To appear in the ACM SIGGRAPH conference proceedings

Vector Solid Textures

Lvdi Wang[1,4] [1]Tsinghua University
Kun Zhou[2] [2]Zhejiang University
Yizhou Yu[3] [3]University of Illinois at Urbana-Champaign
Baining Guo[1,4] [4]Microsoft Research Asia

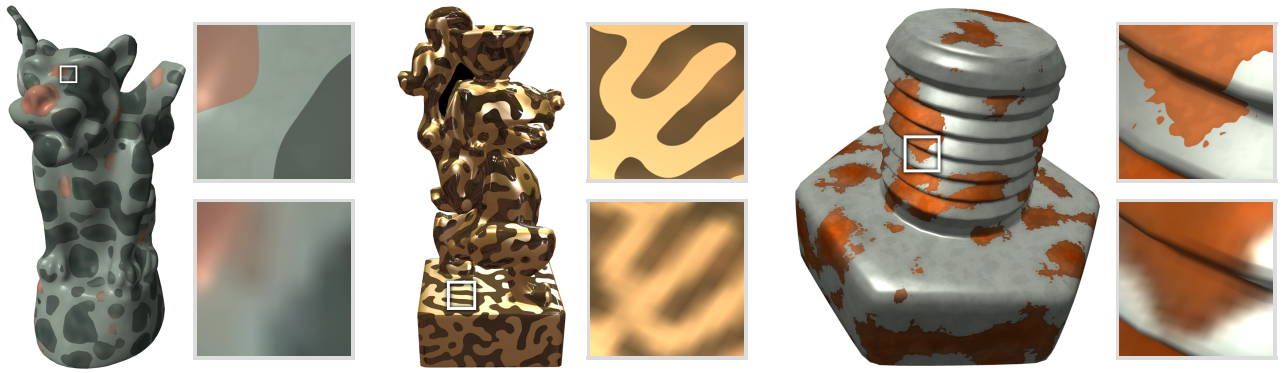

Figure 1: *Several vector solid textures rendered in real time. In each group, the upper inset shows a closer view while the lower inset is rendered using the bitmap solid texture from which the vector version is generated. In the middle group, the reflectance coefficients are stored along with the RGB channels. In the rightmost group, three different scales of the same vector solid texture are composited to yield complex self-similar boundaries.*

Abstract

In this paper, we introduce a compact random-access vector representation for solid textures made of intermixed regions with relatively smooth internal color variations. It is feature-preserving and resolution-independent. In this representation, a texture volume is divided into multiple regions. Region boundaries are implicitly defined using a signed distance function. Color variations within the regions are represented using compactly supported radial basis functions (RBFs). With a spatial indexing structure, such RBFs enable efficient color evaluation during real-time solid texture mapping. Effective techniques have been developed for generating such a vector representation from bitmap solid textures. Data structures and techniques have also been developed to compactly store region labels and distance values for efficient random access during boundary and color evaluation.

Keywords: Vector Images, Solid Textures, Texture Synthesis

1 Introduction

Texture maps used for real-time rendering are traditionally stored as bitmap images on graphics cards for random access. There has been a constant tension between texture resolution and memory usage. An overly fine resolution consumes too much GPU memory while an insufficient resolution leads to blurry interpolated texture mapping results. This is even worse for solid textures because of large memory consumption by 3D grids. A mipmap facilitates texture minification, but not magnification. On the other hand, vector images are well-suited for magnification because of their resolution-independent representation. The same vector image can be rasterized to a high-quality bitmap image at a wide range of resolutions. It seems promising to solve the above dilemma for solid textures by developing a specifically tailored vector representation.

Achieving this goal imposes a few challenging requirements. First, the vector representation should be feature-preserving during magnification. Features refer to edges where there exist sharp color or intensity changes. In a solid texture, such edges typically form surfaces within the texture volume. Second, the vector representation needs to be reasonably compact to alleviate memory consumption. A vector representation becomes advantageous only when it consumes a relatively small amount of memory but delivers a quality equivalent to a relatively high resolution bitmap. Third, the vector representation should support fast random access and real-time solid texture mapping on surfaces. Otherwise, it cannot become a replacement for bitmap textures.

