its early stage. I thank anonymous SIGGRAPH reviewers' essential comments and suggestions which improved my paper a lot. This project was funded in part by Information-technology Promotion Agency (IPA), Japan.

I would like to thank all the members at the Igarashi laboratory for their discussions and funny jokes.

And finally, I am grateful for my family's constant support and love.

Contents

1 Introduction **1**

1.1 Backgrounds 1

1.1.1 Surface-based 3D graphics 1

1.1.2 Volume graphics 2

1.2 Our contributions 3

1.2.1 Publications 4

2 Related Work **5**

3 Classification of Solid Textures **7**

3.1 Anisotropy level 8

3.2 Variation level 8

3.3 Seven types of texture 8

4 Methods for Creating Solid Texture Exemplars **10**

4.1 Review of solid texture synthesis from 2D exemplars 10

4.2 An extension to synthesizing nonhomogeneous solid textures 16

4.3 A painting interface for the manual design of solid textures 20

5 User Interface **23**

5.1 Texture type 0 23

5.2 Texture type 1-a 23

5.3 Texture type 1-b 24

5.4 Texture type 2-a 25

5.5 Texture type 2-b 26

5.6 Manual pasting of textures 27

6 Algorithm **28**

6.1 Rendering an LST model 28

6.1.1 Cutting . . . 29
6.1.2 Volume rendering . . . 29
6.2 Construction of an LST model . . . 29
6.2.1 Creating an alpha mask of the solid texture . . . 30
6.2.2 Constructing a tensor field . . . 31
6.2.3 Selecting a seed tetrahedron . . . 32
6.2.4 Growing a clump of tetrahedra . . . 33
6.2.5 Texture optimization . . . 34
6.2.6 Coverage test of tetrahedron . . . 35
6.2.7 Creation of depth-varying solid models . . . 35

7 Results **37**
7.1 Limitations . . . 38

8 Conclusions **42**
8.1 Future Work . . . 42

References **44**

List of Figures

1.1 Photographs of a kiwi fruit. The appearance of the cross-section differs depending on the orientation of the cutting plane. 3

3.1 Our classification of solid textures. 7

4.1 Input and output of the algorithm. 11

4.2 The nearest neighborhood search in E_x. Note that each voxel $\mathbf{v}$ stores the Euclidean distance d between neighborhoods of S and E_x as well as the pixel location $\mathbf{p}$ in E_x. 12

4.3 The blend of colors from the exemplar neighborhoods to optimize the synthesis. Each voxel $\mathbf{u}$ that lies in the neighborhood of voxel $\mathbf{v}$ contributes to the new blended color of $\mathbf{v}$ with the color sampled from the exemplar neighborhood of $\mathbf{u}$ at the location corresponding to $\mathbf{v}$. 13

4.4 Pseudo-code of solid texture synthesis in multi-resolution. 14

4.5 Pseudo-code of texture optimization phase. $N_i(\mathbf{v})$ denotes a set of voxels that constitute $\mathbf{v}$'s neighborhood on the plane orthogonal to i-axis. 15

4.6 Pseudo-code of histogram matching. 16

4.7 Real-world examples that can be represented using layered solid textures: (a) stratum and (b) cake. 16

4.8 The estimation of a layered solid texture only from a 2D texture exemplar of the cross-section parallel to the depth direction. 17

4.9 Additional control map channel encoding depth information fed to the synthesis algorithm. (a) A depth control map augmented to the 2D exemplar. (b) The depth-aware initialization of the synthesis volume at the coarsest level. 18

4.10 A problem of severe sweeping artifact appears with our naive extension of only using additional depth map channel. 18

4.11 The cause of the sweeping artifact. Neighborhoods of the synthesis volume in the two directions tends to match the same neighborhood in the 2D exemplar. 19
4.12 A modified version of the neighborhood matching process to deal with the sweeping artifact. (middle) We collect the two best matching neighborhoods for each of the two directions (numbers displayed near the neighborhood squares mean distances of the matching). (right) A simple selection is performed to avoid the matched neighborhoods in the two directions pointing to the identical location in the 2D exemplar. . 19
4.13 The effect of our simple scheme removing the severe sweeping artifact seen in Figure 4.10. Notice how the appearances on the cross-sections orthogonal to the depth direction are well synthesized thoroughly from a 2D exemplar parallel to the depth direction. 20
4.14 Photographs of cross-sections of a tree (left) and a carrot (right). Notice that the variance in the direction of the central axis is very small. . . . 21
4.15 The process of creating a texture of carrots. The system sweeps the user-specified 2D image in 3D space (left), applies some noise (middle), and adds a thin layer of exterior skin to create the carrot texture (right). 21
4.16 The process of creating a texture of kiwi fruits. After the system sweeps the user-specified 2D cross-sectional image without seeds (left), the user loads a 3D model representing the seed geometry (middle left), and pastes it one-by-one onto the 3D texture (middle right), to finally obtain the kiwi fruit texture (right). 22

5.1 Modeling process for texture type 0. (a) Moving the texture patch by dragging with the mouse. (b) Changing the texture scale with the mouse wheel. (c) User-specified texture scaling. (d) Result of automatic filling (rendered with displacement mapping). 24
5.2 Modeling process for texture type 1-a. (a) Drawing strokes to specify local vector fields. (b) Setting the texture scaling. (c) Result of automatic filling. 24
5.3 Modeling process for texture type 1-b. (a) Multi-face-fill tool. (b) Single-face-fill tool modifying the colored region. (c) Stroke tool. (d) Computed depth field. (e) Result of automatic filling. 25

5.4 Modeling process for texture type 2-a. (a) Specifying the depth field. (b) Specifying the secondary directions by drawing strokes on layers. (c) Setting the texture scaling. (d) Result of automatic filling. 26
5.5 Modeling process for texture type 2-b. (a) Specifying the depth field. (b) Specifying the secondary directions by drawing strokes on layers. (c) Result of automatic filling. 26
5.6 Manual pasting of additional textures. (a) Solid texture exemplar to be pasted. (b) Moving and rotating the texture patch by dragging with the mouse. (c) Changing the texture scale with the mouse wheel. . . . 27

6.1 Cutting operation. (a) User-drawn stroke across the 3D model. (b) Scalar field computed from the stroke. (c) Resulting cross-sectional surface mesh. 29
6.2 Manual creation of a 3D alpha mask. (a) 3D model of the shape of the mask. (b) Cross-sections of the alpha mask. 30
6.3 The optimization minimizes the difference vectors $\mathbf{d_r^i}, \mathbf{d_s^i}, \mathbf{d_t^i}$ between the texture coordinate axes $(\hat{\mathbf{r}}, \hat{\mathbf{s}}, \hat{\mathbf{t}})$ and the transformed tensor axes $(\mathbf{R_i'}, \mathbf{S_i'}, \mathbf{T_i'})$. 34
6.4 Predefined sampling points used for coverage test of overlapping texture patches. 35
6.5 (a) A problem occurs if we use only a single alpha mask. (b) The use of three types of alpha mask solves this problem. 36
6.6 Three types of alpha mask: (a) outer part, (b) middle part, and (c) inner part. 36

7.1 Models filled with overlapping solid textures: (a) kiwi fruit, (b) carrot, and (c) tree (the grayscale texture represents the displacement map channel). Note that the input solid textures include surface textures as well as interior textures. 38
7.2 Results of our method. (a) Watermelon. (b) Volume rendering of a fibrous tube. (c) Strata. (d) Cake. 40
7.3 Artifacts around tensor field singularities. 41
7.4 Failure case with a highly structured texture. (a) A curved cylinder filled with bricks shows (b) blurring and (c) misalignment artifacts. . . 41

Chapter 1

Introduction

1.1 Backgrounds

1.1.1 Surface-based 3D graphics

The most basic way to model a 3D object on a computer is to define its 3D surface shape using one of those geometric representations, such as triangular or quadrilateral polygon mesh, Non-Uniform Rational B-Spline (NURBS) surface, and subdivision surface. In order to decorate the object surface with further detailed and small-scale features that cannot be handled with geometric elements, texture images are usually applied onto the surface, which are either made from real-world photographs, painted by artists, or generated automatically using some procedural functions.

The most basic form of such texture is RGB color texture, but other types of textures have also been explored to achieve further rich and detailed appearance: bump map [2] is used to show bumpy effect by altering surface normals, displacement map [4] is used to alter the geometry itself by moving each surface vertex along its normal direction, and bidirectional texture function (BTF) [7] is used to model view- and light-dependent behavior of material appearance appropriately by exploiting huge image database captured under various lighting and viewing directions.

In addition to these techniques for planar 2D textures, volumetric 3D textures have also been applied onto the object surface creating "thin shells" around the object, to achieve various effects, such as fur [15], small-scale geometric detail [26], and nonhomogeneous subsurface scattering [3].

1.1.2 Volume graphics

While surface-based techniques described in the previous section allow us to produce quite convincing rendering results, it is still difficult to perform volumetric operations on 3D models in a realistic and interactive manner. Examples of volumetric operations include cutting, tearing, peeling, carving, and smashing of 3D objects, which we do with real-world objects in our daily lives. Possible applications of such interactions on a computer would include virtual cooking practice system, virtual surgery training system, communication support system for doctors and patients, and educational tool for science classes.

Despite its importance, advances in techniques aimed at such volumetric effects, or *volume graphics*, have been relatively slow compared to surface-based graphics techniques. This is mainly because of difficulties in obtaining 3D models with internal material information, which we call *solid textured models*.

The most straightforward method for obtaining such a model would be to capture internal appearance of a 3D object by making many thin slices of it and taking a photograph for each slice [1]. However, this method requires a high-accuracy slicer machine, which is not accessible for many people. Also, it obviously destroys the object, which would be problematic in some cases. Computed Tomography (CT) and Magnetic Resonance Imaging (MRI) technologies have been commonly used in the field of medicine and engineering in order to examine the internal structure of a 3D object in a noninvasive manner, but the expensive measurement equipments required for these methods prohibit the frequent use of them in other fields.

Methods for obtaining solid textured models without capturing real-world volume data can be roughly categorized into three approaches: the procedural approach, runtime 2D texture synthesis on cross-sections, and example-based 3D solid texture synthesis. The first approach (e.g., [32, 6]) allows the design of an arbitrary solid texture by writing an explicit program, but this is difficult for nonexpert users. The second approach (e.g., [29, 33]) provides efficient and intuitive ways to create quasi-solid textured models simply by giving example 2D images on cross-sections. However, these methods have several limitations (e.g., inconsistency among different cross-sections or difficulty in handling textures with discontinuous elements, such as seeds), which sometimes cause unrealistic artifacts. The last approach (e.g., [14, 17]) allows the user to create realistic and consistent solid textured models made of various materials by explicitly synthesizing volumetric textures from 2D examples. However, the amount of data and computational cost become problematic for large-scale 3D models because

the number of voxels grows cubically as the model size increases.

1.2 Our contributions

Our goal is to create large-scale solid textured models efficiently using 3D solid texture exemplars. The basic concept is to extend the 2D texture patch-pasting approach of lapped textures [34] to 3D solids by replacing the 2D texture and triangular mesh with a 3D texture and tetrahedral mesh. This enables the creation of consistent large-scale solid textured models without computing and storing individual voxel colors.

We made various extensions to the original technique to make it work for solids. First, our method can arrange solid textures along a tensor field (i.e., a set of three orthogonal vector fields) instead of a vector field. This is important because many real-world objects actually have internal local tensor fields. For example, the horizontal and vertical cross-sections of a kiwi fruit appear different (Fig. 1.1), which shows that there is a local tensor field inside the fruit that consists of the circumferential and the vertical directions in addition to the depth direction. We can create such anisotropic solid models using appropriate solid texture exemplars and arranging them along user-specified tensor fields.

Figure 1.1: Photographs of a kiwi fruit. The appearance of the cross-section differs depending on the orientation of the cutting plane.

While the original technique was limited to homogeneous textures, we have extended it to handle nonhomogeneous (or ‘layered’) textures. By considering the depth of the layers during the texturing process, we can create depth-varying solid textured models for objects such as kiwi fruits, carrots, and trees, whose appearance changes gradually in the depth direction.

We classify solid textures into several types according to the amount of anisotropy and spatial variation, and discuss how to obtain such texture exemplars based on our

classification. We first briefly review an existing method for synthesizing homogeneous 3D solid textures from 2D images, and then describe how to extend it to nonhomogeneous solid texture synthesis. In contrast to these completely automatic texture synthesis methods, we also propose a painting interface for manually designing 3D textures which would be difficult to handle with existing automatic algorithms.

