A survey on solid texture synthesis

Nico Pietroni, Paolo Cignoni, Miguel A. Otaduy, and Roberto Scopigno

Abstract—In this survey, we illustrate the different algorithms proposed in literature to synthesize and represent solid textures. Solid textures are an efficient instrument to compactly represent both the external and internal appearance of 3D objects, providing practical advantages with respect to classical 2D texturing. Recently, several methods have been proposed to synthesize solid textures. For some of those, which are commonly referred as procedural, colors are obtained by means of functions that algorithmically encode appearance and structure properties of the texture. Alternatively, example-based methods aim to capture and replicate the appearance as described by a set of input exemplars.
Within this framework, we propose a novel classification of solid texture synthesis methods: boundary-independent and boundary-dependent methods. In the case of boundary-independent methods, the shape of the object to be textured is irrelevant and texture information can be freely generated for each point in the space. Conversely, boundary-dependent methods conform the synthesis process to the actual shape of the object, so that they can exploit this information to orient and guide the texture generation. For better understanding the different algorithms proposed in the literature, we first provide a short introduction on 2D texture synthesis methods, focusing on the main principles which are also exploited for 3D texture synthesis. We review the different methodologies by considering their strengths and weaknesses, the class of appearances they can successfully synthesize, and failure cases. In particular, we focus our attention on advantages and drawbacks of boundary-independent methods with respect to boundary-dependent ones.

Index Terms—texture synthesis, solid texture

1 Modeling the Interior of an Object

It is common belief that textures provide a simple and efficient way of modeling 3D objects by separating appearance properties from their geometric description. Textures have been profusely used in computer graphics for modeling the external structure of objects, either through photographs or through procedural models [1]. While traditional 2D textures are usually used to encode information about the external surface of an object; extensions have been proposed for providing volumetric information, allowing the encoding of the internal appearance of objects, i.e., appearance properties are provided for each point belonging to a predefined volumetric domain $\mathcal{D} \subset \mathbb{R}^3$. This class of textures is usually referred in the literature as *solid textures*.

In surface texturing, one usually relies on a planar parameterization for associating texture attributes to a 3D object. A planar parameterization maps each 3D point belonging to an object's surface to a 2D domain, which encodes texture attributes. This $3D \rightarrow 2D$ mapping may introduce a distortion, which is generally dependent on the complexity of the object's topology and shape. Finding a good planar parameterization, i.e., a parameterization which minimizes the amount of introduced distortion, still remains a challenging task.

- *N. Pietroni, P. Cignoni and R.Scopigno are with the Visual Computing Lab at Instituto of Science e Tecnologie dell'Informazione (ISTI), National Research Council (CNR), Pisa, Italy E-mail: pietroni, cignoni, scopigno@isti.cnr.it*
- *Miguel A. Otaduy is assistant professor at the Modeling and Virtual Reality Group,Department of Computer Science URJC Madrid, Spain. E-mail: miguel.otaduy@urjc.es*

Several methods have been proposed in the literature to synthesize colors directly on the surface without the need of a planar parameterization (See [2], [3]). This task relies on two main steps: create an orientation field over the surface, and perform texture synthesis according to the orientation field. Thanks to the fact that the texture is orientated according to the underlying geometry, those methods may produce interesting results. Unfortunately, they lack reusability, i.e., since the color information is specifically defined for a given surface, then it cannot be reused to colorize a different one.

Solid textures provide two main advantages at the same time: first, we do not need any planar parameterization (since 3D coordinates of the surface constitute a valid parameterization); second, we can, in general, use the same solid texture to colorize different surfaces. Indeed, by simply carving out a surface from a solid texture, we define its color attributes. On the other hand, guaranteeing that a texture is free from visual artifacts is more complex in 3D than in 2D. In practice, a high quality solid texture must show a plausible appearance along any oriented slicing plane.
Solid textures exhibit advantages in several application domains. For example, they can be used to encode volumetric information needed to perform high-quality sub-surface scattering. In simulation of fracturing objects, solid textures can be used to synthesize the appearance of the internal surfaces revealed by fracture. Moreover, particular classes of materials, such as wood or rocks, can be more efficiently defined by using solid textures rather than 2D textures.

The major issue concerning solid textures is their accessibility. While external appearance of an object may be easily captured, for example, by taking photographs and "pasting" them onto a 3D model, producing a coherent solid texture representing its internal properties is a more complex task. Let us consider the example of capturing the internal appearance of a solid block made of marble. One possibility could be to repeatedly slice such block to get pictures of its internal sections, or, alternatively obtain a volumetric dataset by using a CT scan system. Since those strategies require some complex machinery, they are not useful in practice, and it appears more practical to synthesize internal colors from a reduced set of photos.

Algorithms for solid texture synthesis are mainly characterized by their *generality* and *controllability*.
The generality of a texture synthesis algorithm is its capacity of capturing and reproducing features which are present at different scales in the input image. Generality is one important characteristic of a synthesis algorithm, since it measures its versatility in modeling the different appearances present in the real world. The controllability is the level to which the user can guide the final result of the synthesis process.
According to the regularity of their appearance, textures can be classified on an interval which smoothly varies from *Structured regular* to *Stochastic* :

- **Structured regular textures**
  These textures present regular and structured patterns. An example of a structured regular texture is a brickwall.
- **Structured irregular textures**
  These textures present structured patterns which are not regular. An example of a structured irregular texture is a stonewall.
- **Stochastic textures**
  This class of textures look like noise showing a high degree of randomness. An example of a stochastic texture is roughcast or grass.

Such 2D texture classification is extensible to 3D textures, by considering the presence of regular patterns along the three directions, instead of two, of a solid texture. Since algorithms for modeling an objects's internal color are often defined by extending basic concepts of 2D texture synthesis, we briefly introduce some basic concepts regarding 2D texture synthesis algorithms in Section 2.

The internal appearance of an object $\mathcal{M}$ can be defined by a function $\mathcal{F}$ which maps each point p belonging to $\mathcal{M}$ to the respective color attribute $color(p) = \mathcal{F}(p), p \in \mathcal{M}$. As previously stated, this mapping is extrapolated by using a reduced input provided by the user. We divide the methods for modeling the internal appearance of an object into two main categories: *boundary-independent* and *boundary-dependent* solid texturing:

- **Boundary-dependent solid texturing methods**
  The texture conforms to the boundary of the object on which they are mapped.
- **Boundary-independent solid texturing methods**
  The texture does not rely on boundary information, and is computed on a boundary-free 3D domain.

Sections 3 and 4 make this classification more clear, providing an exhaustive description of existing approaches. These two classes of methods are finally compared in Section 5.

2 A Brief Introduction to 2D Texture Synthesis

The problem of texture synthesis is typically posed as producing a large (non-periodic) texture from a small input data provided by the user. In the next sections we provide a brief overview of existing 2D texture synthesis techniques, classifying existing methods as: procedural, statistical feature-matching, neighborhood matching, patch-based, and optimization-based. The last four categories (all except for procedural methods) can be grouped under the denomination of *example-based* synthesis methods, as they all use a small user-provided exemplar image to describe the characteristics of the output texture. The algorithm captures the appearance of the example texture so that it is possible, in a further step, to synthesize a new image (usually larger and non-periodic), which visually resembles such example texture. We refer to [4] for a more detailed description of existing methods for example-based 2D texture synthesis.

2.1 Procedural Methods

Procedural methods synthesize textures as a function of pixel coordinates and a set of tuning parameters. Among all procedural methods, the most used in Computer Graphics is *Perlin Noise*[1]. Perlin noise is a smooth gradient noise function that is invariant with respect to rotation and translation and is band-limited in frequency. This noise function is used to perturb mathematical functions in order to create pseudo-random patterns. Perlin noise has been widely used in various application domains, to cite a few: rendering of water waves, rendering of fire, or realistic synthesis of the appearance of marble or crystal.

Different classes of textures, such as organic texture patterns, may be efficiently synthesized by simulating natural process (usually modeled as small interacting geometric elements distributed on the domain).

2D procedural methods are, in general, efficient and easily extendable to solid texture synthesis.