In this paper, we introduce an effective vector representation for solid textures to meet the aforementioned requirements. It has the following important characteristics. First, it decomposes a texture volume into nonoverlapping regions along texture features. Region boundaries are implicitly defined using a signed distance function. Such a decomposition enables the preservation of rapid color changes across features. Second, color variations within the regions are compactly represented using radial basis functions (RBFs) with a finite support. Such RBFs support efficient color evaluation during real-time solid texture mapping. Third, data structures and techniques have been developed to compactly store region labels and distance values for efficient random access during boundary and color evaluation. Furthermore, this representation facilitates region-based texture composition and real-time texture editing operations, including parametric warp and local boundary softness.

We have developed effective techniques for generating such a vector representation from either 2D exemplars or existing solid textures, and for mapping such vectorized solid textures onto mesh surfaces in real time. During vectorization, color variations are fitted with a minimal number of RBFs using both nonlinear optimization and teleportation. The number of distinct region labels is

minimized by casting it as a graph coloring problem. A spatial indexing structure is also set up for RBFs so that relevant RBFs can be quickly looked up during real-time solid texture mapping.

Due to the use of signed distance functions and RBFs, our representation requires that the sharp features (region boundaries) in a solid texture can be identified by a binary mask and the regions have relatively smooth internal color variations.

2 Related Work

Solid Texture Synthesis Solid textures are useful in many scenarios, including modeling natural textures such as wood and marble, and representing the interior or cross sections of volumetric objects, such as fruits. Heeger and Bergen [1995] as well as Ghazanfarpour and Dischler [1995; 1996] did early work on example-based solid texture synthesis using parametric approaches. Wei attempted to synthesize solid textures from 2D input using a non-parametric method [Efros and Leung 1999; Wei 2001]. Based on stereological techniques, Jagnow et al. [2004] generate impressive results for aggregate materials. But their method is not applicable for general solid textures. Qin and Yang [2007] propose an approach for generating solid textures using aura matrices of the input exemplars. Kopf et al. [2007] present a widely applicable method that utilizes 2D texture optimization [Kwatra et al. 2005] and histogram matching. We use their method to generate solid textures from 2D exemplars. By considering the coherence within the 2D exemplars, Dong et al. [2008] propose a parallel solid synthesis algorithm that runs efficiently on the GPU. Finally, Takayama et al. [2008] develop an intuitive interface that allows the user to design anisotropic solid textures directly on objects using 3D exemplars.

In addition to texture synthesis, procedural textures (e.g. [Ebert et al. 1994; Worley 1996]) provide a practical alternative for solid texture generation. Such textures typically do not save explicit copies, but evaluate pointwise texture colors on the fly. Although impressive visual results have been achieved on certain types of textures, in general, it is still challenging to conceive a procedure that convincingly reproduces an arbitrarily chosen natural pattern. Lagae et al. [2009] introduce a procedural noise based on sparse Gabor convolution that offers intuitive control over the spectral density, anisotropy etc. of the noise. But the variety of materials that can be generated is still limited. Note that our vector texture representation relies on the same tricubic interpolation as Perlin's noise. However, we use it on signed distance values to determine region boundaries while Perlin used it for noise and gradient interpolation.

Vector Graphics and Image Vectorization 2D vector graphics, often in the form of charts, maps and clip arts, are curve-based and regions in-between curves are filled with uniform colors or color gradients. Nehab and Hoppe [2008] introduced an algorithm for random-access rendering of antialiased 2D vector graphics on the GPU. It can map vector graphics onto arbitrary surfaces. There has been much work on non-photographic image vectorization [Chang and Hong 1998; Zou and Yan 2001; Hilaire and Tombre 2006], i.e. converting such an image into 2D vector graphics. These algorithms are mainly designed for contour tracing and curve fitting.

Vectorization of full-color 2D raster images [Lecot and Levy 2006; Price and Barrett 2006; Sun et al. 2007; Orzan et al. 2008; Lai et al. 2009; Xia et al. 2009] has been popular recently. These images need a more generic vector representation that accounts for color variations across the image plane in addition to feature curves. Among them, high-quality image vectorization via a rectangular grid of Ferguson patches has been explored in [Sun et al. 2007; Lai et al. 2009]. Instead of using rectangular patches, Xia et al. [2009] exploits the topological flexibility of triangular patches with curved boundaries to perform automatic feature alignment and image vectorization. Diffusion curves [Orzan et al. 2008] model spatial color variations in a vectorized image as a diffusion from curves with both color and blur attributes.

