Lapped Solid Textures: Filling a Model with Anisotropic Textures

Kenshi Takayama*
The University of Tokyo

Makoto Okabe
The University of Tokyo

Takashi Ijiri
The University of Tokyo

Takeo Igarashi
The University of Tokyo,
JST/ERATO

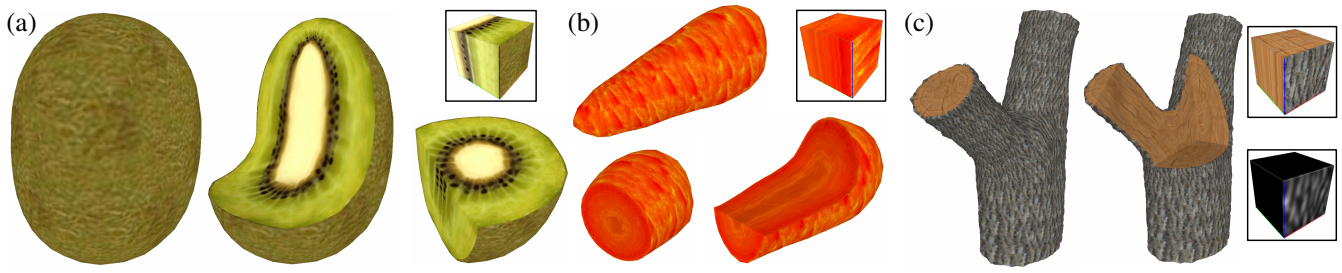


Figure 1: *Models filled with overlapping solid textures: (a) kiwi fruit, (b) carrot, and (c) tree (the grayscale texture represents the displacement map channel). Note that the input solid textures include surface textures as well as interior textures.*

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 fields over the mesh to represent oriented textures. We have also extended the original technique to handle nonhomogeneous textures for creating solid models whose textural patterns change gradually along the depth fields. We identify several texture types considering the amount of anisotropy and spatial variation and provide a tailored user interface for each. With our simple framework, large-scale realistic solid 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.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: lapped textures, solid texture, tensor field

1 Introduction

There are three main approaches to creating solid textured models: the procedural approach, run-time 2D texture synthesis on cross-sections, and example-based 3D solid texture synthesis. The first approach (e.g., [Perlin 1985; Cutler et al. 2002]) allows the design of an arbitrary solid texture by writing an explicit program, but this is difficult for non-expert users. The second approach (e.g., [Owada et al. 2004; Pietroni et al. 2007]) provides efficient and intuitive ways to create quasi-solid 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., [Jagnow et al. 2004; Kopf et al. 2007]) allows the user to create realistic and consistent solid 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 solid models because the number of voxels grows cubically as the texture size increases.

Our goal is to create large-scale solid models efficiently using 3D solid texture exemplars. The basic concept is to extend the 2D texture patch-pasting approach of lapped textures [Praun et al. 2000] 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. 2), 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 2: *Photographs of a kiwi fruit. The appearance of the cross-section differs depending on the orientation of the cutting plane.*

*E-mail: kenshi@ui.is.s.u-tokyo.ac.jp

Although the original technique was limited to homogeneous textures, we have extended it to handle spatially-varying textures. By considering the depth of the layers during the texturing process, we can create depth-varying solid models for objects such as kiwi fruit, 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 provide a tailored user interface and synthesis algorithm for each. 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.

2 Related work

One common approach to creating solid textured models is to use procedural methods. Earlier work by Perlin [1985] produced realistic solid textures by developing material-specific mathematical models using noise functions. Cutler et al. [2002] created layered solid models by specifying depth and material information in a scripting language. However, these methods are not accessible to non-expert 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. [2004] 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. [2007] 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.

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 [Heeger and Bergen 1995], spectra [Ghazanfarpour and Dischler 1996], and their combination [Dischler et al. 1998]. 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 non-parametric approach was later used [Wei 2002; Qin and Yang 2007; Kopf et al. 2007], and recent work by Kopf et al. [2007] produced realistic solid textures from 2D exemplars by combining texture optimization [Kwatra et al. 2005] and histogram matching [Heeger and Bergen 1995]. There are some other approaches to solid texture synthesis including procedural shader-based methods [Lefebvre and Poulin 2000] and stereology-based methods [Jagnow et al. 2004], although they were designed for relatively limited texture types.