2.2 Statistical Feature-Matching Methods

The main strategy of this class of methods consists in capturing a set of statistical features or abstract characteristics from an exemplar image and transfer them into a synthesized image.
Heeger et al. [5] uses an image pyramid to capture

statistical properties which are present in the exemplar image at different levels of resolution. The synthesized texture is initialized with random noise. Then, *Histogram matching* operations (see Section 3.3.1 for details) are repeatedly applied in order to make each level of the synthesized pyramid converge to the appearance specified by the exemplar image pyramid. This method and its extension [6] work well on stochastic textures, but their quality degrades in general if the example texture is structured.

2.3 Neighborhood Matching Methods

The main idea of neighborhood matching methods consists of enforcing and deploying the relation between pixel color and its spatial neighborhood. After an initial phase of training, where each pixel of the example texture is correlated to its neighborhood kernel, the target image is synthesized pixel by pixel. The synthesis step consists in substituting each pixel with the one that has the most similar neighborhood, chosen from the example texture [7]. Wei et al. [8] extends this algorithm in a multi-resolution fashion using *Gaussian pyramids*.

The neighborhood matching methods discussed above are inherently *order-dependent*, i.e., the resulting image depends on the order in which pixels are synthesized. Wei et al. [9] modified their original neighborhood matching algorithm to make it *order-independent*. The main idea can be summarized as follows: the value of a synthesized pixel is stored in a new image (instead of overwriting), while the kernel used for neighborhood search is made by pixels that were synthesized in the previous step.

In [10] the order-independent synthesis is performed on the GPU. In this case, synthesis can be performed in real-time, opening new application scenarios.

2.4 Patch-Based Methods

This class of methods relies on a different philosophy: the example texture is divided into a set of patches, which are re-arranged in the output image.

In [11], an overlap region is used between adjacent patches to appropriately *quilt* them making sure they all fit together. Patches that minimize an overlap error are chosen step-by-step randomly from a set of candidates, and iteratively placed over the synthesized image. Once patches are placed, the overlap region is quilted appropriately to minimize the error. Kwatra et al. [12] improve this approach by minimizing the error using a different strategy: a graph-cut algorithm.

2.5 Texture Optimization Method

This technique, introduced in [13], relies on a global optimization framework to synthesize a new texture. It consists essentially of minimizing an energy function that considers all the pixels together. This energy function measures the similarity with respect to the example texture and it is locally defined for each pixel. The local energy contributions coming from pixels are merged together in a global metric that is minimized.

This method produces very good results, furthermore the energy formulation can be easily extended to create flow-guided synthesis.

3 Boundary - Independent Solid Textures

Solid texturing copes with texture functions defined throughout a region of three-dimensional space instead of a 2D image space. A common approach, which we call boundary-independent, consists of synthesizing a volumetric color dataset (which commonly corresponds to a cube), from a reduced set of information without taking into account the target object's shape onto which the color is mapped. In this sense, this approach is similar to the classic 2D texture synthesis formulation problem (see Section 2): given a reduced set of information that encodes internal properties, produce a volumetric color dataset that visually resembles as much as possible the input data, along any arbitrary cross section.

A generic object can then be colorized by simply embedding it into the synthesized volumetric domain. Similarly to 2D texture synthesis, boundary-independent methods for the synthesis of solid textures can be classified into:

- **Procedural methods**
  The color is a function of the 3D position and a set of parameters provided by the user.
- **Statistical feature-matching methods**
  Statistics are extracted from 2D textures and replicated on the solid texture.
- **Neighborhood matching methods**
  The color of each voxel belonging to the solid texture depends on its neighbors.
- **Optimization-based methods**
  The solid texture is the result of a global minimization.

In the following sections, we present the most significant approaches according to this classification.

3.1 Notation

We introduce a simple notation used in the following sections. The reader may refer to Figure 1 for better understanding. We call *voxels* the cells belonging to a solid texture to distinguish them from the *pixels* that belong to a 2D texture. The *3D neighborhood* of a voxel v is formed by assembling 2D neighborhoods centered in v and slicing the solid texture along each axis. A *3D slice* refers to each orthogonal 2D neighborhood defining a *3D neighborhood*.

3.2 Procedural Methods

Procedural methods for the synthesis of solid textures are, in general, derived directly from the 2D methods.

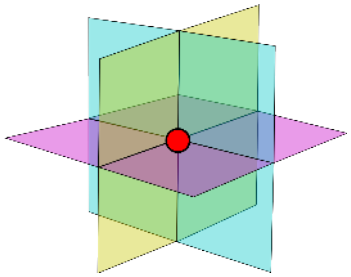
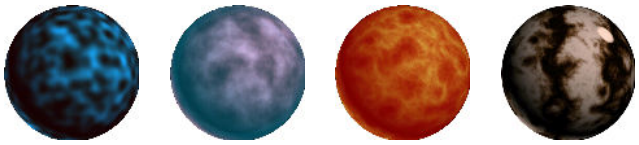


Fig. 1. A 3D neighborhood composed of three orthogonal 3D slices (See section 3.1).



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

Fig. 2. Examples of solid textures produced using Perlin noise [1] (see Section 3.2). Source www.noisemachine.com/talk1/ courtesy of Ken Perlin. (a)Simply the Perlin $noise()$ function. (b)A fractal sum of noise calls $\sum 1/f(noise)$ where $f(noise) = noise(p) + 1/2noise(2p) + 1/4noise(4p)$.. generates cloudy patterns. (c)Fire may be simply generated as $\sum 1/f(|noise|)$. (d)Marble patterns can be generated as $sin(x + \sum 1/f(|noise|))$ where x refers to the coordinate of the surface.

Indeed, thanks to their "dimension-independent" formulation, procedural methods are, in general, easily extendible to the 3D case.

For example, the noise functions defined by Perlin [1] can be used to synthesize solid textures. Solid noise is a 3D function used to perturb a basis 3D function in order to create realistic solid patterns. Perlin noise has been largely used in computer graphics to produce solid textures of marble, clouds or fire (see Figure 2 for application examples).

Procedural methods for solid texture synthesis are, in general, easy to implement and computationally light. Since the color of a voxel is a function of its coordinates, procedural methods can synthesize each voxel independently, while the majority of the other methods require the synthesis of the entire solid block.

Potentially, procedural methods are *general enough* to synthesize every possible pattern, furthermore the user may have a direct *control* of the result by tuning parameters. Unfortunately, from the point of view of the final user, it is difficult to express procedurally a desired texture appearance. Specifically, the user must elaborate an analytic description of the desired texture effect. Although theoretically any pattern may have its analytic description, in general it does not correspond to an intuitive formulation.

Recently, Lagae et al. [14] have proposed a novel noise formulation which provides more intuitive parameters.

3.3 Statistical Feature-Matching Methods

Similarly to the 2D case, the main purpose of this method is to extract a set of statistical properties from an exemplar image in order to replicate it in the synthesized texture. However, solid texture synthesis is a more complex scenario: properties are defined in a 2D image, while the synthesis is performed in 3D. Since no 3D information is provided, these methods transfer statistical properties defined over a 2D space to a higher order space, i.e., the 3D space embedding the solid texture.

3.3.1 Histograms matching

In [5] (which has been introduced in section 2.2), authors generalize the proposed method to the synthesis of solid textures. Since the CDF (Cumulative Distribution Function) expressed by the image histogram is independent with respect to dimensionality of the input data, it is possible to apply the same histogram matching on a solid texture, rather than an image. In this specific case, input histograms rely on 2D steerable pyramids of the exemplar image, while output histograms rely on 3D steerable pyramids of the solid texture.

More precisely, color range belonging to 2D exemplars and 3D solid texture are quantized separately into a set of uniform interval bins, each of which represents the probability of a pixel to fall into such interval (evaluated using color distribution).