For each texture type, we provide a tailored user interface for the tensor field design and an algorithm for the texture patch arrangement. A sketching interface is used to specify the vector field, and a painting interface is used to define the depth field inside the model. The system computes a tensor field from the user-specified vector orientation and the gradient of the depth field, and pastes texture exemplars onto the model so that they align with the tensor field.

Using our method, various solid textured objects can be designed easily and created efficiently with little memory and computational cost. We demonstrate the effectiveness of our approach on several examples including trees, fruits, and vegetables.

1.2.1 Publications

Our technique to create solid textured models by repeatedly pasting solid texture patches was previously published and presented as: *Lapped Solid Textures: Filling a Model with Anisotropic Textures* [39] at ACM SIGGRAPH 2008 in Los Angeles, USA, in collaboration with Makoto Okabe, Takashi Ijiri and Takeo Igarashi at The University of Tokyo.

We applied our sketch-based interface for volumetric tensor field design to the modeling of myocardial fiber orientation aimed at electrophysiological simulation of ventricles of heart, which was published as: *A Sketch-Based Interface for Modeling Myocardial Fiber Orientation that Considers the Layered Structure of the Ventricles* [38], in collaboration with Takashi Ashihara at Shiga University of Medical Science, Takashi Ijiri and Takeo Igarashi at The University of Tokyo, Ryo Haraguchi and Kazuo Nakazawa at National Cardiovascular Center Research Institute.

Chapter 2

Related Work

One common approach to creating solid textured models is to use procedural methods. Earlier work by Perlin [32] produced realistic solid textures by developing material-specific mathematical models using noise functions. Cutler et al. [6] created layered solid models by specifying depth and material information in a scripting language. However, these methods are not accessible to nonexpert users because of the difficulties in writing the appropriate code.

Another approach is to synthesize 2D cross-sectional images every time the model is cut. Owada et al. [29] proposed a modeling system in which the user associates 2D reference images with the object's cross-sections via an intuitive user interface. The system then performs 2D texture synthesis on the cross-sections while considering the user-specified guidance information. By combining three types of texture, complex volumetric illustrations such as teeth and cakes can be created in a short time. However, the consistency among different cross-sections was not considered, and the approach leads to unrealistic appearances in some cases.

Pietroni et al. [33] proposed a similar method to produce photorealistic images on cross-sections. In their system, the user first takes several photographs of the cross-sections of a real object and places them in 3D space so that they align with the cross-sections of a virtual 3D model. When the model is cut, the system morphs the input photographs to produce cross-sectional images. However, the morphing approach is applicable to smoothly varying patterns only and cannot handle textures with discontinuous elements, such as seeds.

Owada et al. [28] proposed an interface for directly designing volumetric models by segmenting the internal region of a 3D model into several regions and placing many colored particles inside the segmented regions. The user draws strokes on the cross

sections of the 3D model to specify the segmentations, and puts several colored particles inside each segmented region as a sample. The system analyzes the distribution of the input particles and synthesizes many particles accordingly to fill the entire region. However, their system relies solely on the manual specification of colored particles and does not allow image-based modeling, which makes it difficult to create realistic results.

Example-based solid texture synthesis actually fills 3D volumetric space with patterns seen in example 2D images. Earlier methods were based on a parametric approach using global statistics, such as histograms [12], spectra [11], and their combination [8]. These methods only work well for textures whose appearance can be fully captured by such global statistics, and cannot synthesize textures with macro structures, such as a brick wall. To overcome this issue, the nonparametric approach was later used [42, 35, 17], and recent work by Kopf et al. [17] produced realistic solid textures from 2D exemplars by combining texture optimization [18] and histogram matching [12]. There are some other approaches to solid texture synthesis including procedural shader-based methods [19] and stereology-based methods [14], although they were designed for relatively limited texture types.

Example-based solid texture synthesis has advantages over the other approaches because it can generate consistent and detailed textures from 2D examples. The drawback, however, is the cost in both computation and memory, as it explicitly computes and stores a dense 3D array of voxels covering the entire target model. Although recent work by Dong et al. [9] alleviated this problem by limiting the synthesis domain to only around the surface and performing a spatially deterministic synthesis algorithm on GPU which is extended from the parallel deterministic texture synthesis in 2D [20], the fundamental problem of large memory requirements when dealing with large-scale solid textured models still exists. In addition, a nontrivial extension is necessary to create spatially-varying oriented textures in a geometry-dependent manner. Our aim is to solve these problems by applying the 2D patch-based approach of lapped textures [34] to 3D solid textures. While the lapped textures technique has already been extended to 3D shell textures for real-time furs on surfaces [21], to our knowledge, application of this approach to solid textures has not been explored previously.

Chapter 3

Classification of Solid Textures

We first classify solid textures into several types as we provide a different user interface and construction algorithm for each. As shown in Figure 3.1, our classification is based on two aspects of solid textures: *anisotropy level* and *variation level*.

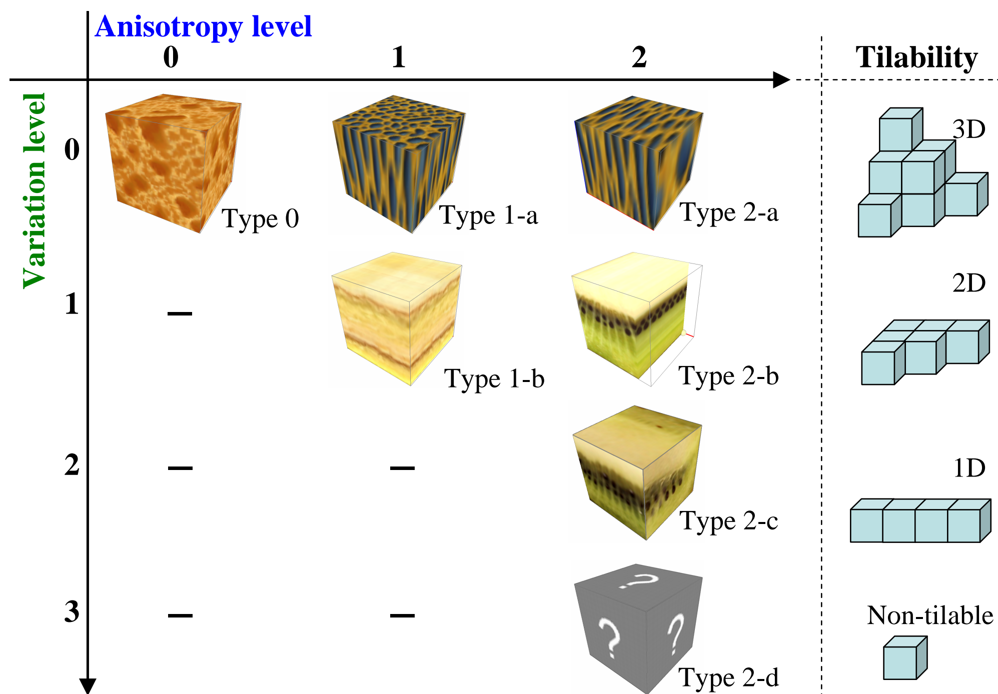

Figure 3.1: Our classification of solid textures.

3.1 Anisotropy level

The anisotropy level describes how the appearance of a cross-section varies depending on the orientation of the cutting plane. Anisotropy level 0 means that the texture is isotropic and the cross-section looks similar regardless of its orientation. Anisotropy level 1 means that the texture has an axis, and its cross-section shows two different appearances depending on whether its orientation is parallel or perpendicular to the axis. This type of texture requires a vector field for alignment when placed in 3D space. Anisotropy level 2 means that the cross-section shows three different appearances depending on the orientation. A tensor field (a set of three orthogonal vector fields) is required to place this type of texture in 3D space.

3.2 Variation level

The variation level corresponds to the number of directions in which the textural pattern changes gradually. Variation level 0 means that there is no gradual variation in the texture and therefore the texture is homogeneous everywhere. Variation level 1 means that the texture has a single direction in which its appearance changes gradually. Variation levels 2 and 3 mean that the texture has two and three axes of variation respectively. The variation level also represents the tilability of the texture. Variation level 0 texture can be tiled three-dimensionally, variation level 1 texture can be tiled two-dimensionally, and variation level 2 texture can only be tiled linearly. Variation level 3 texture cannot be tiled in any dimension.

3.3 Seven types of texture

The variation level is limited by the anisotropy level and therefore there can be only 7 types of texture in our classification. Type 0 is the well-known isotropic textures. Type 1-a corresponds to "anisotropic" textures in [17] or "oriented" textures in [29]. Type 1-b corresponds to "layered" textures in [29]. This paper covers types 2-a and 2-b in addition to these other texture types. Although the basic framework is sufficiently general to cover all 7 types, we do not support 2-c and 2-d in the current prototype implementation because we have not encountered many interesting real-world examples of these types. In addition, our approach is essentially a tiling method and is not very effective for textures with limited tilability.

We do not claim that our classification describes all the real-world objects, although we believe that it is possible to represent most objects using the solid textures classified

as outlined above individually or in combination.

Chapter 4

Methods for Creating Solid Texture Exemplars

In this chapter, we discuss methods for creating solid texture exemplars required for our lapped solid texture framework. We first briefly review a method for synthesizing homogeneous solid textures from 2D exemplars proposed by Kopf et al. [17] which can be used to create texture types 0, 1-a, and 2-a in Chapter 3. This will next be extended to nonhomogeneous (or 'layered') solid texture synthesis which can handle the texture type 1-b. We also present a painting interface for manually designing solid textures that would be difficult to handle with existing automatic synthesis algorithms.

4.1 Review of solid texture synthesis from 2D exemplars

Here we only review the most fundamental part required for implementing solid texture synthesis algorithm in [17], and omit other things such as algorithmic backgrounds and nonessential details, because our aim is only to make it easier for readers to understand our extension of the technique to layered solid texture synthesis, which will be described in the next section. Readers may refer to the original article [17] for more details.

The input to the algorithm consists of three exemplar images E_x, E_y, and E_z corresponding to three cross-sections orthogonal to x-, y- and z-axis, respectively (Fig. 4.1(left)). Note that E_y and E_z can be identical to E_x when synthesizing isotropic solid texture. The output is a solid texture S whose appearances on its three cross-sections orthogonal to x-, y-, and z-axis are always similar to E_x, E_y, and E_z, respectively (Fig. 4.1(right)).

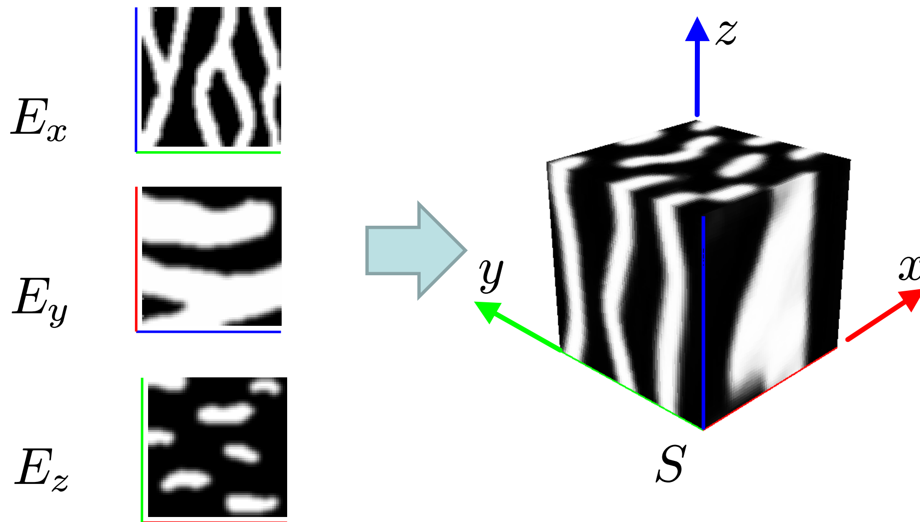

Figure 4.1: Input and output of the algorithm.

The synthesis progresses in a multi-resolution manner, so the first step is to down-sample the exemplar images several times (usually twice is enough) to create exemplars for each level. The solid texture at the coarsest level is first initialized with random color values sampled from the exemplars. At each synthesis level, the currently synthesized result is refined iteratively through two phases: *neighborhood search* and *texture optimization*.

In the neighborhood search phase, we search for each voxel the best matching neighborhood in each of three exemplars, and store its pixel location as well as its Euclidean distance to the neighborhood of the current synthesis volume (Fig. 4.2). This is a standard nearest neighbor search problem in a high dimensional vector space, which can be accelerated with approximation using the Principal Component Analysis (PCA) projection and the Approximate Nearest Neighbor (ANN) library [24].