Solid texture synthesis has advantages over the other approaches because it can generate consistent and detailed textures from 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. In addition, a non-trivial 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 [Praun et al. 2000] to 3D solid textures. While the lapped textures technique has already been extended to 3D shell textures for real-time furs on surfaces [Lengyel et al. 2001], to our knowledge, application of this approach to solid textures has not been explored previously.

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 Fig. 3, our classification is based on two aspects of solid textures: *anisotropy level* and *variation level*.

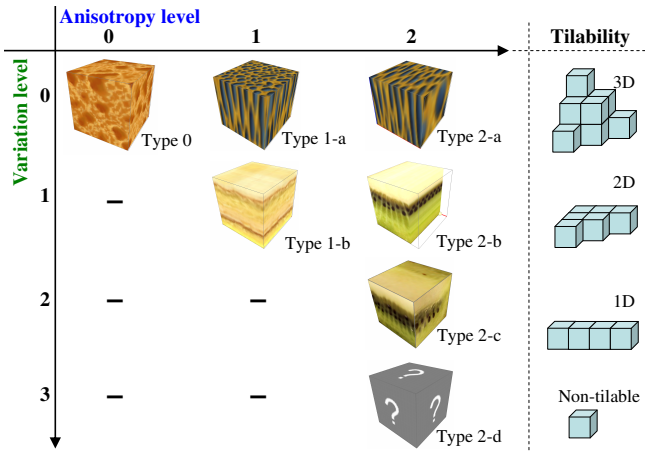


Figure 3: *Our classification of solid textures.*

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.

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.

Note that 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 [Kopf et al. 2007] or "oriented" textures [Owada et al. 2004]. Type 1-b corresponds to "layered" textures [Owada et al. 2004]. 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.

4 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 [Igarashi et al. 1999]. 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 subsections. Note that the details of the algorithm are described in Section 5.

4.1 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. 4a). The user can also change the texture scale interactively using the mouse wheel (Fig. 4b). 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. 4c), the system fills the model with the texture taking into account such user-specified texture scaling (Fig. 4d).

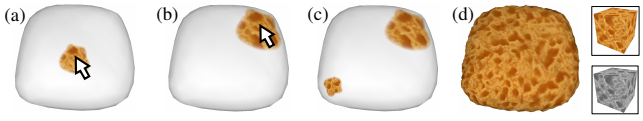


Figure 4: *Modeling process for texture type 0. (a) Moving the texture patch by dragging with the mouse. (b) Changing the texture scale with the mouse wheel. (c) User-specified texture scaling. (d) Result of automatic filling (rendered with displacement mapping).*

4.2 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 field (Fig. 5a). A similar interface was described previously [Owada et al. 2004]. After several strokes are drawn, the user then sets the texture scaling (Fig. 5b) as described in Section 4.1. Finally, the system fills the model with the texture (Fig. 5c).

Figure 5: *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.*

4.3 Texture type 1-b

This type represents models with depth-varying texture, such as cakes and watermelons. The user specifies a depth field over the model using a paint-like user interface similar to that reported by Owada et al. [2004]. 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. 6a). The single-face-fill tool assigns a color to a single surface triangle clicked by the user (Fig. 6b), 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. 6c). 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. 6d). After the depth field is set appropriately, the system then fills the model with the texture while considering the depth (Fig. 6e). Texture scaling and orientation are derived automatically from the gradient of the depth field unlike the case of texture type 1-a.

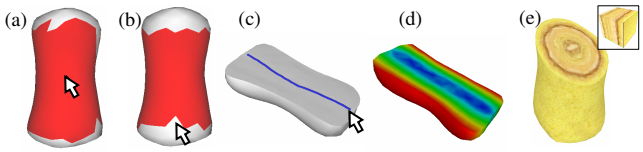


Figure 6: *Modeling process for texture type 1-b. (a) Multi-face-fill tool. (b) Single-face-fill tool modifying the colored region. (c) Stroke tool. (d) Computed depth field. (e) Result of automatic filling.*

4.4 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 4.3 (Fig. 7a). The primary directions are set as the gradient directions of the depth field. Next, the user can draw strokes on each "layer" (isosurface of the depth field) to specify the secondary directions (Fig. 7b). 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. 7c), followed by automatic filling (Fig. 7d).