Histogram matching is used to match the color distribution of the synthesized texture with the color distribution of the exemplar. Histogram matching is based on the Cumulative Distribution Function $CDF_H : [bins] \rightarrow [0, 1]$ and its inverse $CDF_H^{-1} : [0, 1] \rightarrow [bins]$, were H is an image histogram. Given an output image I (which in this specific case is the synthesized solid texture) and an input image I', and considering their histograms H_I and $H_{I'}$, histogram matching consists of substituting each color of the output image $v \in I$ with the one having the same CDF value in the input image I':

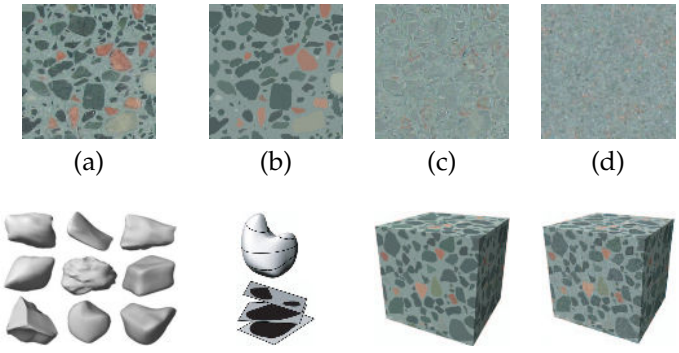
$$v' = CDF_{H'}^{-1}(CDF_H(v)) \quad (1)$$

The overall algorithm proposed by [5] can be finally outlined as follows: the output solid texture is initialized with random noise; then histogram matching between noise and example textures is performed; the algorithm continues by iteratively applying histogram matching across each pair of steerable pyramid levels; and, finally, the image is fully reconstructed from the processed pyramid levels.

Notice that this method treats each color channel independently, i.e., histograms are calculated and matched independently for each channel. Finally, the synthesized texture is obtained by reassembling the processed color channels. In general, since statical methods de-correlate color channels, they may produce visual artifacts.

(a) (b)

Fig. 3. (a) Examples of solid textures produced by [5] (see Section 3.3.1). The textured model is carved from the synthesized texture block. (b) Anisotropic solid textures generated by [15] using multiple example textures.



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

(e) (f) (g) (h)

Fig. 4. The synthesis pipeline of [19] (see Section 3.3.3 for details) : The initial image (a) is filtered to extract two components: a profile image (b) and a residual image (c). The profile image, together with the shape of particles (f) is used to infer, through stereology, the 3D distribution of particles (e) (encoded as triangle meshes), while the residual image is used to synthesize a residual solid texture (d). The final solid texture (h) is obtained by adding the residual solid texture, which encodes the fine details, to the solid texture obtained from the distributed particles (g), which encode the rough structure.

3.3.2 Spectral analysis

Ghazanfarpour et al. [16] propose to use *spectral analysis* for solid texture synthesis. Spectral information is extracted from the example texture using the *Fast Fourier Transform* (FFT), and used to obtain a basis and a noise function. Finally, the solid texture is obtained procedurally as in [1]. This method is extended by [17] to use multiple images. Each image defines the appearance of the solid texture along an imaginary axis-aligned slice. The algorithm is built upon the assumption that the appearance of axis-aligned cross-sections are invariant with respect to translation, while the non-orthogonal ones blend the appearance of the three example textures according to their orientation. Modifications are obtained by using *spectral* and *phase* processing of image FFT. The synthesis process takes as input a solid block initialized with noise, and modifies axis-aligned slices, extracted from the solid texture, according to the corresponding example texture. Since each voxel belongs to three different slices, it defines three possible colors, which are simply averaged. By repeating this step, the noise block slowly converges to the appearance of example textures. In [15] this approach was modified to avoid phase processing. The solid texture is generated by repeatedly applying spectral and histogram matching. While methods based on spectral analysis([16],[17] and [15]) produce pleasant results when applied to stochastic textures, they usually perform worse with structured textures. The reader may refer to [18] for a survey on spectral analysis methods.

3.3.3 Stereology

A significantly different approach in generating structured textures is proposed by Jagnow et al. [19]. Their method is limited to the synthesis of a particular class of materials that can be described as "particles embedded in a homogenous material". It is based on *classical stereology*, an interdisciplinary field that provides techniques to extract three-dimensional information from measurements made on two-dimensional planar sections. Figure 4 gives an overview of the method.

In this method, stereology relates the particle area distribution in the profile image with particle area distribution revealed by an arbitrary cross-section of the solid texture. Profile image and particle shape concur to define the 3D particle distribution, since:

- The profile image captures the distribution of particle area. This distribution must be replicated in the solid texture, so that it is preserved along every cross-section.
- On the other hand, a cross-section of the solid texture cuts some particles defining an area distribution that is obviously related to particle shape. Authors propose to capture the area distribution generated by a particle by cutting randomly its meshed model (Figure 4.f).

These probability distributions concur to extract a *particle density function* which defines implicitly how particles have to be distributed.

In [20], Jagnow et al. present an interesting analysis about how different methods for approximating particle shape influence the perception of the generated solid texture. This stereology-based synthesis technique produces very realistic results, however it can be applied only to the specific class of textures that can be described as particles distributed on a homogenous material.

3.3.4 Aura 3D textures

Aura 3D textures [21] is the most general among the statistical based methods. Aura 3D solid texture synthesis is based on Basic Gray Level Aura Matrices (BGLAM)[22].

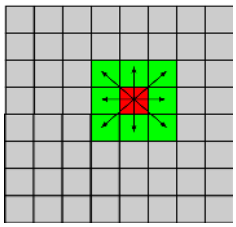


Fig. 5. Displacement configurations of [21] generated by a 3^2 kernel.

The information stored in BGLAMs characterizes the co-occurrence probability of each grey level at all possible neighbor positions, which are called also *displacement configurations* (see Figure 5). The synthesis algorithm is based on the consideration that two textures look similar if their *Aura matrix distance* is within a certain threshold. Aura matrix distance between two images is defined considering their BGLAMs. This approach, similarly to [17] and [15], produces a solid texture given a set of oriented example textures. Usually two or three axis-aligned example textures are enough to define the anisotropic nature of a solid texture, nevertheless this method supports an arbitrary number of input textures.

As previously introduced, the structure of a texture is captured by the BGLAM. More precisely, given a grey level image I quantized into G grey levels, and considering the $n \times n$ squared neighborhood of a pixel t, there are $(n^2 - 1) = m$ possible BGLAMs, one for each possible displacement configuration with respect to t (see Figure 5). The *BGLAM distance* $A_i \in R^{GXG}$ for a given displacement configuration $i : 0 \leq i < m$ is computed as follows:

- Initialize A_i with zero.
- For each pixel s belonging to I, consider its neighbor k defined by the current displacement configuration i.
- Increment $A_i[g_s][g_k]$ by 1. Where g_s and g_k are respectively the grey levels of s and k.
- Normalize A_i, such that $\sum_{i,j=0}^{G-1} A[i][j] = 1$.

Then, the distance $D(A, B)$ between two BGLAMs is defined as follows:

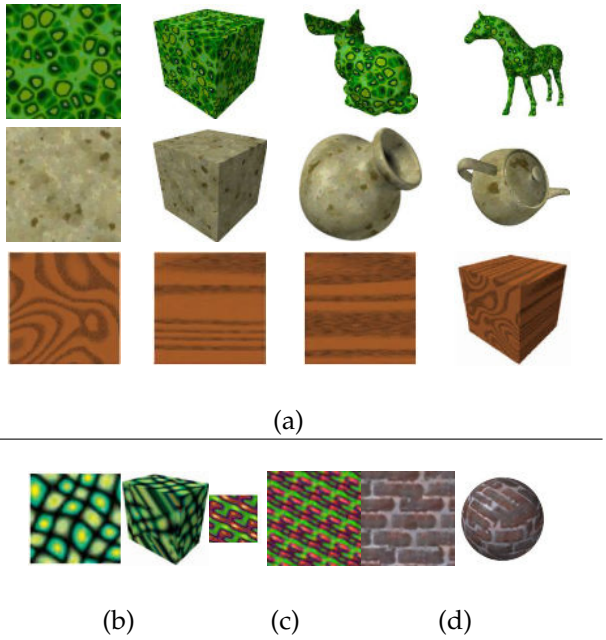
$$D(A, B) = \frac{1}{m} \sum_{i=0}^{i<m} \|A_i - B_i\|, \qquad (2)$$

where $\|A\| = \sum_{i,j=0}^{G-1} A[i][j]$.