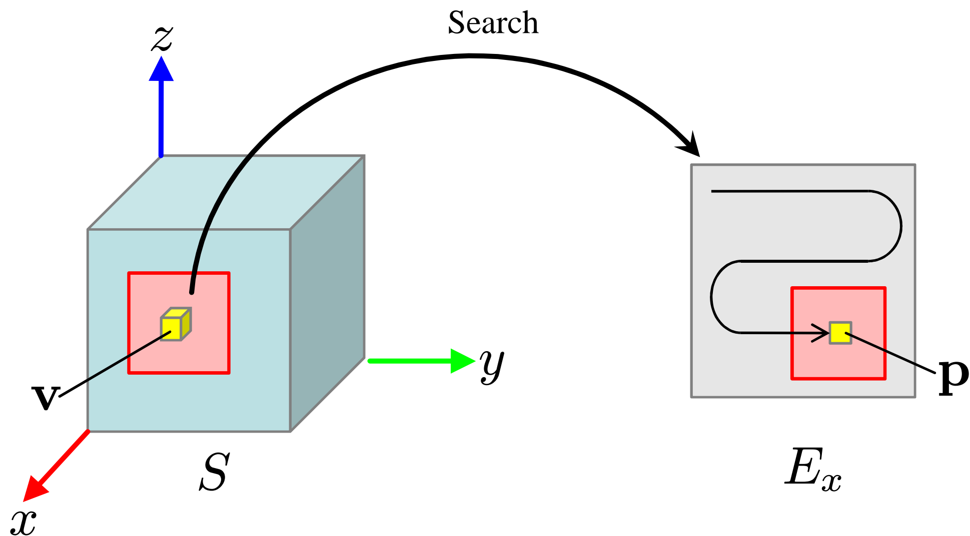

Figure 4.2: The nearest neighborhood search in E_x. Note that each voxel $\mathbf{v}$ stores the Euclidean distance d between neighborhoods of S and E_x as well as the pixel location $\mathbf{p}$ in E_x.

In the texture optimization phase, voxel colors in the current synthesis volume are updated according to the neighborhood matching result. The update of a voxel is simply done by taking the weighted blend of pixels in all the exemplar neighborhoods that overlaps the voxel (Fig. 4.3). The blending weight is set inversely proportional to the distance between the neighborhood in the synthesis volume and the neighborhood in the exemplar. The weight is further altered by the histogram matching scheme [12], which makes it even small if a color to be blended is too much abundant in the currently synthesized volume compared to the exemplars. Note that the histogram of the synthesized volume is updated every time a new blended color is computed for each voxel.

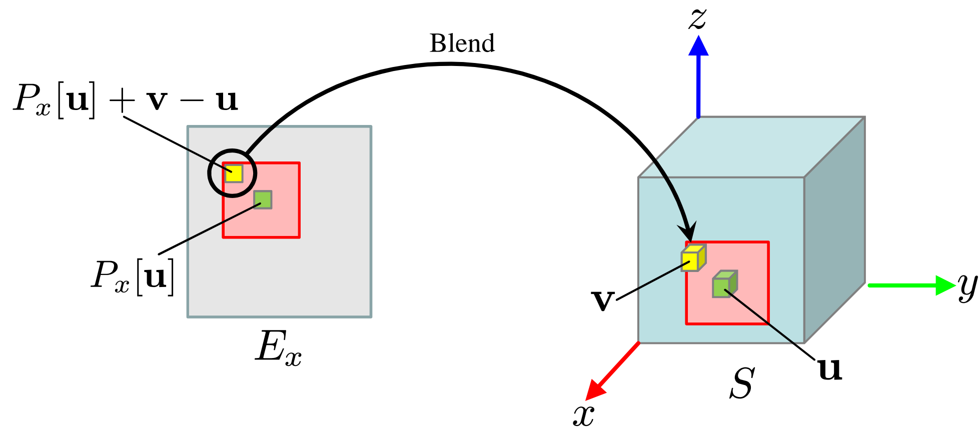

Figure 4.3: The blend of colors from the exemplar neighborhoods to optimize the synthesis. Each voxel $\mathbf{u}$ that lies in the neighborhood of voxel $\mathbf{v}$ contributes to the new blended color of $\mathbf{v}$ with the color sampled from the exemplar neighborhood of $\mathbf{u}$ at the location corresponding to $\mathbf{v}$.

Pseudo-codes for the algorithm overview, the texture optimization procedure, and the histogram matching are shown in Figures 4.4, 4.5, and 4.6, respectively.

```
SolidSynthesisFrom2D(E_x^{L_max}, E_y^{L_max}, E_z^{L_max})
    for l = L_max − 1 to 0
        E_x^l = downsample(E_x^{l+1});
        E_y^l = downsample(E_y^{l+1});
        E_z^l = downsample(E_z^{l+1});
    end
    S^0 = random_init(E_x^0, E_y^0, E_z^0);
    for l = 0 to L_max
        H_E = calc_histogram_E(E_x^l, E_y^l, E_z^l);   // compute histograms
        H_S = calc_histogram_S(S^l);
        repeat
            (P_x, D_x) = search_x(S^l, E_x^l);   // (P, D) = (pixel location, neighborhood distance)
            (P_y, D_y) = search_y(S^l, E_y^l);
            (P_z, D_z) = search_z(S^l, E_z^l);
            S^l := optimize(S^l, E_x^l, E_y^l, E_z^l, H_E, H_S, P_x, D_x, P_y, D_y, P_z, D_z);
        until converged(l);   // some convergence criteria
        if l == L_max return S^{L_max};
        S^{l+1} = upsample(S^l);
    end
end
```

Figure 4.4: Pseudo-code of solid texture synthesis in multi-resolution.

```
optimize(S, E_x, E_y, E_z, H_E, H_S, P_x, D_x, P_y, D_y, P_z, D_z)
    for each voxel v ∈ S (in random order)
        color_acc = 0;
        weight_acc = 0;
        for i ∈ {x, y, z}
            for each u ∈ N_i(v)    // neighborhood of v
                color = E_i[P_i[u] + v − u];
                weight = D_i[u]^(−1.2);
                weight := weight/(1 + histogram_matching(color, H_E, H_S));
                color_acc := color_acc + weight * color;
                weight_acc := weight_acc + weight;
            end
        end
        color_old = S[v];
        color_new = color_acc/weight_acc;
        S[v] := color_new;
        H_S := update_histogram_S(H_S, color_old, color_new);
    end
    return S;
end
```

Figure 4.5: Pseudo-code of texture optimization phase. $N_i(\mathbf{v})$ denotes a set of voxels that constitute $\mathbf{v}$'s neighborhood on the plane orthogonal to i-axis.

```
histogram_matching(color, H_E, H_S)
    result = 0;
    for each channel j
        bin = get_bin(color_j);
        result := result + max(0, H_{S,j}[bin] − H_{E,j}[bin]);
    end
    return result;
end
```

Figure 4.6: Pseudo-code of histogram matching.

4.2 An extension to synthesizing nonhomogeneous solid textures

The goal of our method is to synthesize nonhomogeneous, or 'layered', solid texture exemplars, which belong to the texture type 1-b in our classification (see Chapter 3). Examples of real world objects that can be represented using such layered texture exemplars include strata and cakes, as shown in Figure 4.7.

(a)

(b)

Figure 4.7: Real-world examples that can be represented using layered solid textures: (a) stratum and (b) cake.

This kind of solid textures is difficult to handle with the original algorithm of Kopf et al. [17], because its 2D exemplars corresponding to the cross-sections orthogonal to the depth direction are spatially-variant, and they are not available in many cases.

Therefore, we need to estimate the entire solid texture only from a 2D exemplar corresponding to the cross-section parallel to the depth direction, which is abundant in photographs, as shown in Figure 4.8.

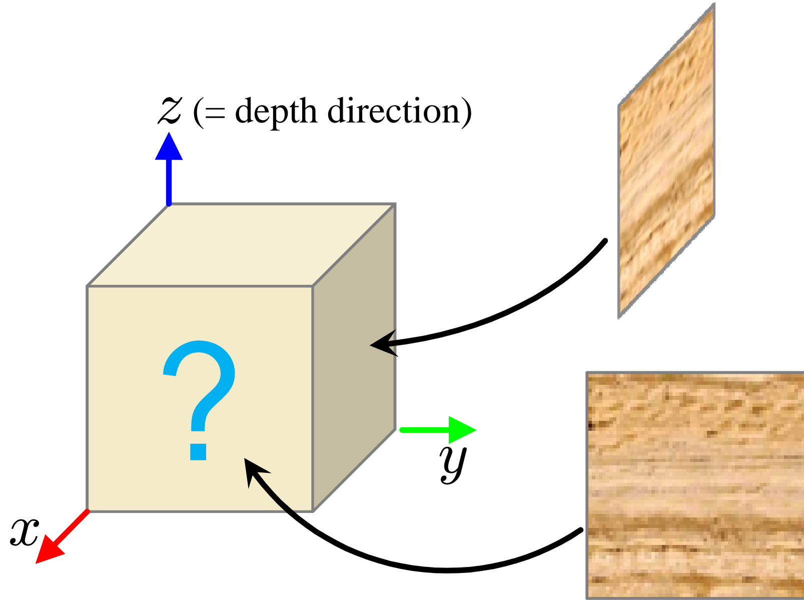

Figure 4.8: The estimation of a layered solid texture only from a 2D texture exemplar of the cross-section parallel to the depth direction.

Our basic idea to solve this problem is to feed the algorithm with another control map channel that encodes the depth information in addition to the three RGB color channels (Fig. 4.9a). As described in the original article [17], their method is already capable of handling multi-channel texture synthesis using additional channels that encode various information such as feature distance, specular, shininess, and displacement, and therefore our extension is quite trivial. We also modified the process of synthesis volume initialization at the coarsest level: instead of initializing each voxel with a pixel sampled from the exemplar in a totally random manner, we initialize each voxel with a pixel randomly sampled from the 2D exemplar which has the same depth value (Fig. 4.9b). Note that, since we do not have any 2D exemplars orthogonal to the depth direction, we perform the neighborhood matching and the texture optimization only in the other two directions.

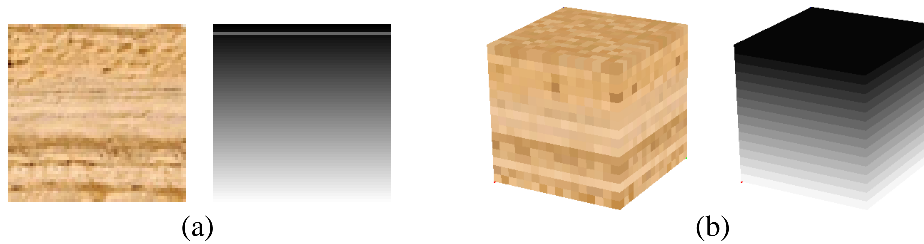

Figure 4.9: Additional control map channel encoding depth information fed to the synthesis algorithm. (a) A depth control map augmented to the 2D exemplar. (b) The depth-aware initialization of the synthesis volume at the coarsest level.

However, this simple extension can easily lead to a severe sweeping artifact as shown in Figure 4.10.

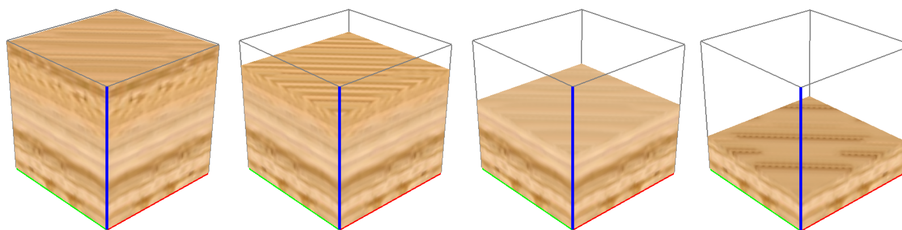

Figure 4.10: A problem of severe sweeping artifact appears with our naive extension of only using additional depth map channel.

The reason of this problem is that the above extension accepts a simple sweep of the 2D exemplar in a direction orthogonal to the depth direction and oblique to both of the other two directions, as an optimized synthesis result. In other words, for each voxel, the two neighborhoods of the current synthesis volume in each of the two directions are very likely to best match with the same neighborhood of the 2D exemplar (Fig. 4.11).