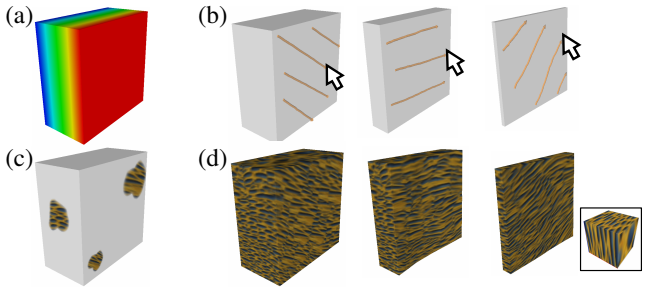


Figure 7: *Modeling process for texture type 2-a. (a) Specifying the depth field. (b) Specifying the secondary directions by drawing strokes on layers. (c) Setting the texture scaling. (d) Result of automatic filling.*

4.5 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 fruit, carrots, and trees. The modeling process is identical to type 2-a (Section 4.4), 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.

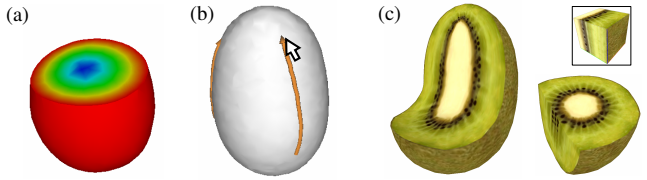


Figure 8: *Modeling process for texture type 2-b. (a) Specifying the depth field. (b) Specifying the secondary directions by drawing strokes on layers. (c) Result of automatic filling.*

4.6 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. 9a), which can then be moved and rotated on the model by dragging the mouse (Fig. 9b). The user can also change the texture scale interactively with the mouse wheel (Fig. 9c). Finally, the texture can be pasted onto the model by double-clicking.

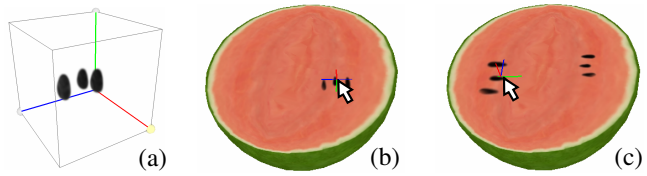


Figure 9: *Manual pasting of additional textures. (a) Solid texture exemplar to be pasted. (b) Moving and rotating the texture patch by dragging with the mouse. (c) Changing the texture scale with the mouse wheel.*

5 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 [Si 2006], which produces nearly uniform meshes using Delaunay tetrahedralization. Preparation of solid texture exemplars is somewhat problematic, but several options are available, such as solid texture synthesis [Kopf et al. 2007], noise functions [Cook and DeRose 2005], and volume capturing using slicers [Banvard 2002]. Most of our exemplars were created manually from photographs using an in-house voxel editor. It is unrealistic to manually design a large volume of these, 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 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 [Praun et al. 2000] to 3D. Finally, cross-sectioning and iso-surface extraction can be performed easily using marching tetrahedra [Treece et al. 1999], which is similar to marching cubes [Lorensen and Cline 1987] 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.

5.1 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 [Praun et al. 2000]. 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.

5.1.1 Cutting

When the user cuts the model by drawing a freeform stroke (Fig. 10a), 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. 10b). We used radial basis function (RBF) interpolation [Turk and O'Brien 1999] 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. 10c). 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.

5.1.2 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.

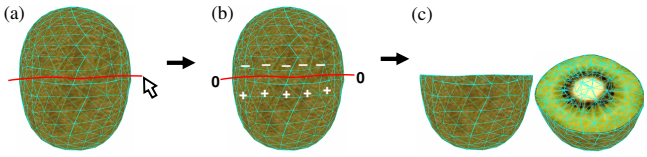


Figure 10: *Cutting operation. (a) User-drawn stroke across the 3D model. (b) Scalar field computed from the stroke. (c) Resulting cross-sectional surface mesh.*

5.2 Construction of an LST model

The overall procedure closely follows the original [Praun et al. 2000], 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 5.2.1). We then construct a tensor field over the mesh based on user input (Section 5.2.2). 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 5.2.3). Then, we grow a clump of tetrahedra around the seed until it is large enough to cover the texture patch being pasted (Section 5.2.4). Next, we perform texture optimization which warps the pasted texture so that it aligns locally with the tensor field (Section 5.2.5). Finally, we update the coverage of textures for each tetrahedron (Section 5.2.6).