This formula relates only two 2D textures. In the case of solid texture synthesis, it has to be extended in order to consider the distance of a voxel (with its volumetric neighborhood) from a set of oriented slices. Such extension is the Aura matrix distance. Aura matrix distance is defined by blending appropriately the BGLAM distances between 3D slices and example textures. This method can be generalized to support an arbitrary number of example textures. As usual, the solid texture is initialized with random noise, then the synthesis process consists in minimizing the Aura matrix distance of each voxel with respect to example textures. In detail, the algorithm repeats the following steps:

- Choose randomly a voxel v.
- Among all possible grey levels $0 \ldots G-1$, select the subset of candidates C_G that reduces the current Aura matrix distance from example textures.
- Substitute the grey value of v, by choosing randomly from C_G.

Since BGLAM works only with grey levels, as in [5], color channels must be decorrelated in a way such that the algorithm can work independently on each channel. The algorithm produces good results, especially for structured textures (see Figure 6.a), we may assert that this methods is the most general among the statistical-based. On the other hand Aura 3D synthesis is not interactive, and, since it decorrelates color channels, it may lead to visual artifacts (as in in [5]). It also may produces inconsistencies in the case oriented structural textures were used as exemplars.



(a)

(b) (c) (d)

Fig. 6. (a) Successful examples of textures synthesized by Aura 3D synthesis [21] (see Section 3.3.4). (b) Effect of convergence to a local minimum. (c) Independent synthesis of decorrelated channels leads to visual artifacts (courtesy of [23]). (d) An inconsistency generated by an oriented structural texture.

3.4 Neighborhood Matching Methods

Pixel-based methods for 2D texture synthesis (previously discussed in 2.3) have also been extended in order to synthesize solid textures. Similarly to the 2D neighborhood matching synthesis, the main intuition consists of characterizing a voxel by using only its neighbors. Again, the solid texture is produced by modifying a

Fig. 7. Examples of solid textures produced by [24] (see Section 3.4).

single voxel at a time, searching in the example texture for the candidate which has a similar neighborhood. While the underlying principles are the same, volumetric synthesis entails novel problems:

- How to compare the 3D neighborhood of a voxel with 2D pixel neighborhoods coming from example textures.
- How to handle multiple oriented example textures that concur to define a single voxel color.

Wei extended [8] to synthesize textures from multiple sources [24]. This method, originally proposed to synthesize 2D textures by mixing multiple sources, is modified to create solid textures from a set of oriented slices. As in [17], [15], [21], the user defines the appearance of the solid texture along its principal directions by providing a set of axis-aligned slices T_x, T_y, T_z. For each voxel v, the best-matching pixel from the example textures is selected by using 3D slices. As in [8], three candidate colors p_x, p_y, p_z are selected by minimizing the energy function E, defined as the squared differences between 3D slices and 2D neighborhoods:

$$E_x(v, p_x) = \|v - p_x\|^2 + \|I_x - N(p_x)\|^2; \quad (3)$$

$$E_y(v, p_y) = \|v - p_y\|^2 + \|I_y - N(p_y)\|^2; \quad (4)$$

$$E_z(v, p_z) = \|v - p_z\|^2 + \|I_z - N(p_z)\|^2; \quad (5)$$

where p_x, p_y, p_z are pixels chosen form the respective example textures T_x, T_y, T_z, and $N(p_i)$ represents the 2D neighborhood of a pixel p_i. A voxel's color is finally assigned by averaging the candidate colors p_x, p_y, p_z. The synthesis process starts with a block of noise and runs over voxels changing the colors. As in [8], the entire process is performed in a multi-resolution fashion by using Gaussian pyramids. This method is simple to implement but, as shown in Figure 7, the resulting textures may exhibit some blurring and have difficulty to preserve patterns that are present in the example textures.

3.5 Optimization-Based Methods

The 2D optimization-based texture synthesis method [13] (see Section 2.5 for details) has been extended by Kopf et al. [23] to synthesize solid textures. As in [13], the main goal of this method is to make the solid texture look like the 2D example texture by globally minimizing an energy function.
For the case of solid texture synthesis, the global energy equation E_T is reformulated in order to consider a 3D neighborhood:

$$E_T(v; \{e\}) = \sum_v \sum_{i \in \{x,y,z\}} \|S_{v,i} - E_{v,i}\|^r. \quad (6)$$

Where the voxel v iterates across the whole solid texture, $S_{v,i}$ are 3D slices at voxel v, and $E_{v,i}$ are is the 2D neighborhood of the candidate for the voxel v, coming from the exemplar image i. Minimization is performed by using again the same Expectation-Maximization process of the 2D case, which consists of two main phases:

- **Optimization phase**
  Keeping E_v fixed, minimize E_T by modifying S^v. In other words, the color of a voxel S^v is modified to resemble locally, as much as possible, to the precomputed candidate.
  By setting the derivative of E_T with respect to S^v to zero, it turns out that the optimal value for a voxel is expressed by the following weighted sum:
  $$S_v = \frac{\sum_{i \in \{x,y,z\}} \sum_{u \in N_i(v)} w_{u,i,v} E_{u,i,v}}{\sum_{i \in \{x,y,z\}} \sum_{u \in N_i(v)} w_{u,i,v}}, \quad (7)$$
  where $N_i(v)$ are the different slices forming the 3D neighborhood of the voxel v.
- **Search Phase**
  Keeping S_v fixed, minimize E_T by updating E_v. For each synthesized pixel v, the corresponding candidate E_v is updated by using best-matching neighborhood search in the exemplar image.

Since this minimization process takes in account only local information, it may converge to a local minimum. To take into account global statistics, [23] proposes to modify weights of equation 7 using the histograms of the synthesized texture and the exemplar images. More precisely, they reduce the weights that increase the difference between the current histogram and the histograms of the example textures.

Starting from a solid block initialized by choosing colors randomly from the example textures, the synthesis is performed in a multi-resolution fashion. To enforce preservation of strong features, it is possible to include a feature map in the synthesis process.

The ability of this method to preserve sharp features is superior if compared with earlier works (see Figure 8.(c)). Moreover, using a user-defined constraint map, it is possible to tune the minimization to create predefined patterns (see Figure 8.(b)).
Since the optimization is performed globally, this method requires to synthesize the entire volumetric data. Furthermore, the time needed for the minimization process is very long (from 10 to 90 minutes to synthesize a 128^3 block).

3.6 Order-independent / Parallel Methods

Dong et al.[25] proposed a method to synthesize solid textures called "lazy solid texture synthesis". The main

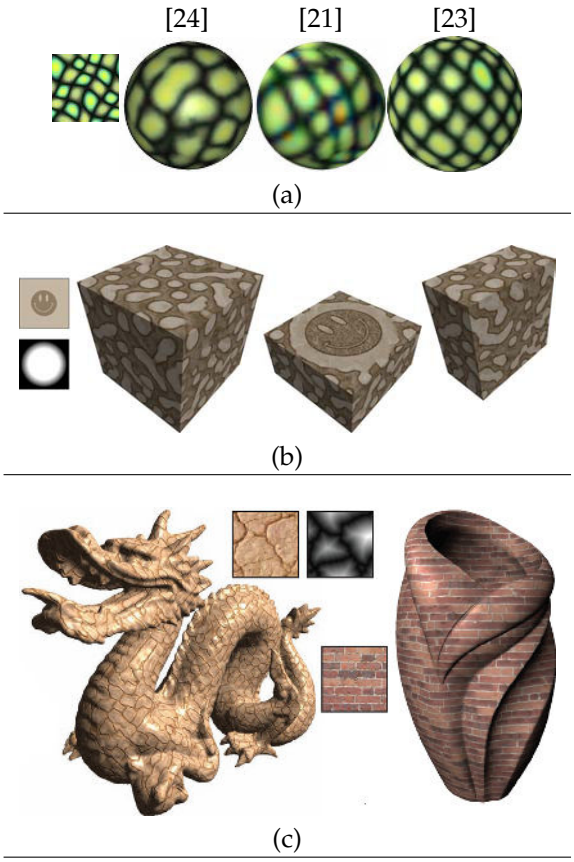


Fig. 8. (a) Comparison of different methods in solid texture synthesis from 2D exemplars. [23] preserves sharp features, while [24] and [21] introduce blurring. (b) An example of constrained synthesis. (c) Examples of surfaces carved from a texture block synthesized using [23] (see Section 3.5).