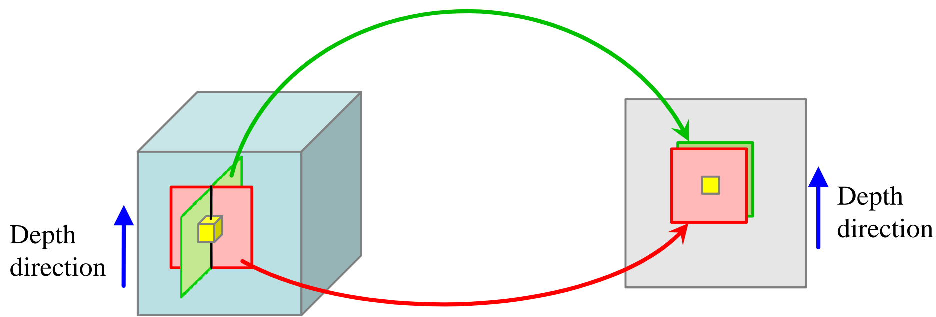

Figure 4.11: The cause of the sweeping artifact. Neighborhoods of the synthesis volume in the two directions tends to match the same neighborhood in the 2D exemplar.

We solve this problem using the following simple scheme. In the neighborhood search phase, for each voxel, we collect the two best matching neighborhoods in each of the two directions (Fig. 4.12(middle)). If the first best matching neighborhoods in borh of the two directions point to the same location in the exemplar, we select one of the two based on the matched distance and assign that pixel as the best matching neighborhood for the selected direction. The neighborhood for the other direction is chosen to the second best matching neighborhood (Fig. 4.12(right)).

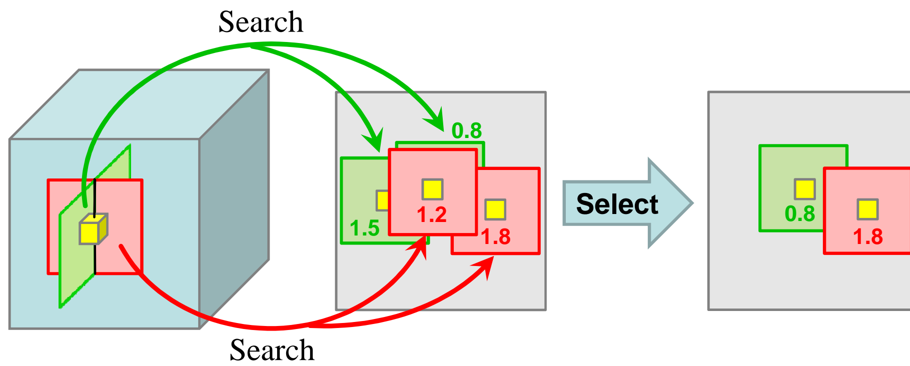

Figure 4.12: A modified version of the neighborhood matching process to deal with the sweeping artifact. (middle) We collect the two best matching neighborhoods for each of the two directions (numbers displayed near the neighborhood squares mean distances of the matching). (right) A simple selection is performed to avoid the matched neighborhoods in the two directions pointing to the identical location in the 2D exemplar.

Figure 4.13 shows how this simple scheme works well to remove the severe sweeping

artifact seen in Figure 4.10.

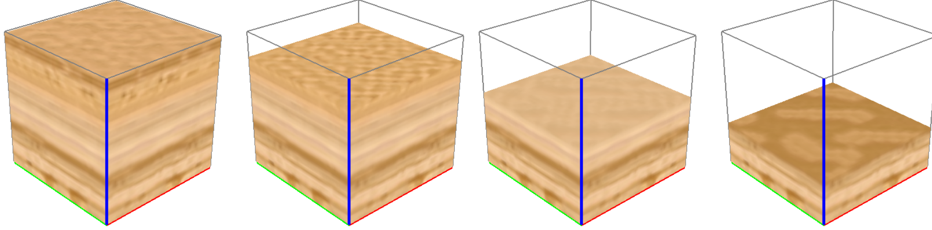

Figure 4.13: The effect of our simple scheme removing the severe sweeping artifact seen in Figure 4.10. Notice how the appearances on the cross-sections orthogonal to the depth direction are well synthesized thoroughly from a 2D exemplar parallel to the depth direction.

4.3 A painting interface for the manual design of solid textures

We propose a simple painting interface for manually designing solid textures that would be still difficult to handle for automatic synthesis algorithms. Specifically, we aim to design textures representing kiwi fruits, carrots, and trees, which are required to create results we will show in Chapter 7. We observe that objects such as trees and carrots show very small variance in the direction parallel to the central axis, as shown in Figure 4.14. Therefore, our system allows the user to create such textures by specifying an image of a cross-seciton perpendicular to the central axis (Fig. 4.15) which is then sweeped along the direction of the axis. The user can further add some randomness by globally applying noise functions [32, 5]. The user can also add a thin layer of the exterior skin by specifying an appropriate image.

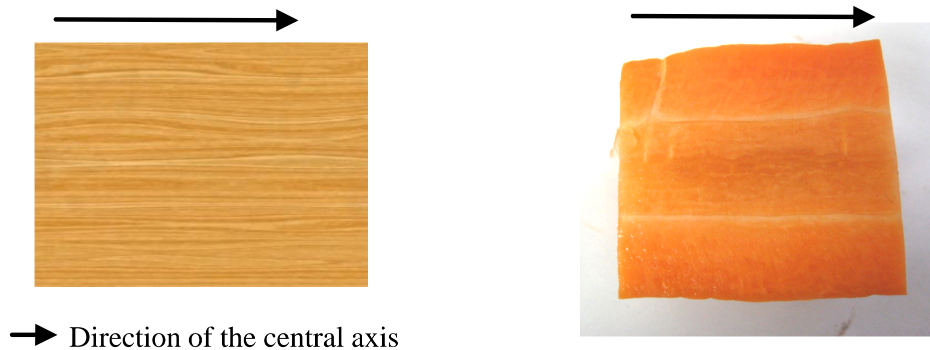

Figure 4.14: Photographs of cross-sections of a tree (left) and a carrot (right). Notice that the variance in the direction of the central axis is very small.

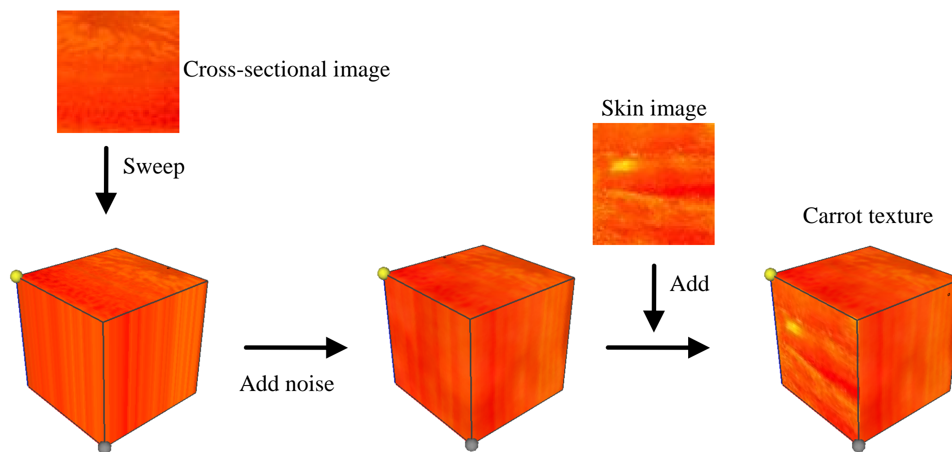

Figure 4.15: The process of creating a texture of carrots. The system sweeps the user-specified 2D image in 3D space (left), applies some noise (middle), and adds a thin layer of exterior skin to create the carrot texture (right).

In the case of the kiwi fruit texture, we observe that the cross-section of kiwi fruits also has small variance in the direction parallel to the central axis similarly to carrots and trees, except for its many discrete seeds as shown in Figure 1.1. Therefore, we allow the user to create such textures by first sweeping a cross-sectional image that does not contain any seeds, and next adding discrete seeds individually (Fig. 4.16). After the user removes seeds from the original cross-sectional image using existing photo editing tools such as PhotoShop, the system sweeps it in the direction parallel to the central axis. The user can then load a 3D model representing the 3D geometry

of the seed created using existing 3D modeling tools such as FiberMesh [25] into the system. The user adjusts the position, orientation and scaling of the seeds with mouse dragging so that they are placed in appropriate locations, and fills its inside with the currently selected color by double-clicking. Repeating this process many times, the kiwi fruit texture can be finally obtained. The user can further modify undesired artifacts by locally blurring the individual voxels.

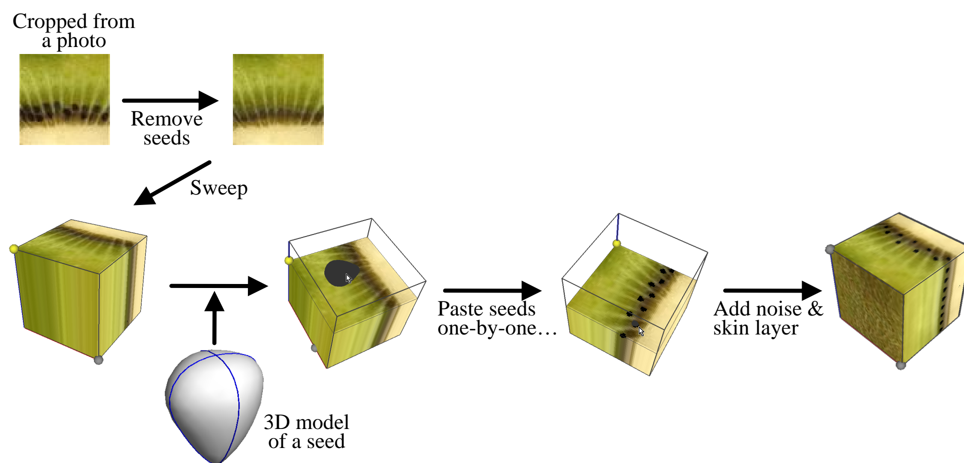

Figure 4.16: The process of creating a texture of kiwi fruits. After the system sweeps the user-specified 2D cross-sectional image without seeds (left), the user loads a 3D model representing the seed geometry (middle left), and pastes it one-by-one onto the 3D texture (middle right), to finally obtain the kiwi fruit texture (right).

Chapter 5

User Interface

The user first loads a geometry model (triangular mesh) and exemplar solid texture data (cubic array of RGB colors). The user can rotate, translate, and scale the camera view by dragging with the right mouse button. The user can also cut the model and see its cross-sectional surface by drawing a freeform stroke across the model [13]. After the input geometry and texture are specified, the system shows a dialog box to allow the selection of a texture type. We explain the modeling process for each texture type in the following sections. Note that the details of the algorithm are described in Chapter 6.

5.1 Texture type 0

This type corresponds to isotropic textures, such as a sponge or concrete. The user specifies the texture scaling in this case. The user first puts a solid texture onto the model by clicking, and moves it interactively by dragging with the mouse (Fig. 5.1a). The user can also change the texture scale interactively using the mouse wheel (Fig. 5.1b). When satisfied, the user can set the local texture scale by double-clicking on the desired position of the model. After the texture scaling is set appropriately (Fig. 5.1c), the system fills the model with the texture taking into account such user-specified texture scaling (Fig. 5.1d).

5.2 Texture type 1-a

This type represents textures with flow or fiber orientation, such as bamboo and muscle. The user first specifies a volumetric vector field over the model. The user can draw strokes on the surface or cross-sections of the model to specify the local vector

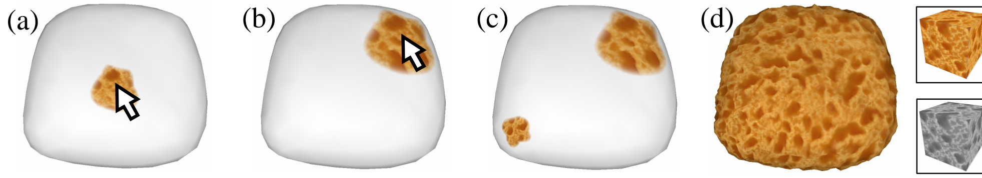

Figure 5.1: Modeling process for texture type 0. (a) Moving the texture patch by dragging with the mouse. (b) Changing the texture scale with the mouse wheel. (c) User-specified texture scaling. (d) Result of automatic filling (rendered with displacement mapping).

field (Fig. 5.2a). A similar interface was described previously [29]. After several strokes are drawn, the user then sets the texture scaling (Fig. 5.2b) as described in Section 5.1. Finally, the system fills the model with the texture (Fig. 5.2c).