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 5.2.7).

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

5.2.1 Creating an alpha mask of the solid texture

In the original 2D case, Praun et al. [2000] 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. [2007] (Fig. 11a). 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. 11b). 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.

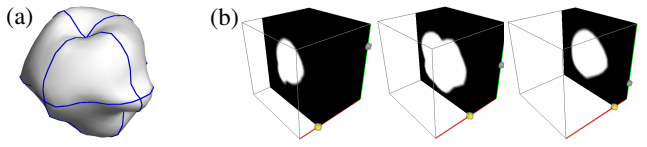


Figure 11: *Manual creation of a 3D alpha mask. (a) 3D model of the shape of the mask. (b) Cross-sections of the alpha mask.*

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 Fig. 11 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.

5.2.2 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 5.2.7 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 [Fu et al. 2007] 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

$$\delta_i = x_i - \sum_{j \in N_i} w_j^i x_j \tag{1}$$

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

$$\lambda_1 x_{i_1} + \lambda_2 x_{i_2} + \lambda_3 x_{i_3} + \lambda_4 x_{i_4} = c. \tag{2}$$

Minimizing Laplacians (Eq. 1) while satisfying the collection of constraints (Eq. 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

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

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 [Turk and O'Brien 1999] 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 5.2.7 for details).

5.2.3 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 tetrahedron. 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.

5.2.4 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.

5.2.5 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 5.2.4. 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. 12). We minimize all these difference vectors, while satisfying the positional constraint given to the seed tetrahedron in the same way as described in Section 5.2.4. The optimized solution $\{\mathbf{w_j}\}$ can be obtained quickly in a least squares sense by solving a sparse linear system.

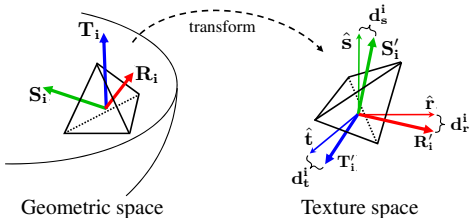
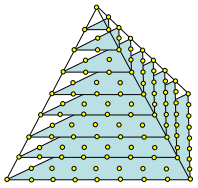


Figure 12: *The optimization minimizes the difference vectors* $\mathbf{d_r^i}, \mathbf{d_s^i}, \mathbf{d_t^i}$ *between the texture coordinate axes* $(\hat{\mathbf{r}}, \hat{\mathbf{s}}, \hat{\mathbf{t}})$ *and the transformed tensor axes* $(\mathbf{R_i'}, \mathbf{S_i'}, \mathbf{T_i'})$.

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.

5.2.6 Coverage test of tetrahedron



The original 2D method [Praun et al. 2000] 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, as shown in the figure at right. 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.

5.2.7 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 5.2.4 and 5.2.5 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. 13a), because the alpha mask covers only the middle part of the texture.

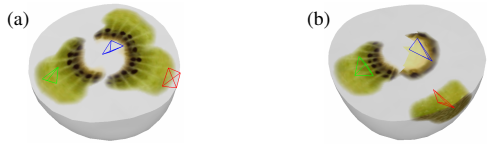


Figure 13: *(a) A problem occurs if we use only a single alpha mask. (b) The use of three types of alpha mask solves this problem.*

To solve this problem, we prepared solid textures corresponding to different layers of the original texture, each with different alpha masks (Fig. 14). 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. 13b).

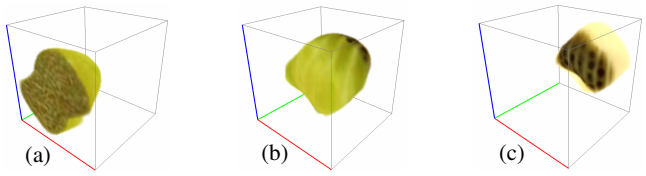


Figure 14: *Three types of alpha mask: (a) outer part, (b) middle part, and (c) inner part.*

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.

6 Results

As shown in Figs. 1 and 16, our results showed consistency among different cross-sections, which was not seen in the work of Owada et al. [2004]. Models in Figs. 1a and 16a contain many seeds, which can cause artifacts in the work of Pietroni et al. [2007]. Texture type 2-b is used in all the results in Fig. 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. 16c and 16d). We can also perform volume rendering on translucent LST models (Fig. 16b), 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 Figs. 1c and 4d 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 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. 16d), because the model has a large thin sponge region that requires pasting a large number of texture patches. However, the rendering and cross-sectioning could still be performed in real-time.