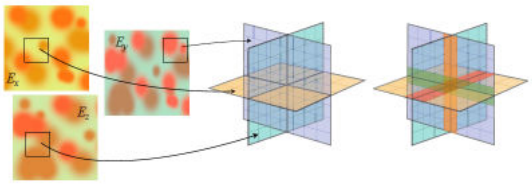


Fig. 9. The candidate of [25]. Left: Three exemplars composing a candidate. Right: The overlap region defined by a candidate (see Section 3.6).

advantage provided by this method is the possibility to synthesize textures in parallel, which makes it particularly suitable for interactive simulations such as real-time fracturing or cutting. More precisely, two main characteristics make this method suitable for real time applications:

- **Parallelism**
  The algorithm can be parallelized. The authors propose a GPU parallel implementation that provides real-time synthesis.
- **Granularity of the synthesis**
  Thanks to its locality, this algorithm can synthesize a small subset of voxels near to a visible surface instead of synthesizing the whole volume.

Similarly to neighborhood matching methods [24], the algorithm proceeds by substituting each voxel of the output solid texture with a candidate chosen from example textures, which has a similar neighborhood. A candidate is a 3D neighborhood created by selecting slices from the exemplar images . The cardinality of possible candidates is huge if we consider that we can create candidates by combining triples of 2D neighborhoods selected from example textures. To speedup the computation, [25] extends the *k-coherence* algorithm [26] to the 3D case.
In a preprocessing step, for each pixel of the exemplar images, they assemble a *candidate set*. This set is initially composed by using the pixel itself and two pixels coming from the other exemplar, along with their respective 2D neighborhoods. Then, each candidate set can be reduced by pruning candidates that produce color incoherences. More precisely, each candidate can be classified according to two metrics:

- **Color Consistency**
  Given that each candidate defines an overlap region (see Figure 9), color consistency is measured as the coherence of a candidate along its overlap region. Based on similarity of colors, it is evaluated by summing squared color differences in the overlap region.
- **Color Coherence**
  It is the ability of the candidate to form coherent patches from example textures. It is evaluated by considering the amount of neighboring pixels that form contiguous patches.

During the synthesis process, the algorithm maintains for each voxel a triple of 2D texture coordinates referring to exemplar images. The color of a voxel is defined by the average of the three colors referred by such texture coordinates.
The synthesis is performed in a multi-resolution fashion, from coarse to fine level, by using Gaussian pyramids. Starting from an initial block, which is formed by tiling the best candidate for each pixel, the synthesis pipeline, as in [10], is divided into three main steps:

- **Upsampling**
  This step is used when the algorithm switches to a finer resolution level. Upsampling is simply performed by texture coordinate inheritance.
- **Jittering**
  It introduces variance in the output data. It is performed by deforming colors in the solid texture.
- **Correction**
  It makes the jittered data look like example textures. For each voxel, a 3D neighborhood is extracted from the solid texture. Then, according to *k-coherence*, a set of candidates is defined as the union of the different candidate sets referred by texture coordinates. The

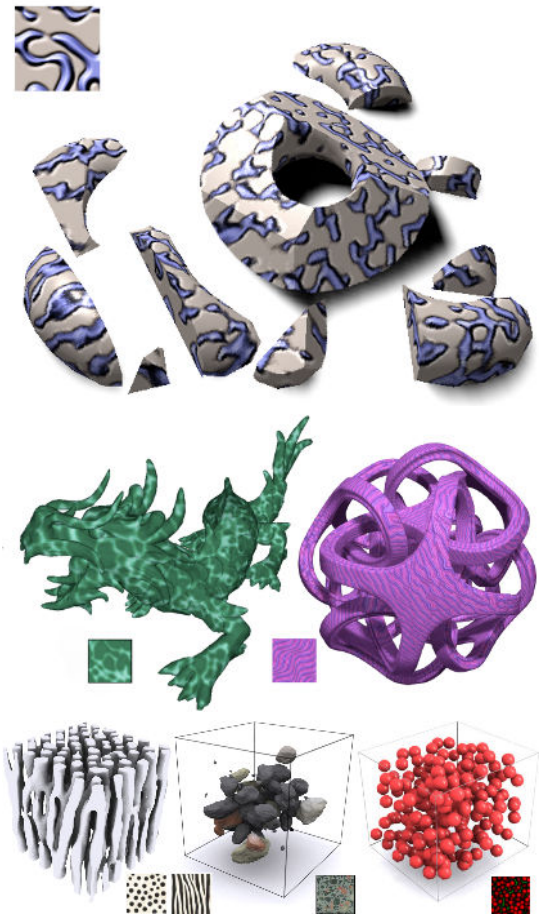


Fig. 10. Some examples of solid textures synthesized by lazy solid texture synthesis [25] using single or multiple exemplars. (see Section 3.6).

search for the best match is limited within such space. Similarity, as usual, is measured as squared difference of color values. Once the best match is found, texture coordinates are substituted.

To enforce parallelism, the whole synthesis process must be independent with respect to the order in which voxels are processed. To achieve order-independency, as in [9], [10], synthesis is performed considering the data that has been evaluated on the previous step.

As previously stated, thanks to the locality of the data involved in the process, it is possible to synthesize on demand a block of voxels instead of synthesizing the complete volumetric dataset. The granularity of the synthesis is limited by neighborhood size. It follows that, when one wants to texturize a triangle mesh, the synthesis can be limited to a solid shell following the surface. As shown in Figure 10, this method produces nice results for a wide variety of input textures.

4 Boundary-Dependent Solid Textures

Boundary-independent methods do not adapt the color information to the object's boundary. Alternatively, the object's volume can be used as the domain on which

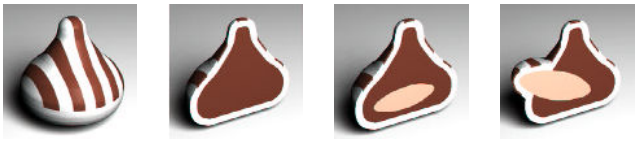


Fig. 11. Examples of layered textures created by [27] (see Section 4).

the synthesis is performed, and the interior texture can conform to the known texture on the boundary or the known shape of the boundary: we call this class of approaches *boundary-dependent*. The main challenge of boundary-dependent methods consists of creating an appropriate representation of the object's volume and use it as the synthesis domain.

Since in boundary-dependent methods the synthesis process is constrained by the boundary surface, it is possible to obtain interesting effects, such as orienting the textures to follows the surface's shape, or to define a layered texture. Boundary-dependent methods are, in general, semi-automatic: the user specifies some appearance property of the object and the system infers how to synthesize the interior. Furthermore, the existing methods do not need to store explicitly the color of each voxel, as it is implicitly defined by the domain model.

We consider [27] as the first example of boundary-dependent synthesis method. In this method, the interior of on object is defined by using a simple scripting language that allows the definition of nested textures. This effect is realized by using a signed distance field. Despite the interesting results shown in the paper, textures are generated procedurally, so the set of possible appearances is limited.

4.1 Volumetric Illustrations

Owada et al. proposed a novel boundary-dependent method to model the internal appearance of an object [28]. The user specifies the interior of an object by using a *browsing interface* and a *modeling interface*. The browsing interface is a model viewer that allows the user to visualize the internal structure of an object (See Figure 12.a). The user freely sketches 2D path lines on the screen to specify the direction along which the object should split. These paths are projected onto the 3D mesh to define cross-sections. Once the surface is split into two parts, its internal surface is re-triangulated, according to the split section, such that internal appearance can be finally rendered.

The modeling interface provides an intuitive way to specify the internal structure of the model. When an object reveals its internal surface it is possible to specify a texture for each closed volumetric region. That allows, for example, to define multiple appearances in the case that the domain contains multiple closed regions (see Figure 12.e). That information is used, together with the triangle mesh, to perform synthesis on cross-sections.