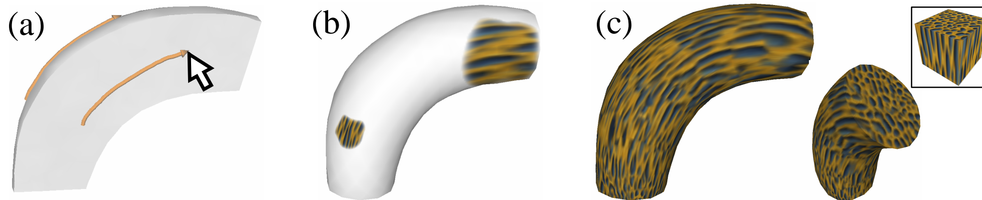

Figure 5.2: Modeling process for texture type 1-a. (a) Drawing strokes to specify local vector fields. (b) Setting the texture scaling. (c) Result of automatic filling.

5.3 Texture type 1-b

This type represents models with depth-varying texture, such as cakes and strata. The user specifies a depth field over the model using a painting user interface similar to that reported by Owada et al. [29]. The user first chooses the color that represents the depth (red and blue correspond to the outermost and the innermost parts, respectively). The user can then paint the model using three tools: a multi-face-fill tool, a single-face-fill tool, and a stroke tool. The multi-face-fill tool assigns a color to multiple surface triangles that are adjacent to each other and have the same color. If adjacent triangles have a curvature larger than a certain threshold, the system treats them as if they were not adjacent, which allows the user to fill, for example, only the side faces of a cylinder (Fig. 5.3a). The single-face-fill tool assigns a color to a single surface triangle clicked by the user (Fig. 5.3b), which allows modifying and controlling the result of

multi-face-fill tool. Finally, the stroke tool allows the user to draw colored strokes on the surface and cross-sections of the model (Fig. 5.3c). This tool is useful to mark the central axis of radial textures. When the user presses the “Update” button, the system interpolates the depth value over the model (Fig. 5.3d). After the depth field is set appropriately, the system then fills the model with the texture while considering the depth (Fig. 5.3e). Texture scaling and orientation are derived automatically from the gradient of the depth field unlike the case of texture type 1-a.

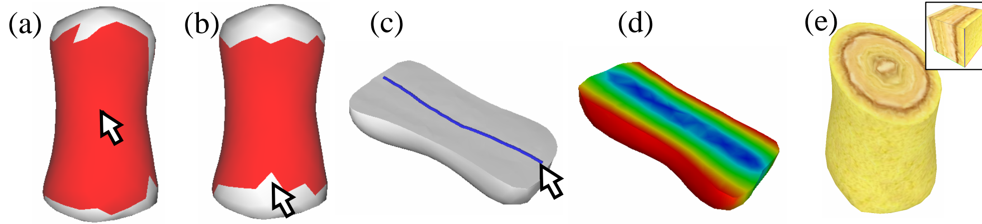

Figure 5.3: Modeling process for texture type 1-b. (a) Multi-face-fill tool. (b) Single-face-fill tool modifying the colored region. (c) Stroke tool. (d) Computed depth field. (e) Result of automatic filling.

5.4 Texture type 2-a

This type represents textures whose cross-sections have three different appearances depending on their relative orientations with respect to the local tensor field; a representative example would be flattened fibers. The user creates a tensor field over the model for this type of texture. As a tensor field is a set of three orthogonal vector fields, it is difficult for the user to create an appropriate tensor field by manually drawing strokes for each vector field separately. Therefore, we divided the process into two steps. The user first creates a depth field over the model, as described in Section 5.3 (Fig. 5.4a). The primary directions are set as the gradient directions of the depth field. Next, the user can draw strokes on each “layer” (iso-surface of the depth field) to specify the secondary directions (Fig. 5.4b). This ensures that the secondary direction is always perpendicular to the first. The third direction is set to the cross-product of the other two. The original depth field is discarded once the tensor field is computed. After the tensor field is set appropriately, the user moves on to the process of setting the texture scaling (Fig. 5.4c), followed by automatic filling (Fig. 5.4d).

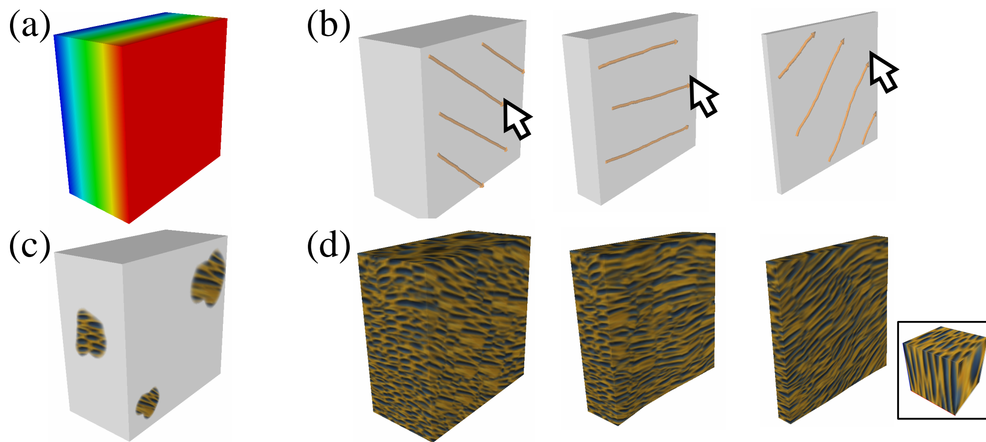

Figure 5.4: Modeling process for texture type 2-a. (a) Specifying the depth field. (b) Specifying the secondary directions by drawing strokes on layers. (c) Setting the texture scaling. (d) Result of automatic filling.

5.5 Texture type 2-b

This type of texture also represents depth-varying models, as in type 1-b, but the two perpendicular cross-sections parallel to the depth direction appear different. Examples include kiwi fruits, carrots, and trees. The modeling process is identical to type 2-a (Section 5.4), but in this case the original depth field is preserved and used in the synthesis process. In addition, the texture scaling is derived automatically from the gradient of the depth field.

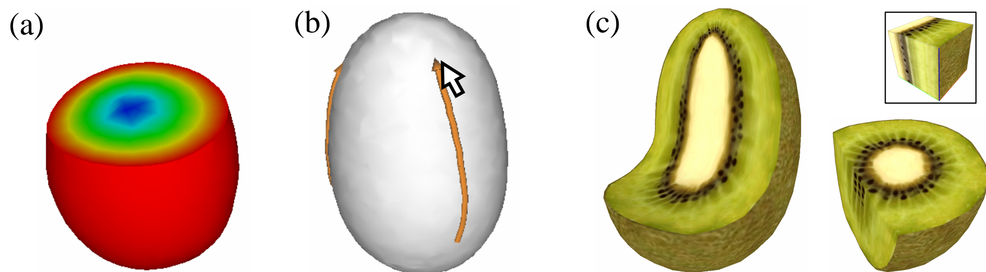

Figure 5.5: Modeling process for texture type 2-b. (a) Specifying the depth field. (b) Specifying the secondary directions by drawing strokes on layers. (c) Result of automatic filling.

5.6 Manual pasting of textures

After the system generates a solid textured model, the user can also manually paste additional solid textures onto the model. The user first loads a solid texture exemplar (Fig. 5.6a), which can then be moved and rotated on the model by dragging the mouse (Fig. 5.6b). The user can also change the texture scale interactively with the mouse wheel (Fig. 5.6c). Finally, the texture can be pasted onto the model by double-clicking.

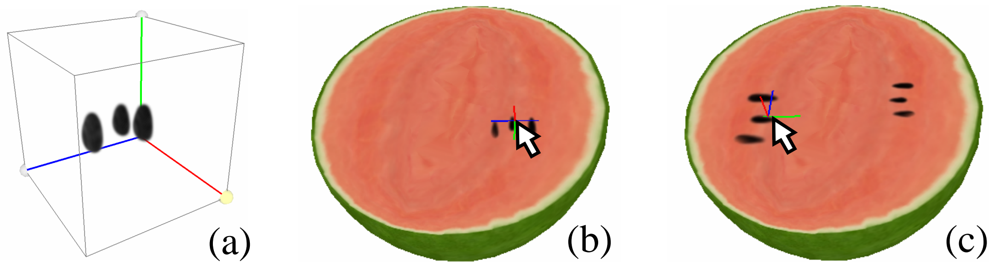

Figure 5.6: Manual pasting of additional textures. (a) Solid texture exemplar to be pasted. (b) Moving and rotating the texture patch by dragging with the mouse. (c) Changing the texture scale with the mouse wheel.

Chapter 6

Algorithm

The input to our system consists of a triangular mesh model and a solid texture exemplar. The output is a lapped solid textured (LST) model; many overlapping pieces of solid texture are pasted inside the mesh. The input mesh model is first converted to a tetrahedral mesh model. Currently, we use the TetGen library [36], which produces nearly uniform meshes using Delaunay tetrahedralization. Most of the solid texture exemplars present in this paper were created manually using the method described in Section 4.3. It is unrealistic to manually design a large volumetric texture, but we only required small exemplars and so manual editing was a viable option.

We used a tetrahedral mesh to represent solid models because this representation has certain advantages over the voxel representation for our purposes. First, it can approximate 3D shapes well with a smaller number of elements. Second, the tetrahedral mesh naturally corresponds to a triangular surface mesh when we extend the original 2D technique [34] to 3D. Finally, cross-sectioning and iso-surface extraction can be performed easily using marching tetrahedra [40], which is similar to marching cubes [22] except that it is faster and easier to implement.

We first describe how to render an LST model created in our system and then describe the process of construction of LST models in detail.

6.1 Rendering an LST model

Each tetrahedron in an LST model has a list of 3D texture coordinates assigned to each of its four vertices. To render such a model, we first convert it into a polygonal model that consists of surface triangles with a list of 3D texture coordinates assigned to each of its three vertices. We can then render this polygonal model using the same run-time compositing algorithm described previously [34]. Each surface triangle is

rendered multiple times (approximately 10–20 times in most of our results) using the texture coordinates in its assigned list, with alpha blending enabled.

6.1.1 Cutting

When the user cuts the model by drawing a freeform stroke (Fig. 6.1a), the system constructs a scalar field over the tetrahedral mesh vertices, which takes negative and positive values on the left- and right-hand sides of the stroke, respectively (Fig. 6.1b). We used radial basis function (RBF) interpolation [41] to construct such a scalar field. The cross-sectional surface is then obtained by extracting the iso-surface of value 0 from the mesh (Fig. 6.1c). The texture coordinates for each triangle on the cross-section are obtained by linearly interpolating the texture coordinates of the original tetrahedron. The tetrahedral mesh is subdivided on the cross-section to allow subsequent cutting operations.

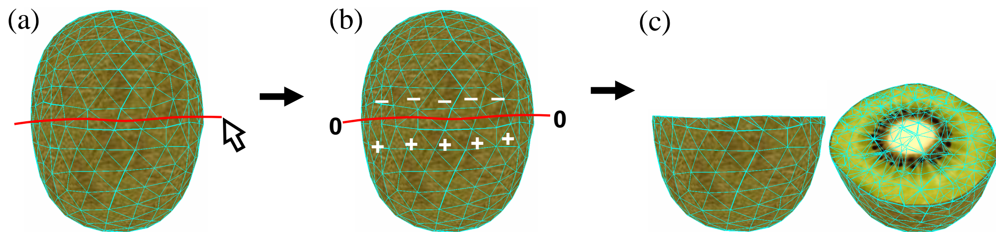

Figure 6.1: Cutting operation. (a) User-drawn stroke across the 3D model. (b) Scalar field computed from the stroke. (c) Resulting cross-sectional surface mesh.

6.1.2 Volume rendering

We can also perform volume rendering on an LST model using the same approach as described above. We first construct a scalar field over the mesh vertices to give the distance between the camera and each vertex. We then calculate a large number of slices of the model perpendicular to the camera direction by iso-surface extraction.

6.2 Construction of an LST model

The overall procedure closely follows the original [34], but each process contains non-trivial extensions, which we describe in detail in the following subsections. We first create an alpha mask of the input solid texture to make the resulting seams between pasted textures less noticeable (Section 6.2.1). We then construct a tensor field over

the mesh based on user input (Section 6.2.2). The direction and magnitude of the tensor field specify the orientation and scaling of the texture, respectively. Finally, textures are pasted repeatedly onto the model while aligning with the tensor field.