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 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.*

7 Limitations and future work

Our method inherits the limitations of the original method [Praun et al. 2000]. 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. 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. 15b). 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 [Kass et al.

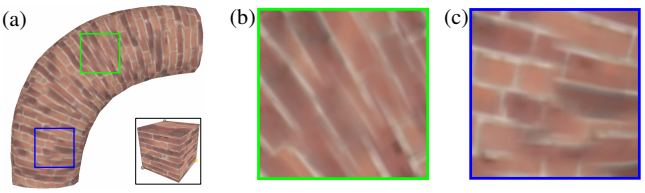


Figure 15: *Failure case with a highly structured texture. (a) A curved cylinder filled with bricks shows (b) blurring and (c) misalignment artifacts.*

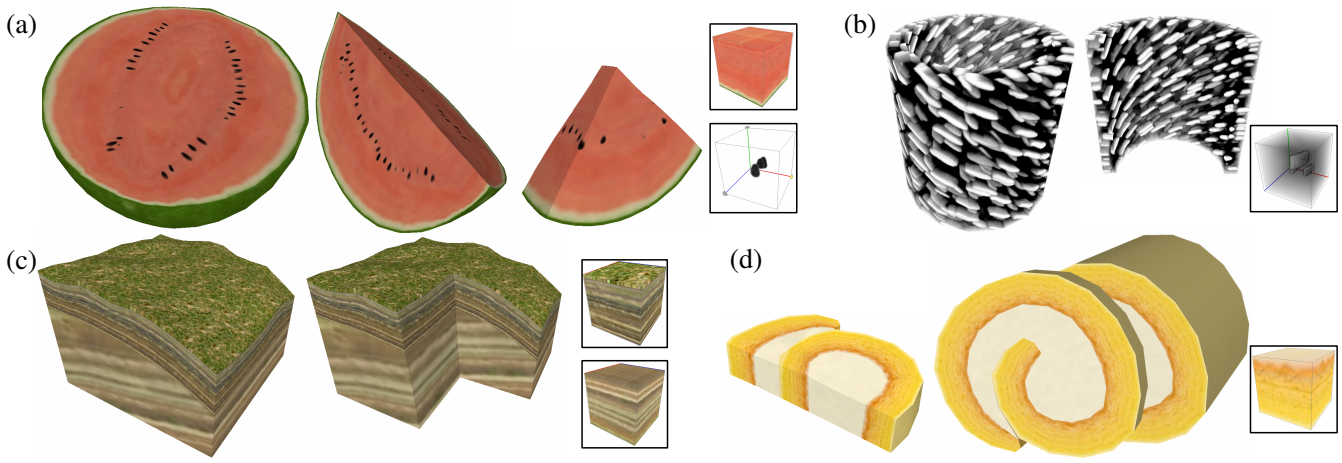


Figure 16: *Results of our method. (a) Watermelon. (b) Volume rendering of a fibrous tube. (c) Strata. (d) Cake.*

1987] to 3D. It is also necessary to consider the alignment between texture patches to avoid misalignment artifacts (Fig. 15c). Soler et al. [2002] proposed hierarchical pattern mapping, which considers the coherency between texture patches on surfaces, but extending their technique to 3D solid appears to be non-trivial.

Preparation of exemplar solid textures is still an unsolved problem, especially for organic objects. First, many organic objects, such as kiwi fruit, 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, and this is an interesting direction for future research.

Acknowledgments

We thank Shigeru Owada and Kazuo Nakazawa for their valuable comments and advice. We also appreciate the anonymous reviewers' various helpful suggestions for improving the paper. The first author was funded by the Information-technology Promotion Agency (IPA), Japan.

References

BANVARD, R. A. 2002. The visible human project (r) image data sets from inception to completion and beyond. In *Proc. of CODATA 2002: Frontiers of Scientific and Technical Data*.

COOK, R. L., AND DEROSE, T. 2005. Wavelet noise. *ACM Trans. Graph. 24*, 3, 803–811.