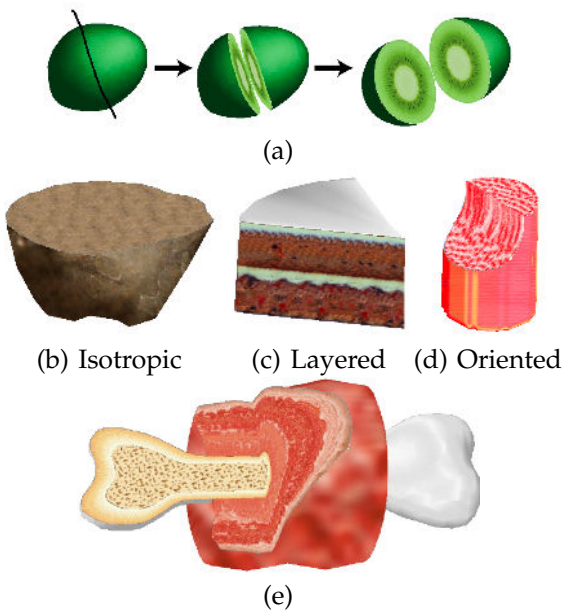


Fig. 12. (a) The browsing interface of [28] (see Section 4.1) (b) Isotropic texture. (c) Layered texture. (d) Oriented texture. (e) Example of subdivided domain (bone and meat) modeled by [28].

More precisely, once each region of the mesh is linked with the respective example texture, then cross sections can be synthesized on-the-fly using a 2D synthesis algorithm. That operation requires the parametrization of cross-sections, since no volumetric textures are created.

The system allows the use of three different kinds of textures:

- **Isotropic textures**
  Such textures do not depend on the surface; they can simply be synthesized in the parametric space of a cross-section using a standard 2D synthesis algorithm such as [8] (see Figure 12.b).
- **Layered textures**
  Their appearance changes according to depth. A smooth 2D distance field is calculated in the cross-section. Then, synthesis is performed using 2D texture synthesis algorithm, but with some texture variation according to a distance field (see Figure 12.c).
- **Oriented textures**
  Such textures have distinct appearance in cross-sections that are perpendicular and parallel to a flow orientation (an example of oriented texture is shown in Figure 12.d). The user defines by sketching a main flow direction, and the system uses this vector to orient a 3D flow field defined inside the volume. A reference volume is synthesized simply by sweeping the texture image along the y direction. This reference volume is used, together with the 3D flow field, to texturize properly the cross-section. A 2D neighborhood matching texture synthesis technique [8] is used to synthesize colors of the parameterized cross-section. To make the synthesis process dependent on surface orientation, the neighborhood search step should be modified as follows: Given a pixel p, with normal n, the set of candidates used in coherent search is formed by slicing patches of the reference volume along planes orthogonal to n.

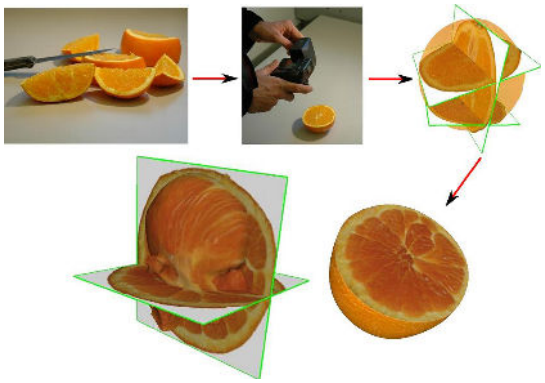


Fig. 13. The synthesis pipeline of [29] (see 4.2) : using an interactive editor, photos of internal surfaces of real objects are placed in the local reference frame of a 3D model. Then, after a preprocessing step, it is possible to color internal surfaces with highly realistic texture, or to carve 3D models out of organic objects in real-time.

As shown by Figure 12, the user can easily produce nice results with some mouse clicks. Thanks to the user-friendly interface, this method is an interesting solution for producing scientific illustrations (useful in medicine, biology or geology). However, the expressive power of the method is limited.

4.2 Texturing Internal Surfaces from a Few Cross Sections

Pietroni et al. proposed to capture the internal structure of an object by using a few photographs of cross-sections of a real object [29]. In a preprocessing step, the user places the cross-section images in the local reference frame of the 3D model, and the images are initially smoothly deformed to fit with the models boundary. Then the synthesis is performed in real-time by *morphing* between the different cross sections.

Splitting iteratively a 3D object with planar cuts produces a *BSP tree*, which constitutes the interpolation domain on which colors are synthesized. Since cross sections can intersect, then they may be subdivided in several *planar sub-domains*, which are, individually, topologically equivalent to a disk. Therefore, the whole object's volume is split among the different regions defined by the BSP tree. Each region is bounded by a set of planar sub-domains and, possibly, a portion of the external surface. The color of a point is a function of the different planar sub-domains defining the BSP region on which such point falls into (See Figure 13). More precisely, each voxel that has to be synthesized is projected onto each sub-domain bounding its BSP

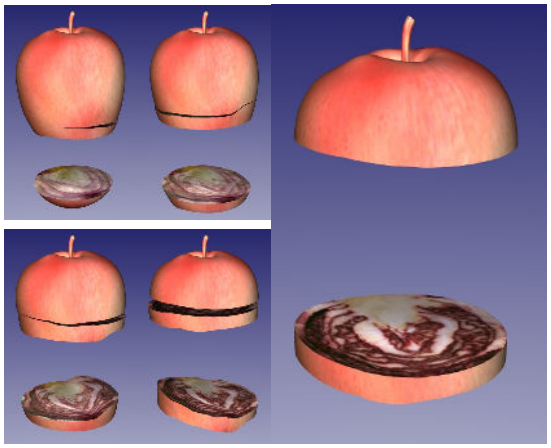


Fig. 14. Example of real-time synthesis for cutting simulation using [29] (see 4.2).

region, identifying a set of *source texels*. Such projection is defined by a path-line field emanating from each subdomain and smoothly covering the entire BSP region, possibly following the external boundary's shape.

Pairwise bijective mapping functions, called *warpings*, relate the exemplars that bound common BSP regions. More precisely, warpings minimize the feature misalignment between pairs of images. Pairwise warpings are computed in a preprocessing step by using an extension of the algorithm defined by Matusik et al [30]. Warping is used to morph between the different source texels such that sharpness is preserved. The morphing formulation proposed by [30] is approximated in order to synthesize colors in real time.

The color $c(v)$ of a voxel v is defined as:

$$c(v) = \sum_i w_i c_i \left(p_i + \left(\sum_{j \neq i} w_j W_{ij}^{-1}(p_i) \right) \right), \qquad (8)$$

where p_i are the different source texels whose colors are identified by the function c_i; W_{ij}^{-1} is the inverse of the warping function; and weights w_i are calculated by using *Shepard* interpolation $w_i = \frac{1}{\|v - p_i\|}$ and normalized such that $\sum_i w_i = 1$.

Finally, in order to reintroduce high frequencies present in the original image, a histogram matching approach is adopted, based on local neighborhoods. This histogram matching can be efficiently performed in real time.

Once the warping is precomputed, we can summarize the real-time synthesis pipeline for a voxel v as follows:

- First, identify the region of the BSP tree in which v falls into;
- Then v is projected onto the different planar subdomains in order to identify source texels p_i;
- The color of v is determined by equation 8;
- A local histogram matching is finally used to enhance features.

As shown by Figure 14, this algorithm captures global and medium-scale features, reintroducing small features through local histogram matching. The synthesis can be performed in real time since the algorithm can synthesize a 141×141 image/sec (on a 1.7 GHz Intel Centrino processor and 1 GB of RAM, [29]). Since the method is based on morphing, it works well with highly structured textures, while it cannot synthesize a stochastic 3D distribution of features, as in [19]. Furthermore it requires the base domain mesh should be closed (at least in correspondence with cross sections) in order to fits the example textures correctly within the geometry.

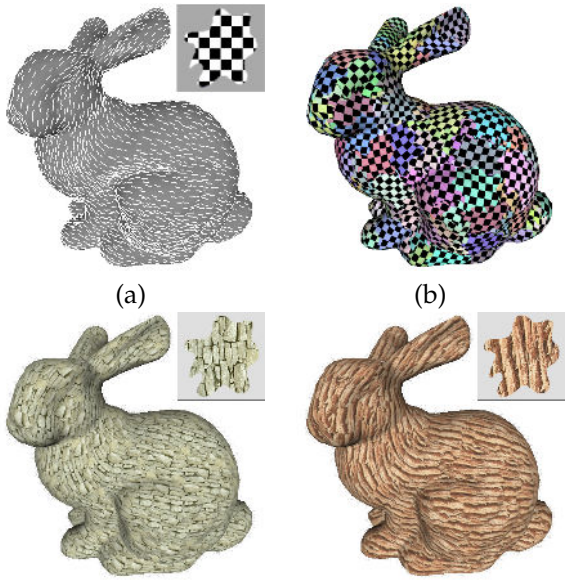


Fig. 15. 2D Lapped textures [31] (see Section 4.3): (a) The continuous tangent field and the 2D patch. (b) The local surface parameterization. (c)&(d) Some results.