Several softwares, such as VectorEye, Vector Magic, and AutoTrace have been developed for automatically converting bitmaps to vector graphics. Commercial tools (CorelDRAW, Adobe Live Trace, etc.) that help artists design and edit vector images are also available.

In addition to complete vector representations, there exist hybrid feature-based 2D texture representations (e.g. [Ramanarayanan et al. 2004; Sen 2004; Tumblin and Choudhury 2004; Tarini and Cignoni 2005; Parilov and Zorin 2008]). The basic idea of these methods is to improve the sharpness of the features in a magnified bitmap image by performing interpolation with respect to explicitly added feature boundaries. These methods represent features using vector primitives but still represent spatial color variations using a bitmap image. In comparison, our method works for 3D solid textures where features are represented as implicit surfaces instead of curves. In addition, we represent color variations using compactly supported RBFs, which are region-filling vector primitives more generic than uniform colors or color gradients.

3 Vector Texture Representation

Our vector solid texture representation consists of three key components: a set of *regions* with distinct region labels, a continuous *3D signed distance function*, and the weights and parameters of a set of *radial basis functions*. The region boundaries are implicitly defined by the zero isosurface of the signed distance function. This isosurface divides the texture volume into multiple connected components, and a region consists of one or multiple such connected components. The color variations within each region are represented separately using a distinct subset of radial basis functions.

Both the signed distance function and region labels are defined through a 3D discrete grid, denoted as $\mathcal{S}$, with $d \times d \times d$ nodes. The sampled signed distance values and region labels at the grid nodes are explicitly stored. We denote x_{ijk} as the 3D location of the grid node (i, j, k), where $i, j, k \in \{0, 1, \ldots, d-1\}$, and $D(x)$, $\ell(x)$ as the *signed distance value* and *region label*, respectively, at an arbitrary location x, which is not necessarily a grid node, inside the texture volume. To define the underlying continuous signed distance function, we calculate the signed distance at an arbitrary location by tricubic interpolation from the node values at the nearest $4 \times 4 \times 4$ subgrid. Tricubic interpolation guarantees a sufficient level of continuity of the signed distance function.

The region label at an arbitrary location x is determined as follows. If all nodes in its nearest $2 \times 2 \times 2$ subgrid have the same region label, $\ell(x)$ is assigned the same label too. Otherwise, there exists one or more boundary surfaces inside the subgrid, and the sign of the interpolated distance value at x determines $\ell(x)$.

Note that representing region labels and the distance function using a discrete grid is inherently different from representing colors on a grid. The former encodes feature-based texture segmentation results as well as the shape of region boundaries. Such geometric information is essential for any vector representations, and plays a crucial role in feature preservation and magnification.

4 Vector Texture Generation

This section describes an algorithm to convert a bitmap solid texture to our vector representation.

Our algorithm requires a bitmap solid texture with an additional

channel of signed distance as the input. Signed distance functions have been widely used as "feature maps" in texture synthesis [Lefebvre and Hoppe 2006; Kopf et al. 2007] to improve the synthesis quality. This signed distance function is an implicit representation of texture features, i.e. surfaces that correspond to sharp edges in the texture volume.

From 2D Exemplars Given an input 2D color texture and a binary feature mask from which a 2D signed distance transform can be computed, we adopt Kopf et al.'s optimization-based algorithm [2007] to synthesize a solid color texture with an additional channel of signed distance (Figure 2). The binary feature mask can be created by color thresholding or user interaction.