CUTLER, B., DORSEY, J., MCMILLAN, L., MÜLLER, M., AND JAGNOW, R. 2002. A procedural approach to authoring solid models. *ACM Trans. Graph. 21*, 3, 302–311.

DISCHLER, J., GHAZANFARPOUR, D., AND FREYDIER, R. 1998. Anisotropic solid texture synthesis using orthogonal 2d views. *Computer Graphics Forum 17*, 3, 87–95.

FU, H., WEI, Y., TAI, C.-L., AND QUAN, L. 2007. Sketching hairstyles. In *Proc. of Fourth Eurographics Workshop on Sketch-Based Interfaces and Modeling*.

GHAZANFARPOUR, D., AND DISCHLER, J.-M. 1996. Generation of 3d texture using multiple 2d models analysis. *Computer Graphics Forum 15*, 3, 311–323.

HEEGER, D. J., AND BERGEN, J. R. 1995. Pyramid-based texture analysis/synthesis. In *Proc. of SIGGRAPH '00*, 229–238.

IGARASHI, T., MATSUOKA, S., AND TANAKA, H. 1999. Teddy: a sketching interface for 3d freeform design. In *Proc. of SIGGRAPH '99*, 409–416.

JAGNOW, R., DORSEY, J., AND RUSHMEIER, H. 2004. Stereological techniques for solid textures. *ACM Trans. Graph. 23*, 3, 329–335.

KASS, M., WITKIN, A., AND TERZOPOULOS, D. 1987. Snakes: Active contour models. *International Journal of Computer Vision 1*, 4, 321–331.

KOPF, J., FU, C.-W., COHEN-OR, D., DEUSSEN, O., LISCHINSKI, D., AND WONG, T.-T. 2007. Solid texture synthesis from 2d exemplars. *ACM Trans. Graph. 26*, 3, 2.

KWATRA, V., ESSA, I., BOBICK, A., AND KWATRA, N. 2005. Texture optimization for example-based synthesis. *ACM Trans. Graph. 24*, 3, 795–802.

LEFEBVRE, L., AND POULIN, P. 2000. Analysis and synthesis of structural textures. In *Proc. of Graphics Interface '00*, 77–86.

LENGYEL, J., PRAUN, E., FINKELSTEIN, A., AND HOPPE, H. 2001. Real-time fur over arbitrary surfaces. In *Proc. of the 2001 symposium on Interactive 3D graphics*, 227–232.

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3d surface construction algorithm. In *Proc. of SIGGRAPH '87*, 163–169.

NEALEN, A., IGARASHI, T., SORKINE, O., AND ALEXA, M. 2007. Fibermesh: designing freeform surfaces with 3d curves. *ACM Trans. Graph. 26*, 3, 41.

OWADA, S., NIELSEN, F., OKABE, M., AND IGARASHI, T. 2004. Volumetric illustration: designing 3d models with internal textures. *ACM Trans. Graph. 23*, 3, 322–328.

PERLIN, K. 1985. An image synthesizer. In *Proc. of SIGGRAPH '85*, 287–296.

PIETRONI, N., OTADUY, M. A., BICKEL, B., GANOVELLI, F., AND GROSS, M. 2007. Texturing internal surfaces from a few cross sections. *Computer Graphics Forum 26*, 3, 637–644.

PRAUN, E., FINKELSTEIN, A., AND HOPPE, H. 2000. Lapped textures. In *Proc. of SIGGRAPH '00*, 465–470.

QIN, X., AND YANG, Y.-H. 2007. Aura 3d textures. *IEEE Transactions on Visualization and Computer Graphics 13*, 2, 379–389.

SI, H. 2006. On refinement of constrained delaunay tetrahedralizations. In *Proc. of the 15th International Meshing Roundtable*, 509–528.

SOLER, C., CANI, M.-P., AND ANGELIDIS, A. 2002. Hierarchical pattern mapping. *ACM Trans. Graph. 21*, 3, 673–680.

TREECE, G. M., PRAGER, R. W., AND GEE, A. H. 1999. Regularised marching tetrahedra: improved iso-surface extraction. *Computers and Graphics 23*, 4, 583–598.

TURK, G., AND O'BRIEN, J. F. 1999. Shape transformation using variational implicit functions. In *Proc. of SIGGRAPH '99*, 335–342.

WEI, L.-Y. 2002. *Texture synthesis by fixed neighborhood searching*. PhD thesis, Stanford University.