4.3 Lapped Solid Textures

Lapped textures [31] is a technique to synthesize textures on surfaces. It consists mainly of overlapping properly a set of irregular patches to cover the entire surface. Figure 15 illustrates how lapped textures work: by using a continuous tangent field and a 2D patch (Figure 15.a), the surface is locally parameterized (Figure 15.b), so that it is possible to texturize it by repeatedly pasting patches (Figure 15.c and Figure 15.d). The method does not require storing explicitly the color, since it is implicitly defined by texture coordinates.

Takayama et al.[32] propose to extended lapped textures to fill volumes instead of surfaces. The basic concepts behind the 2D and the 3D versions are similar. As already mentioned, 2D lapped textures paste irregular patches over triangles, and similarly 3D lapped textures paste and blend solid texture patches over tetrahedra. Moreover, 2D lapped textures use a tangent field on the surface to orient textures, and similarly 3D lapped textures use a smooth tensor field (three orthogonal vector fields) along the volume to arrange solid patches.

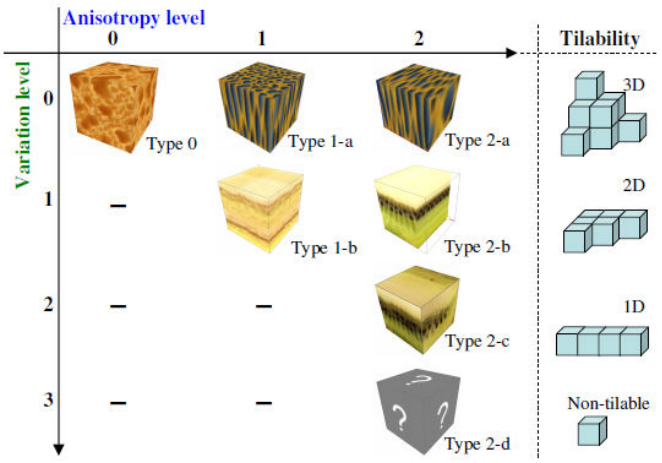


Fig. 16. Classification of solid texture appearance according to [32] (see Section 4.3).

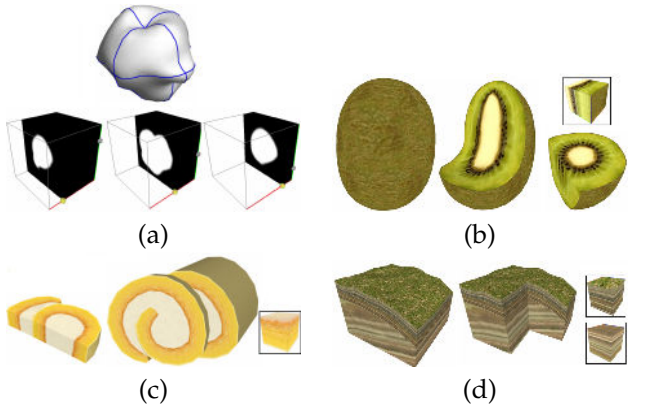


Fig. 17. Lapped solid textures [32] (see Section 4.3): (a) The alpha mask with "splotch" shape used to modify the shape of volumetric samples used for the synthesis process. (b) & (c) & (d) Examples of results.

Furthermore, like in 2D lapped textures, 3D lapped textures require storing only the 3D texture coordinates of each vertex belonging to the tetrahedral mesh. Finally, in order to avoid artifacts in the final texturing, [31] propose to use a "splotch" alpha mask (shown in Figure 15), while the 3D lapped-textures algorithm uses a volumetric alpha mask to produce a "splotch" shape.

[32] classify solid textures by considering both their anisotropy and variation (See Figure 16). *Anisotropy level* describes how the appearance of a cross section varies with respect to the orientation of the slice, while *variation level* expresses the number of directions along which the texture changes. The tileability of the texture depends on the variation level, i.e., a solid block is tileable along the directions that preserve the appearance.

The user first selects the appearance class (according to a table) he or she wants to model. Then, if required by the texture class, the user specifies directions by sketching strokes (the interface is similar to the one in [28]). In case the solid texture is anisotropic, the system creates a consistent global tensor field to force the texture to follow the orientations. The tensor field is calculated by Laplacian smoothing of user-defined directions along the tetrahedral mesh. The algorithm can be summarized as follows: Initially, a patch is pasted by the user in the object's volume; then, tetrahedra that are inside the alpha mask are marked as "covered"; then the "covered" region is expanded until it includes the entire tetrahedral mesh, by repeatedly pasting patches. Each pasting operation implies that covered tetrahedra must be transformed in texture space according to the tensor field, such that it is possible to assign per-vertex 3D texture coordinates.

This method can model a wide variety of textures (as shown by Figure 17), and requires low memory consumption. However, it has one strict requirement: the initial set of solid textures has to be provided a priori, together with alpha masks.

5 Discussion

There are similarities and differences between boundary-dependent and boundary-independent methods.

- **Reusability**
  Solid textures produced by boundary-independent algorithms can be reused to color every possible surface. Indeed, by simply embedding any object into the solid texture domain (which is typically a cube) one may derive the color information.
  Boundary-dependent methods are designed to be strictly coupled with the object on which they are defined. However, it is possible to make them reusable. For example, for sake of clarity, if we use a cube as external boundary we can generate a homogeneous texture volume, then, in a further step, by simply embedding a generic object into such cube, we can define its internal appearance with an approach similar to boundary-independent methods. Unfortunately, such strategy does not exploit the main advantages provided by boundary-dependent methods.
  Furthermore, it is important to notice that to build the spatial structures needed to use boundary-dependent methods, the boundary geometry should be well conditioned. For example, to apply [28], the geometry has to be subdivided, in order to synthesize from multiple exemplars, and closed, to parameterize the internal surfaces on which 2D synthesis is performed. In the case of [29], the geometry has to be closed, at least in correspondence with cross sections, while to apply [32] it must be tetrahedralized.
- **User interaction and controllability**
  Boundary-independent solid texture creation is, usually, completely automatic. With example-based methods, once the user has provided the example texture, the system automatically constructs the solid texture. In the case of procedural methods the

user may directly control the final result by tuning parameters.
Boundary-dependent methods often require some user-interaction. For example, [28] and [32] may ask the user to provide a direction to orient textures inside the volume, while [29] requires the user to place the cross sections onto the 3D model. That is not necessarily a drawback since they empower the user with an interface to design the final appearance of the solid texture.

- **Boundary constraints**
  Boundary-dependent methods conform to boundary constraints. This means it is possible to obtain a large variety of volumetric effects including layered textures or textures that follow the shape of the object. That is, precisely, one of the main aspects that has motivated researchers to develop such methods. In particular, [28] and [29] focus on interaction and provide user control.
  The distinction between boundary-independent and boundary-dependent methods can become fuzzy if boundary-independent methods are extended to use some sort of guidance that considers the object's boundary. For example, in [23], the synthesis can be constrained by a 2D mask which approximates the external boundary of an object. Also, [24] and [25] may be extended to synthesize from multiple sources according to a given mask. That strategy allows boundary-independent methods to create layered textures. Furthermore, given a 3D tangent field it may be possible to orient the synthesis. In particular, if such tangent directions follow an object's shape, consequently the synthesis will follow such shape.
  In general boundary-independent methods do not have any intrinsic limit to conforms to boundary constraints. On the other hand, most of the boundary-dependent methods have been designed with the modeler in mind, providing the user an intuitive interface to control the final appearance based on the object's shape.
- **Distortion**
  Boundary-independent methods introduce no distortion. Since the solid texture is explicitly maintained as a volumetric grid of values, then one may obtain the color of each point belonging to such domain through trilinear interpolation, introducing no distortion. On the other hand, boundary-dependent methods use auxiliary spatial data structures as metaphors to represent the domain volume and to retrieve color information. For example, [28] relies on a planar parameterization of the surface that has to be textured, [29] needs a projection step onto the cross sections, while [32] uses 3D texture coordinates to repeatedly paste an example solid texture over a tetrahedral mesh. All the operations cited above may introduce, in different ways, a certain quantity of distortion.