The texture pasting process is as follows. First, a seed tetrahedron is selected (Section 6.2.3). Then, we grow a clump of tetrahedra around the seed until it is large enough to cover the texture patch being pasted (Section 6.2.4). Next, we perform texture optimization which warps the pasted texture so that it aligns locally with the tensor field (Section 6.2.5). Finally, we update the coverage of textures for each tetrahedron (Section 6.2.6).

A depth-varying solid model is a new feature in our system. We prepared several exemplar textures with different alpha masks and pasted them according to the depth (Section 6.2.7).

In the following subsections, we explain the details for each process.

6.2.1 Creating an alpha mask of the solid texture

In the original 2D case, Praun et al. [34] created an alpha mask of the 2D texture using a standard image editing tool. For a less-structured texture, they used a "splotch" mask independent of the content of the texture. For a highly structured texture, they created an appropriate alpha mask that preserved the important features of the texture as much as possible.

In our 3D case, we manually created an alpha mask of the solid texture by modeling a 3D shape of the mask using existing 3D modeling techniques, such as that reported by Nealen et al. [25] (Fig. 6.2a). This mask is the 3D version of the "splotch" mask in the 2D case, which can be applied to a less-structured solid texture (Fig. 6.2b). The alpha value drops off around the boundary of the mask, which makes the resulting seams between pasted textures less noticeable. We found that an appropriate width of this drop-off is about 5–10% of the texture size in our experiments.

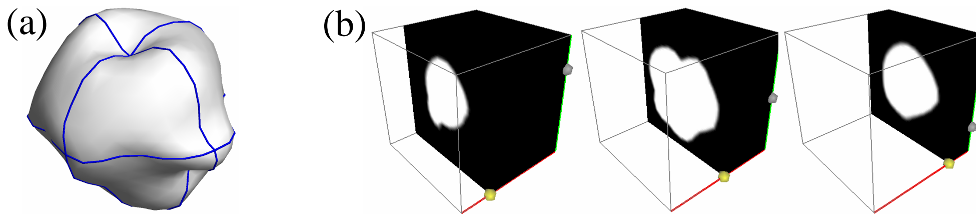

Figure 6.2: Manual creation of a 3D alpha mask. (a) 3D model of the shape of the mask. (b) Cross-sections of the alpha mask.

It is still very difficult, however, to create an appropriate alpha mask manually for a highly structured solid texture that preserves the important features of the texture as much as possible. For now, we assume all the textures in our examples are less structured, and therefore we use a constant "splotch" mask shown in Figure 6.2 for all the textures. However, this assumption often causes some artifacts when using highly structured textures, and this will be discussed in detail in Section 7.1.

6.2.2 Constructing a tensor field

This process depends on the texture type. In the case of texture type 0, the system does not create a consistent global tensor field and pastes a texture in a random orientation each time. In the case of texture types 1-a and 1-b, the first direction is globally defined according to the user-drawn strokes (1-a) or is set to the gradient direction of the depth field (1-b), and the other direction is chosen randomly when pasting each patch. In the case of texture types 2-a and 2-b, the system defines a global tensor field whose first direction is set to the gradient direction of the depth field with the second direction specified by the user-drawn strokes. The third direction of the tensor is set to the cross product of the two. The magnitudes of tensors are set to the user-specified texture scaling values, except for types 1-b and 2-b where the texture scaling is set automatically from the depth field (see Section 6.2.7 for these cases).

The original 2D lapped textures used Gaussian RBF over the mesh surface for interpolation of user-specified vectors, but we used Laplacian smoothing on the tetrahedral mesh vertices to interpolate user-specified vectors and scaling values, because this allows more detailed control over the interpolation process by adjusting weight parameters. After obtaining tensors at the mesh vertices, the tensor of a tetrahedron is given as the average of the tensors of its four vertices.

Laplacian smoothing [10] minimizes the difference between the value assigned to each vertex and the weighted average of the values assigned to its neighboring vertices while satisfying the user-specified constraints as much as possible. More precisely, suppose we are solving for the texture scaling values x_i assigned to each vertex $\mathbf{v}_i$ ($i = 1, \ldots, n$). The Laplacian δ_i is then defined as

$$\delta_i = x_i - \sum_{j \in N_i} w_j^i x_j \tag{6.1}$$

where N_i is the index set of one-ring neighboring vertices of $\mathbf{v}_i$ and w_j^i are the corresponding weights. For now, we set $w_j^i = \frac{1}{|N_i|}$, which means that $\sum_{j \in N_i} w_j^i x_j$ is simply

the average of the values of the neighboring vertices. The goal is to minimize all these Laplacians while satisfying user-specified constraints, which are formulated as follows. When a constraint scaling value c is given at 3D position $\mathbf{p}$, we first search for a tetrahedron T in the mesh whose barycenter is closest to $\mathbf{p}$. We then calculate barycentric coordinates $\lambda_1, \ldots, \lambda_4$ on T to represent $\mathbf{p}$ as

$$\begin{aligned} \lambda_1 \mathbf{v}_{i_1} + \lambda_2 \mathbf{v}_{i_2} + \lambda_3 \mathbf{v}_{i_3} + \lambda_4 \mathbf{v}_{i_4} &= \mathbf{p} \\ \lambda_1 + \lambda_2 + \lambda_3 + \lambda_4 &= 1 \end{aligned}$$

where $i_1, \ldots, i_4$ are the indices of the four vertices of T. The constraint is then given as

$$\lambda_1 x_{i_1} + \lambda_2 x_{i_2} + \lambda_3 x_{i_3} + \lambda_4 x_{i_4} = c. \tag{6.2}$$

Minimizing Laplacians (Eq. 6.1) while satisfying the collection of constraints (Eq. 6.2) in a least squares sense forms a sparse linear system, which can be solved quickly.

For interpolation of the user-specified vectors, we perform Laplacian smoothing for each x-, y-, and z-component of the vectors, which are later combined and normalized. In the case of texture types 2-a and 2-b, there is no guarantee that resulting vectors will always be orthogonal to the first direction, i.e., the gradient direction of the depth field. Therefore, we orthogonalize these vectors to the first direction after smoothing.

In addition, note that in the case of texture types 2-a and 2-b, we alter w_j^i so that the resulting vector field is smoother on the same depth (layer) than on different depths. To achieve this, we set weights as

$$w_j^i = \frac{\exp(-(d_i - d_j)^2)}{\sum\limits_{k \in N_i} \exp(-(d_i - d_k)^2)}$$

where d_i is the depth value assigned to $\mathbf{v}_i$.

While we use Laplacian smoothing for the vectors and scaling values, we use thin-plate RBF interpolation in the 3D Euclidean space [41] to obtain a depth field. This is because the depth field must be defined as a smooth function in 3D space to calculate its gradient directions accurately. We assign depth values of 0 and 1 to the outermost (red) and the innermost (blue) regions, respectively. In the case of texture types 1-b and 2-b, these depth values are used directly as one of the three texture coordinates (see Section 6.2.7 for details).

6.2.3 Selecting a seed tetrahedron

We first initialize a list of "uncovered" tetrahedra with all the tetrahedra in the mesh. For each pasting operation, one is selected at random from this list as a seed tetra-

hedron. After the pasting operation, tetrahedra are removed from the list if they are completely covered by the previously pasted textures. We repeat this process until the "uncovered" list becomes empty. In the case of manual pasting of the textures, the seed tetrahedron is set to the one clicked by the user.

6.2.4 Growing a clump of tetrahedra

We first map the seed tetrahedron from the geometric space into the texture space, so that its mapped tensor axes align with the standard axes of the texture space, and its transformed central position is located in the center of the texture.

Let $(\mathbf{R}, \mathbf{S}, \mathbf{T})$ be the three orthogonal vectors of the tensor associated with the seed tetrahedron T. We first compute barycentric coordinates $r_1, \ldots, r_4$ on T to represent $\mathbf{R}$ as

$$\begin{aligned} r_1\mathbf{v_1} + r_2\mathbf{v_2} + r_3\mathbf{v_3} + r_4\mathbf{v_4} &= \mathbf{R} \\ r_1 + r_2 + r_3 + r_4 &= 0 \end{aligned}$$

where $\mathbf{v_1}, \ldots, \mathbf{v_4}$ are the four vertices of T. We do the same with $\mathbf{S}$ and $\mathbf{T}$. We then compute the transformed vertex positions $\mathbf{w_1}, \ldots, \mathbf{w_4}$ in the texture space by solving the following equations

$$\begin{aligned} r_1\mathbf{w_1} + r_2\mathbf{w_2} + r_3\mathbf{w_3} + r_4\mathbf{w_4} &= (1, 0, 0)^t \\ s_1\mathbf{w_1} + s_2\mathbf{w_2} + s_3\mathbf{w_3} + s_4\mathbf{w_4} &= (0, 1, 0)^t \\ t_1\mathbf{w_1} + t_2\mathbf{w_2} + t_3\mathbf{w_3} + t_4\mathbf{w_4} &= (0, 0, 1)^t \\ c_1\mathbf{w_1} + c_2\mathbf{w_2} + c_3\mathbf{w_3} + c_4\mathbf{w_4} &= (0.5, 0.5, 0.5)^t \end{aligned}$$

where $c_1, \ldots, c_4$ are the barycentric coordinates on T, which represent the position inside T where the center of the texture should be. In the case of automatic filling, the position is set to the barycenter of T ($c_1 = \cdots = c_4 = 0.25$), while it is set to the user-specified position in the case of manual pasting. After appropriate transformation of vertex positions, we finally compute an affine transform matrix M that maps $\mathbf{v_i}$ to $\mathbf{w_i}$.

Next, we grow the clump by adding adjacent tetrahedra. We visit each tetrahedron around the clump and add it to the clump if the tetrahedron satisfies the following two conditions: its tensor is not markedly different from that of the seed, and it is partially inside the alpha mask in the texture space when transformed by M.

6.2.5 Texture optimization

The purpose of texture optimization is to warp the texture so that it aligns locally with the tensor field. More precisely, for each tetrahedron in the clump, we minimize the difference between the tensor axes of the tetrahedron transformed into the texture space and the standard texture coordinate axes.

The input to this process is a clump of tetrahedra $\{T_i\}$ and its associated tensors $\{(\mathbf{R_i}, \mathbf{S_i}, \mathbf{T_i})\}$ $(i = 1, \ldots, n)$. The output is the 3D texture coordinates $\{\mathbf{w_j}\}$ for all the vertices $\{\mathbf{v_j}\}$ $(j = 1, \ldots, m)$ in the clump.

For each T_i, we first compute the barycentric coordinates r_k^i, which represent $\mathbf{R_i}$ in the same way as described in Section 6.2.4. We then define the difference vector $\mathbf{d_r^i}$ between the transformed tensor axis $\mathbf{R_i'}$ and the standard texture axis $\hat{\mathbf{r}}$ as

$$\mathbf{d_r^i} = r_1^i \mathbf{w_{j_1}} + r_2^i \mathbf{w_{j_2}} + r_3^i \mathbf{w_{j_3}} + r_4^i \mathbf{w_{j_4}} - (1, 0, 0)^t$$

where $j_1, \ldots, j_4$ are the indices of the four vertices of T_i. We do the same for the s and t directions (Fig. 6.3). We minimize all these difference vectors, while satisfying the positional constraint given to the seed tetrahedron in the same way as described in Section 6.2.4. The optimized solution $\{\mathbf{w_j}\}$ can be obtained quickly in a least squares sense by solving a sparse linear system.

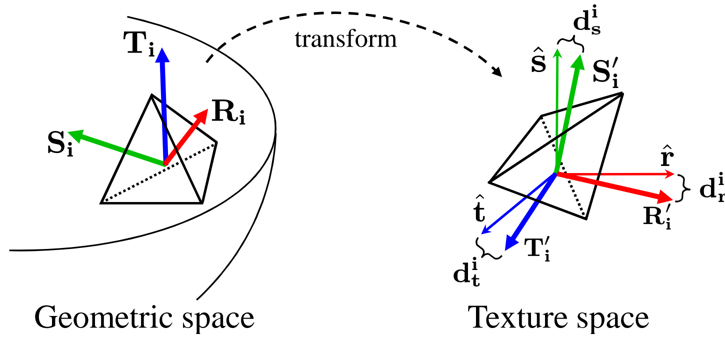

Figure 6.3: The optimization minimizes the difference vectors $\mathbf{d_r^i}, \mathbf{d_s^i}, \mathbf{d_t^i}$ between the texture coordinate axes $(\hat{\mathbf{r}}, \hat{\mathbf{s}}, \hat{\mathbf{t}})$ and the transformed tensor axes $(\mathbf{R_i'}, \mathbf{S_i'}, \mathbf{T_i'})$.