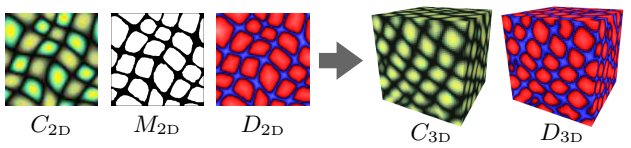

Figure 2: *Synthesizing the solid color texture C_{3D} with signed distance channel D_{3D} (colorized according to the sign for clarity) from a 2D exemplar C_{2D} and a signed distance transform D_{2D} computed from a binary feature mask M_{2D}.*

From Existing Solid Textures Given an existing solid texture where a signed distance is absent, we do not need the user to provide a 3D binary mask for computing the distance channel since manually creating a 3D mask is often impractical. Instead, the user only needs to create a 2D mask in a 2D slice or the original 2D exemplar (if available) of the solid texture, and we can synthesize a 3D signed distance channel automatically. This problem is in the same spirit as Image Analogies [Hertzmann et al. 2001]: given a 2D color texture C_{2D}, a 2D signed distance function D_{2D}, and a 3D color texture C_{3D}, we would like to generate a 3D function D_{3D} that relates to C_{3D} in the same way as D_{2D} relates to C_{2D} (Figure 3). Although the idea is relatively simple, we find it works surprisingly well for solid textures we have seen, especially those where simple color thresholding cannot yield a satisfactory result.

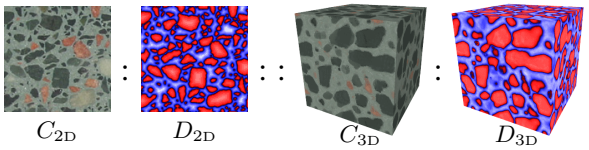

Figure 3: *The problem of synthesizing a 3D distance function for an existing solid texture can be formulated using the notation in Image Analogies [Hertzmann et al. 2001].*

Note that the synthesized distance channel is no longer an accurate distance function, but an approximate one. Nevertheless, it suffices as an implicit definition of feature surfaces which only need distance values at locations close to the zero isosurface.

The zero isosurface of the signed distance function divides the original solid texture along its sharp features into multiple connected components. We initially consider every connected component as a distinct region, and later may merge multiple connected components into the same region (Section 5.2). Every grid *cell* (a $2\times2\times2$ subgrid) can be part of at most two different connected components if connectedness is defined using 26 neighbors. We must also consider the tileability of the original texture so that nodes that are connected after tiling should belong to the same region.

4.1 RBF Color Fitting

Given the color C of the original solid texture (defined on the regular grid $\mathcal{S}$), we use a set of radial basis functions (RBFs) to approximate C. To avoid color bleeding across sharp features, we assign a dedicated set of RBFs to each region. The approximated color at a location x inside region p is defined as

$$\tilde{C}(x) = \bar{C}_p + \sum_{q=1}^{m_p} w_{pq} B_{pq}(x), \tag{1}$$

where $\bar{C}_p$ is the average color of the nodes in region p, m_p is the number of RBFs assigned to region p, and B_{pq} is the q-th basis function in region p and is defined as

$$B_{pq}(x) = \phi\left(\frac{\|x - c_{pq}\|}{r_{pq}}\right). \tag{2}$$

We choose the Wyvill function [Wyvill et al. 1986] for ϕ because of its compact support and low cost to evaluate:

$$\phi(r) = \begin{cases} 1 - \frac{4}{9}r^6 + \frac{17}{9}r^4 - \frac{22}{9}r^2, & r \le 1 \\ 0, & r > 1 \end{cases}$$

The goal of RBF color fitting can now be formulated as: given the original texture color C, κ regions in the texture volume, find n RBFs that minimize the following objective function:

$$\sum_{p=1}^{\kappa} \sum_{x_{ijk}\in\mathcal{S}_p} \left\|\tilde{C}(x_{ijk}) - C(x_{ijk})\right\|^2, \tag{3}$$

where $\sum_p m_p = n$, $\mathcal{S}_p$ is the subset of grid nodes inside region p.