We have seen that the two classes of methods proposed so far have pros and cons, and there is no approach valid for all goals. We think that an interesting subject for future research could be to find an approach for synthesizing and representing solid textures that might combine the benefits of both classes of methods.

ACKNOWLEDGMENT

The ISTI-CNR co-authors acknowledge the financial support of the EC IST IP project "3D-COFORM" (IST-2008-231809).

REFERENCES

[1] K. Perlin, "An image synthesizer," *Computer Graphics*, vol. 19, no. 3, pp. 287–296, Jul. 1985.
[2] L.-Y. Wei and M. Levoy, "Texture synthesis over arbitrary manifold surfaces," in *SIGGRAPH*, 2001, pp. 355–360. [Online]. Available: http://portal.acm.org/citation.cfm?id=383259.383298
[3] G. Turk, "Texture synthesis on surfaces," in *SIGGRAPH01*, 2001, pp. 347–354.
[4] L.-Y. Wei, S. Lefebvre, V. Kwatra, and G. Turk, "State of the art in example-based texture synthesis," in *Eurographics 2009, State of the Art Report, EG-STAR*. Eurographics Association, 2009. [Online]. Available: http://www-sop.inria.fr/reves/Basilic/2009/WLKT09
[5] D. J. Heeger and J. R. Bergen, "Pyramid-based texture analysis/synthesis," in *ICIP*, 1995, pp. III: 648–651. [Online]. Available: http://dx.doi.org/10.1109/ICIP.1995.537718
[6] J. Portilla and E. P. Simoncelli, "A parametric texture model based on joint statistics of complex wavelet coefficients," *International Journal of Computer Vision*, vol. 40, no. 1, pp. 49–70, Oct. 2000. [Online]. Available: http://dx.doi.org/10.1023/A:1026553619983
[7] A. A. Efros and T. K. Leung, "Texture synthesis by non-parametric sampling," in *ICCV*, 1999, pp. 1033–1038. [Online]. Available: http://dx.doi.org/10.1109/ICCV.1999.790383
[8] L. Y. Wei and M. Levoy, "Fast texture synthesis using tree-structured vector quantization," in *SIGGraph-00*, 2000, pp. 479–488.
[9] L.-Y. Wei and M. Levoy, "Order-independent texture synthesis," Computer Science Department Stanford University, Tech. Rep., 2001.
[10] S. Lefebvre and H. Hoppe, "Parallel controllable texture synthesis," *ACM Transactions on Graphics*, vol. 24, no. 3, pp. 777–786, Jul. 2005.
[11] A. A. Efros and W. T. Freeman, "Image quilting for texture synthesis and transfer," in *SIGGRAPH 2001, Computer Graphics Proceedings*, ser. Annual Conference Series, E. Fiume, Ed. ACM Press / ACM SIGGRAPH, 2001, pp. 341–346. [Online]. Available: http://visinfo.zib.de/EVlib/Show?EVL-2001-122
[12] V. Kwatra, A. Schödl, I. Essa, G. Turk, and A. Bobick, "Graphcut textures: Image and video synthesis using graph cuts," *ACM Transactions on Graphics, SIGGRAPH 2003*, vol. 22, no. 3, pp. 277–286, July 2003.
[13] V. Kwatra, I. Essa, A. Bobick, and N. Kwatra, "Texture optimization for example-based synthesis," *ACM Transactions on Graphics*, vol. 24, no. 3, pp. 795–802, Jul. 2005.
[14] A. Lagae, S. Lefebvre, G. Drettakis, and P. Dutré, "Procedural noise using sparse Gabor convolution (proceedings of acm siggraph 2009)," *ACM Transactions on Graphics*, vol. 28, no. 3, 2009, to appear.
[15] J. M. Dischler, D. Ghazanfarpour, and R. Freydier, "Anisotropic solid texture synthesis using orthogonal 2D views," in *Computer Graphics Forum*, D. Duke, S. Coquillart, and T. Howard, Eds., vol. 17(3). Eurographics Association, 1998, pp. 87–95. [Online]. Available: http://visinfo.zib.de/EVlib/Show?EVL-1998-455
[16] D. Ghazanfarpour and J. M. Dischler, "Spectral analysis for automatic 3-D texture generation," *Computers & Graphics*, vol. 19, no. 3, pp. 413–422, May 1995.
[17] D. Ghazanfarpour and J.-M. Dischler, "Generation of 3D texture using multiple 2D models analysis," *Computer Graphics Forum*, vol. 15, no. 3, pp. 311–324, Aug. 1996, ISSN 1067-7055.

[18] J.-M. Dischler and D. Ghazanfarpour, "A survey of 3D texturing," *Computers & Graphics*, vol. 25, no. 1, pp. 135–151, 2001. [Online]. Available: http://dx.doi.org/10.1016/S0097-8493(00)00113-8
[19] R. Jagnow, J. Dorsey, and H. Rushmeier, "Stereological techniques for solid textures," *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 329–335, Aug. 2004.
[20] R. Jagnow, J. Dorsey, and H. Rushmeier, "Evaluation of methods for approximating shapes used to synthesize 3d solid textures," *ACM Transactions on Applied Perception*, vol. 4, no. 4, Jan. 2008.
[21] X. Qin and Y.-H. Yang, "Aura 3D textures," *IEEE Trans. Vis. Comput. Graph*, vol. 13, no. 2, pp. 379–389, 2007. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2007.31
[22] X. J. Qin and Y. H. Yang, "Basic gray level aura matrices: Theory and its application to texture synthesis," in *ICCV*, 2005, pp. I: 128–135. [Online]. Available: http://dx.doi.org/10.1109/ICCV.2005.43
[23] J. Kopf, C.-W. Fu, D. Cohen-Or, O. Deussen, D. Lischinski, and T.-T. Wong, "Solid texture synthesis from 2d exemplars," *ACM Trans. Graph.*, vol. 26, no. 3, p. 2, 2007.
[24] L.-Y. Wei, "Texture synthesis from multiple sources," in *SIGGRAPH '03: ACM SIGGRAPH 2003 Sketches & Applications*. New York, NY, USA: ACM, 2003, pp. 1–1.
[25] Y. Dong, S. Lefebvre, X. Tong, and G. Drettakis, "Lazy solid texture synthesis," *Comput. Graph. Forum*, vol. 27, no. 4, pp. 1165–1174, 2008. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2008.01254.x
[26] M. Ashikhmin, "Synthesizing natural textures," in *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*. New York, NY, USA: ACM, 2001, pp. 217–226.
[27] B. Cutler, J. Dorsey, L. McMillan, M. Müller, and R. Jagnow, "A procedural approach to authoring solid models," *ACM Transactions on Graphics*, vol. 21, no. 3, pp. 302–311, Jul. 2002.
[28] S. Owada, F. Nielsen, M. Okabe, and T. Igarashi, "Volumetric illustration: designing 3D models with internal textures," *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 322–328, Aug. 2004.
[29] N. Pietroni, M. A. Otaduy, B. Bickel, F. Ganovelli, and M. H. Gross, "Texturing internal surfaces from a few cross sections," *Comput. Graph. Forum*, vol. 26, no. 3, pp. 637–644, 2007. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2007.01087.x
[30] W. Matusik, M. Zwicker, and F. Durand, "Texture design using a simplicial complex of morphable textures," *ACM Transactions on Graphics*, vol. 24, no. 3, pp. 787–794, Jul. 2005.
[31] E. Praun, A. Finkelstein, and H. Hoppe, "Lapped textures," in *Proceedings of the Computer Graphics Conference 2000 (SIGGRAPH-00)*, S. Hoffmeyer, Ed. New York: ACMPress, Jul. 23–28 2000, pp. 465–470.
[32] K. Takayama, M. Okabe, T. Ijiri, and T. Igarashi, "Lapped solid textures: filling a model with anisotropic textures," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–9, 2008.