Note that the optimization may warp the texture coordinates such that the image of the clump in the texture space no longer fully covers the splotch mask. In such cases, we add the lacking tetrahedra to the clump and re-compute the optimization.

6.2.6 Coverage test of tetrahedron

The original 2D method [34] used a rasterization technique to test the coverage of overlapping textures. We perform a similar computation over sampling points inside the tetrahedra. We first create several predefined discrete sampling points (165 points in our current prototype) inside each tetrahedron in the mesh (Fig. 6.4). Each time a texture is pasted, we linearly sample the alpha values of the mask at these discrete points of each tetrahedron in the clump, which are then accumulated. If the accumulated alpha values of all the sampling points of a tetrahedron reach 255, we assume that the tetrahedron is completely covered by the overlapping textures.

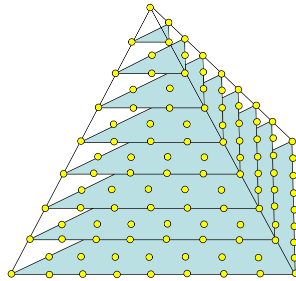

Figure 6.4: Predefined sampling points used for coverage test of overlapping texture patches.

6.2.7 Creation of depth-varying solid models

We create depth-varying solid models by arranging a depth-varying solid texture so that it aligns with the depth field defined over the target 3D model. The basic concept is to map the clump of tetrahedra into the corresponding depth position in the texture space instead of the central position. To achieve this, we alter the positional constraints in Section 6.2.4 and 6.2.5 from $(0.5, 0.5, 0.5)^t$ to $(0.5, d_{seed}, 0.5,)^t$, where d_{seed} is the depth value assigned to T_{seed} assuming the s-axis corresponds to the depth orientation. However, a problem occurs when we paste textures onto the inner and outer parts of the model (Fig. 6.5a), because the alpha mask covers only the middle part of the texture.

To solve this problem, we prepared solid textures corresponding to different layers of the original texture, each with different alpha masks (Fig. 6.6). These were created by simply applying the same alpha mask to different places (outer, middle, and inner). In the texture pasting process, an appropriate texture is chosen from these three according to the depth value of the seed tetrahedron (Fig. 6.5b).

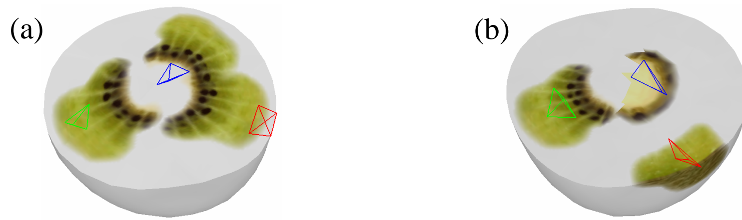

Figure 6.5: (a) A problem occurs if we use only a single alpha mask. (b) The use of three types of alpha mask solves this problem.

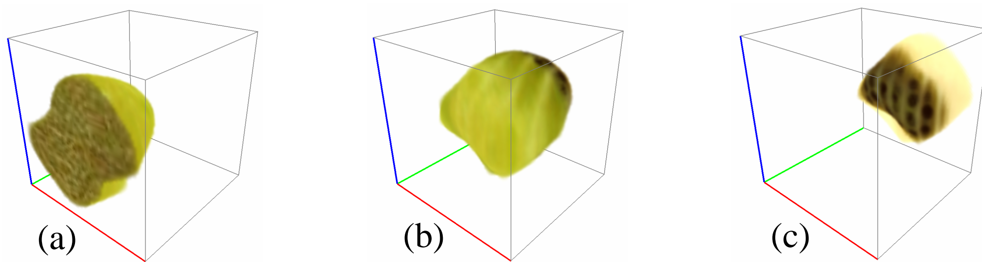

Figure 6.6: Three types of alpha mask: (a) outer part, (b) middle part, and (c) inner part.

The depth values defined over the 3D model can be used directly as the texture coordinates of the depth direction, so we only solve for texture coordinates of the other two directions. We have also found that the appropriate texture scaling is the inverse of the magnitude of the depth gradient vector. This can be explained as follows. Suppose we have a large depth gradient vector at a certain position in the 3D model. This means that the depth value changes rapidly there, which also implies that the region corresponds to the thin part of the 3D model. Therefore, the texture scale should be small.

Chapter 7

Results

As shown in Figures 7.1 and 7.2, our results showed consistency among different cross-sections, which was not seen in the work of Owada et al. [29]. Models in Figures 7.1a and 7.2a contain many seeds, which can cause artifacts in the work of Pietroni et al. [33]. Texture type 2-b is used in all the results in Figure 7.1, and the appearance of the cross-sections differs depending on the orientation with respect to the radial axes. Most textures of type 1-b and 2-b in our results have a thin (1–3 voxel thickness) slice of outer skin, and our depth adjustment technique arranges solid textures successfully so that such outer skin regions align precisely with the surfaces of the models. Cross-sectioning is faster than run-time synthesis approaches because the computation only involves linear sampling of texture coordinates. We can create more complex solid models by combining several LST models together (Figs. 7.2c and 7.2d). We can also perform volume rendering on translucent LST models (Fig. 7.2b), which is impossible when using inconsistent quasi-solid models. This result was obtained by taking 200 slices from the model, a process that took about 3 s. Our method can be extended easily to support other channels of textures, and we show the displacement mapping results in Figures 7.1c and 5.1d where the grayscale displacement map channel is shown next to the RGB texture. This is done by first subdividing the surface mesh and then moving each vertex along its normal direction according to the displacement value sampled there.

We implemented our prototype system using C++ and OpenGL on a notebook PC with a 2.3-GHz CPU and 1.0 GB of RAM. The statistics of our results are summarized in Table 7.1, which shows that our method is fairly inexpensive in terms of both computation and memory for representing large-scale solid models. It took a relatively long time to fill a cake model (Fig. 7.2d), because the model has a large thin sponge

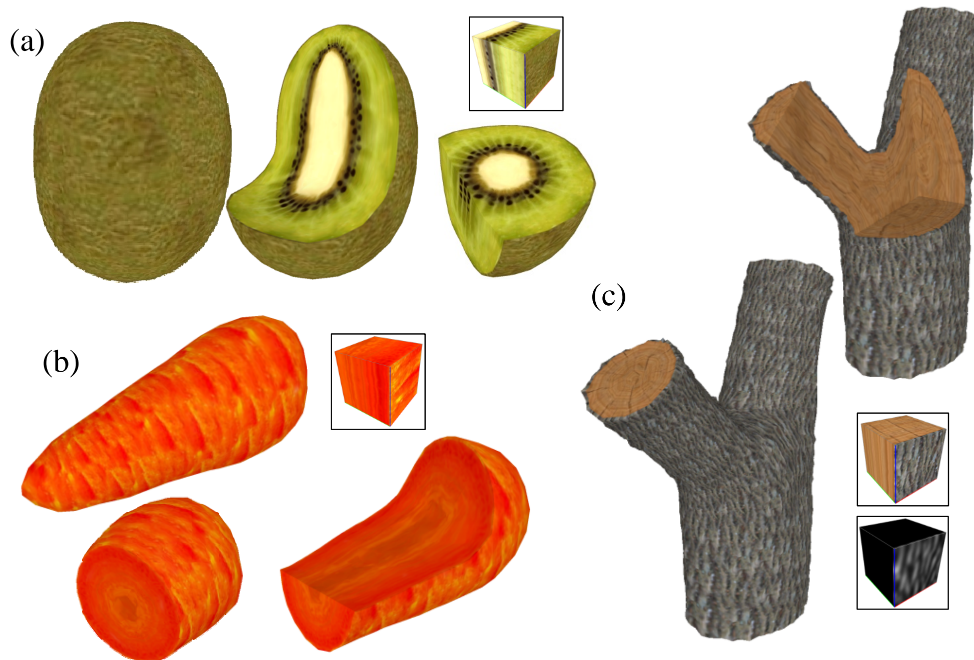

Figure 7.1: Models filled with overlapping solid textures: (a) kiwi fruit, (b) carrot, and (c) tree (the grayscale texture represents the displacement map channel). Note that the input solid textures include surface textures as well as interior textures.

region that requires pasting a large number of texture patches. However, the rendering and cross-sectioning could still be performed in real-time.

7.1 Limitations

Our method inherits the limitations of the original method [34]. First, the patch seams become noticeable when using a texture with strong low-frequency components. Second, artifacts appear around singularities of the tensor field, such as the center of a depth-varying object with a radial axis (Fig. 7.3). This can be alleviated by locally subdividing tetrahedra in such areas. Finally, as we use a constant "splotch" mask for all the textures, blurring artifacts appear when a highly structured texture is used (Fig. 7.4b). It is necessary to create an appropriate alpha mask that preserves the structure of the texture as much as possible, and this may be achieved by extending the existing 2D contour detection technique [16] to 3D. It is also necessary to consider the alignment between texture patches to avoid misalignment artifacts (Fig. 7.4c).

Title	Tetra	Design [sec]	Fill [sec]	Cut [msec]	Size [MB]
Kiwi fruit	4126	29	39	78	9.1
Carrot	2313	38	31	63	7.1
Tree	5012	76	104	125	12.2
Watermelon	2717	17	25	63	9.0
Tube	1089	27	18	31	2.7
Strata	2827	113	77	110	10.4
Cake	2734	34	416	187	14.5

Table 7.1: Statistics of our results. Column describe (from left to right): title, number of tetrahedron, time for tensor field design, time for automatic filling, time for cross-sectioning (without subdivision), and total data size of LST model (including texture exemplars). The size of texture exemplars was 64^3 throughout.

Soler et al. [37] proposed hierarchical pattern mapping, which considers the coherency between texture patches on surfaces, but extending their technique to 3D solid appears to be nontrivial.

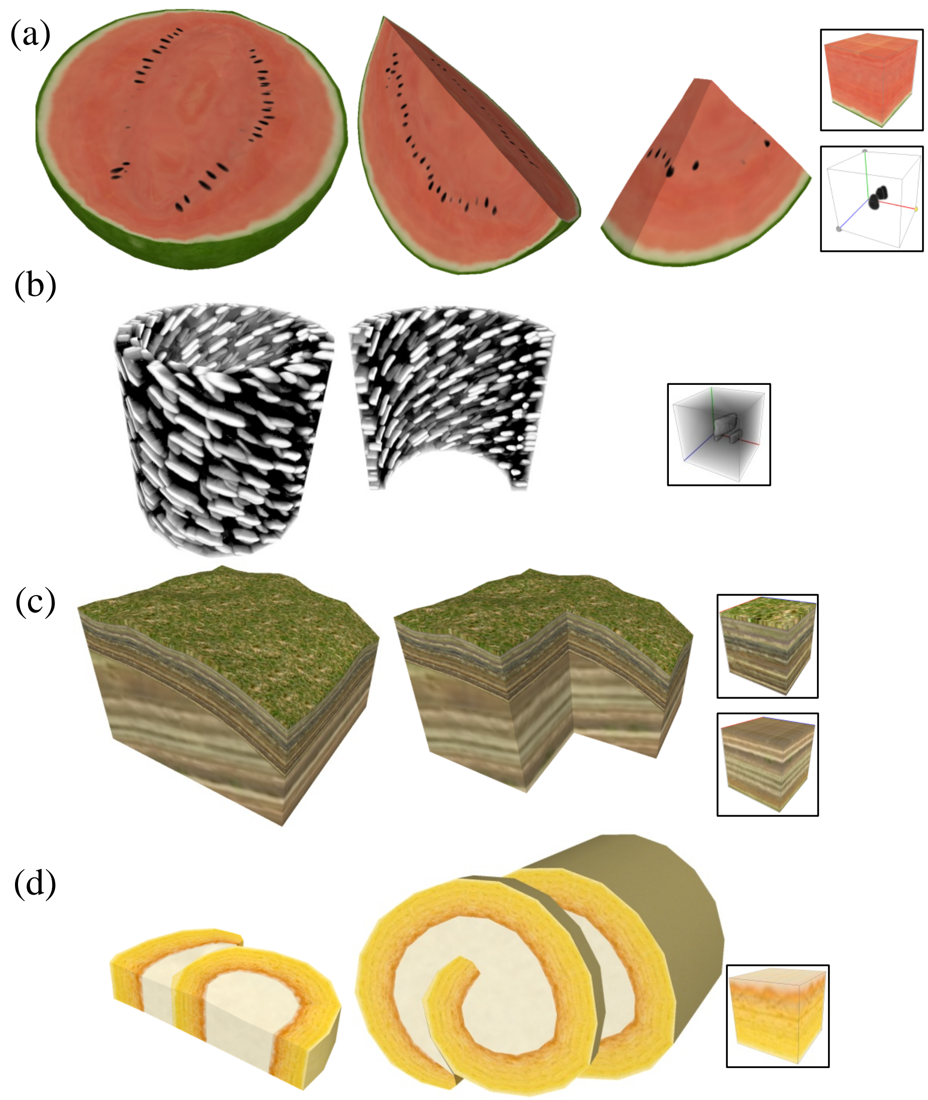

Figure 7.2: Results of our method. (a) Watermelon. (b) Volume rendering of a fibrous tube. (c) Strata. (d) Cake.

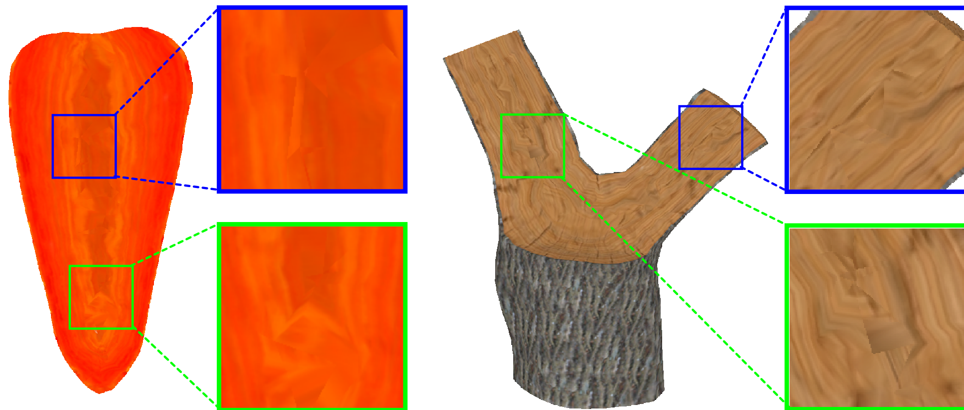

Figure 7.3: Artifacts around tensor field singularities.

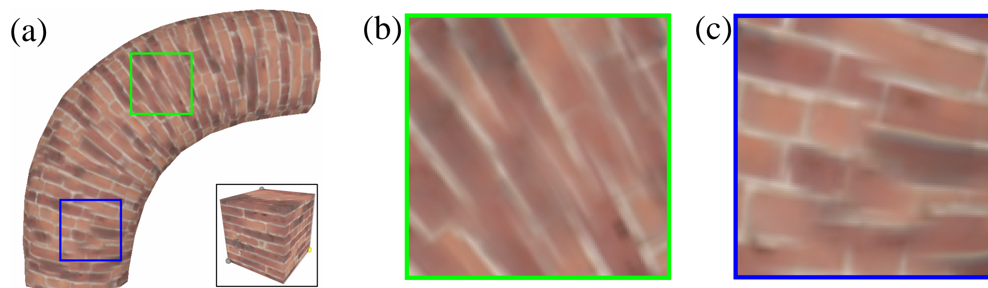

Figure 7.4: Failure case with a highly structured texture. (a) A curved cylinder filled with bricks shows (b) blurring and (c) misalignment artifacts.

Chapter 8

Conclusions

We proposed in this thesis a novel method for synthesizing large-scale solid textured models efficiently by extending the approach of lapped textures to 3D solids. We classified solid textures into several types according to their anisotropy and spatial variation. Based on this classification, we provided a different user interface for each texture type to specify the texture orientation and scaling. Our method also handles nonhomogeneous textures by aligning texture patches to the depth field. Using our method, various solid textured models, such as kiwi fruits, carrots, and trees, can be created with little memory and computation.

8.1 Future Work

Our future work obviously includes the preparation of exemplar solid textures, which is still an unsolved problem especially for organic objects. First, many organic objects, such as kiwi fruits, contain translucent regions, and the color seen on a cross-section depends on the materials beneath it. Another problem is that objects such as carrots that contain fibrous structures can cause anisotropic reflection; the appearance of these fibers differs depending on the orientation of the cross-section. The existing texture synthesis methods from 2D exemplars assume consistent color of a given voxel and cannot handle such cases. A sort of inverse volume rendering may be necessary to obtain volumetric representation from 2D photographs.

Another possible future direction is to explore more various ways of interacting with LST models, other than the simple cutting by a stroke [23, 27]. Depending on the application, the user may want to peel, carve, tear, or smash LST models. In some cases fracture simulations [30] of LST models may be necessary. It may also be possible to apply more sophisticated lighting simulations on LST models such as

subsurface scattering of heterogeneous materials [3, 31].

References

[1] Richard A. Banvard. The visible human project ®image data sets from inception to completion and beyond. In *Proc. of CODATA 2002: Frontiers of Scientific and Technical Data*, 2002.

[2] James F. Blinn. Simulation of wrinkled surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 286–292, New York, NY, USA, 1978. ACM.

[3] Yanyun Chen, Xin Tong, Jiaping Wang, Stephen Lin, Baining Guo, and Heung-Yeung Shum. Shell texture functions. *ACM Trans. Graph.*, 23(3):343–353, 2004.

[4] Robert L. Cook. Shade trees. *SIGGRAPH Comput. Graph.*, 18(3):223–231, 1984.

[5] Robert L. Cook and Tony DeRose. Wavelet noise. *ACM Trans. Graph.*, 24(3):803–811, 2005.

[6] Barbara Cutler, Julie Dorsey, Leonard McMillan, Matthias Müller, and Robert Jagnow. A procedural approach to authoring solid models. *ACM Trans. Graph.*, 21(3):302–311, 2002.

[7] Kristin J. Dana, Bram van Ginneken, Shree K. Nayar, and Jan J. Koenderink. Reflectance and texture of real-world surfaces. *ACM Trans. Graph.*, 18(1):1–34, 1999.

[8] J.M. Dischler, D. Ghazanfarpour, and R. Freydier. Anisotropic solid texture synthesis using orthogonal 2d views. *Computer Graphics Forum*, 17(3):87–95, 1998.

[9] Yue Dong, Sylvain Lefebvre, Xin Tong, and George Drettakis. Lazy solid texture synthesis. *Computer Graphics Forum*, 27(4):1165–1174, 2008.

[10] Hongbo Fu, Yichen Wei, Chiew-Lan Tai, and Long Quan. Sketching hairstyles. In *Proc. of Fourth Eurographics Workshop on Sketch-Based Interfaces and Modeling*, 2007.

[11] Djamchid Ghazanfarpour and Jean-Michel Dischler. Generation of 3d texture using multiple 2d models analysis. *Computer Graphics Forum*, 15(3):311–323, 1996.

[12] David J. Heeger and James R. Bergen. Pyramid-based texture analysis/synthesis. In *Proc. of SIGGRAPH '00*, pages 229–238, 1995.

[13] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: a sketching interface for 3d freeform design. In *Proc. of SIGGRAPH '99*, pages 409–416, 1999.

[14] Robert Jagnow, Julie Dorsey, and Holly Rushmeier. Stereological techniques for solid textures. *ACM Trans. Graph.*, 23(3):329–335, 2004.

[15] J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. *SIGGRAPH Comput. Graph.*, 23(3):271–280, 1989.

[16] Michael Kass, Andrew Witkin, and Demetri Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321–331, 1987.

[17] Johannes Kopf, Chi-Wing Fu, Daniel Cohen-Or, Oliver Deussen, Dani Lischinski, and Tien-Tsin Wong. Solid texture synthesis from 2d exemplars. *ACM Trans. Graph.*, 26(3):2, 2007.

[18] Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. Texture optimization for example-based synthesis. *ACM Trans. Graph.*, 24(3):795–802, 2005.

[19] Laurent Lefebvre and Pierre Poulin. Analysis and synthesis of structural textures. In *Proc. of Graphics Interface '00*, pages 77–86, 2000.

[20] Sylvain Lefebvre and Hugues Hoppe. Parallel controllable texture synthesis. *ACM Trans. Graph.*, 24(3):777–786, 2005.

[21] Jerome Lengyel, Emil Praun, Adam Finkelstein, and Hugues Hoppe. Real-time fur over arbitrary surfaces. In *Proc. of the 2001 symposium on Interactive 3D graphics*, pages 227–232, 2001.

[22] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proc. of SIGGRAPH '87*, pages 163–169, 1987.

[23] Michael J. McGuffin, Liviu Tancau, and Ravin Balakrishnan. Using deformations for browsing volumetric data. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 53, Washington, DC, USA, 2003. IEEE Computer Society.

[24] David M. Mount and Sunil Arya. Ann: A library for approximate nearest neighbor searching, Aug 2006.

[25] Andrew Nealen, Takeo Igarashi, Olga Sorkine, and Marc Alexa. Fibermesh: designing freeform surfaces with 3d curves. *ACM Trans. Graph.*, 26(3):41, 2007.

[26] Fabrice Neyret. Modeling, animating, and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):55–70, 1998.

[27] Shigeru Owada, Ayumi Akaboya, Frank Nielsen, Fusako Kusunoki, and Takeo Igarashi. Kiru ("cut", in japanese). In *12th Workshop on Interactive Systems and Software (WISS 2004)*, pages 1–4, 2004.

[28] Shigeru Owada, Takahiro Harada, Philipp Holzer, and Takeo Igarashi. Volume painter: Geometry-guided volume modeling by sketching on the cross-section. In *Proceedings of Eurographics Symposium on Sketcy-Based Interfaces and Modeling*, pages 9–16, 2008.

[29] Shigeru Owada, Frank Nielsen, Makoto Okabe, and Takeo Igarashi. Volumetric illustration: designing 3d models with internal textures. *ACM Trans. Graph.*, 23(3):322–328, 2004.

[30] Mark Pauly, Richard Keiser, Bart Adams, Philip Dutré, Markus Gross, and Leonidas J. Guibas. Meshless animation of fracturing solids. *ACM Trans. Graph.*, 24(3):957–964, 2005.

[31] Pieter Peers, Karl vom Berge, Wojciech Matusik, Ravi Ramamoorthi, Jason Lawrence, Szymon Rusinkiewicz, and Philip Dutré. A compact factored representation of heterogeneous subsurface scattering. *ACM Trans. Graph.*, 25(3):746–753, 2006.

[32] Ken Perlin. An image synthesizer. In *Proc. of SIGGRAPH '85*, pages 287–296, 1985.

[33] Nico Pietroni, Miguel A. Otaduy, Bernd Bickel, Fabio Ganovelli, and Markus Gross. Texturing internal surfaces from a few cross sections. *Computer Graphics Forum*, 26(3):637–644, 2007.

[34] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. In *Proc. of SIGGRAPH '00*, pages 465–470, 2000.

[35] Xuejie Qin and Yee-Hong Yang. Aura 3d textures. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):379–389, 2007.

[36] Hang Si. On refinement of constrained delaunay tetrahedralizations. In *Proc. of the 15th International Meshing Roundtable*, pages 509–528, 2006.

[37] Cyril Soler, Marie-Paule Cani, and Alexis Angelidis. Hierarchical pattern mapping. *ACM Trans. Graph.*, 21(3):673–680, 2002.

[38] Kenshi Takayama, Takashi Ashihara, Takashi Ijiri, Takeo Igarashi, Ryo Haraguchi, and Kazuo Nakazawa. A sketch-based interface for modeling myocardial fiber orientation that considers the layered structure of the ventricles. *The Journal of Physiological Sciences*, 58(7):487–492, 2008.

[39] Kenshi Takayama, Makoto Okabe, Takashi Ijiri, and Takeo Igarashi. Lapped solid textures: filling a model with anisotropic textures. *ACM Trans. Graph.*, 27(3):1–9, 2008.

[40] G. M. Treece, R. W. Prager, and A. H. Gee. Regularised marching tetrahedra: improved iso-surface extraction. *Computers and Graphics*, 23(4):583–598, 1999.

[41] Greg Turk and James F. O'Brien. Shape transformation using variational implicit functions. In *Proc. of SIGGRAPH '99*, pages 335–342, 1999.

[42] Li-Yi Wei. *Texture synthesis by fixed neighborhood searching.* PhD thesis, Stanford University, 2002.