The color fitting algorithm first chooses n random locations inside the texture volume as the initial centers of the RBFs. A relaxation procedure can be applied to make the initial distribution of RBF centers more uniform. An RBF is assigned to a region if its initial center is located inside that region. Its initial weight $w = C(x) - \bar{C}(x)$ and radii set to a user-provided default value. To solve the nonlinear optimization in (3), we adopt the L-BFGS-B minimizer [Zhu et al. 1997]. L-BFGS-B is a gradient-based method with bound constraints. Let c_{p*}, r_{p*}, and w_{p*} be the center location, radius, and weight of an RBF in region p, respectively. We bound these variables as follows:

$$\begin{aligned} 0 \quad &< c_{p*} < \quad 1, \\ 0.5n^{-1/3} \quad &\le r_{p*} \le \quad 2n^{-1/3}, \\ w_{\min} - \frac{w_{\max} - w_{\min}}{2} \quad &\le w_{p*} \le \quad w_{\max} + \frac{w_{\max} - w_{\min}}{2}, \end{aligned}$$

where

$$w_{\min} = \min_{x_{ijk}\in\mathcal{S}_p} \left[C(x_{ijk}) - \bar{C}(x_{ijk})\right],$$
$$w_{\max} = \max_{x_{ijk}\in\mathcal{S}_p} \left[C(x_{ijk}) - \bar{C}(x_{ijk})\right].$$

Due to high nonlinearity, the minimizer can be easily trapped in local minima. We employ a teleportation scheme similar to that in [Cohen-Steiner et al. 2004] and [Zhou et al. 2008]: after the minimizer converges, we move the most insignificant RBF to the location of the maximum fitting error and invoke the minimizer again to see if the overall error can be further reduced. We define the *significance* of an RBF as the difference between the objective function computed with and without this RBF. Let B_{p*} be an RBF in region p, the significance of B_{p*} is defined as

$$\sum_{x_{ijk}\in\mathcal{S}_p} \|\tilde{C}(x_{ijk}) - w_{p*}B_{p*}(x_{ijk}) - C(x_{ijk})\|^2 - \sum_{x_{ijk}\in\mathcal{S}_p} \|\tilde{C}(x_{ijk}) - C(x_{ijk})\|^2. \tag{4}$$

When teleporting an RBF to a new location that is inside a different region, the algorithm also dynamically changes the membership of the RBF to that region. The inclusion of this teleportation scheme leads to a further reduction of the objective function by $40\%\sim50\%$.

5 Compact Storage

In this section, we discuss a few effective techniques for reducing the amount of memory required for storing the signed distance function and region labels.

5.1 Distance Quantization

The sampled signed distance values are initially stored as a $d\times d\times d$ array of 32-bit floating-point numbers. However, notice that the nodes far from the region boundaries (with larger absolute distance value) have little effect on the shape of the region boundaries. We can thus clamp the distance values to a relative small range. Furthermore, we have found that the clamped distance values can be quantized using only 4 bits without visually degrading the quality of the implicitly defined region boundaries (see Figure 4).

For a generic distance function, it may be feasible to adopt an adaptively-sampled distance field (ADF) [Frisken et al. 2000] for compression. However, due to the complexity of the internal regions in most solid textures, we have found that the overhead of storing the octree structure is too high to make it practical in our case (with an exception discussed in Section 5.3).

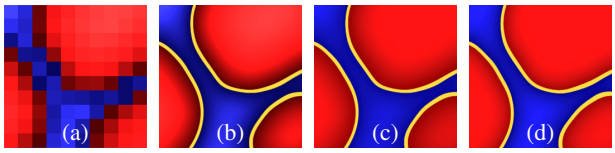

Figure 4: *The discrete signed distance field (a) implicitly defines the region boundaries as highlighted in (b). Quantizing the 32-bit distance values in (b) using 8 bits (c) or even 4 bits (d) does not visually degrade the quality of the boundary surfaces.*

5.2 Region Relabeling

We have previously assigned a distinct region label to every connected component. For a typical 128^3 solid texture, there could be over 1000 connected components, requiring at least 10 bits for each region label. Let us recall two facts. First, the main goal of labeling a region is to prevent this region from being affected by RBFs in other regions; second, all the RBFs are compactly supported and the maximum radius is bounded. They indicate that if two regions satisfy the condition that neither region's RBFs affect the other region, they can actually share the same label (Figure 5). Since each region has its own average color, we add a special "RBF" to each region. The center location and the radius of this RBF are set to those of the bounding sphere of the region.

This becomes a classical graph coloring problem: given the current region labels and the RBFs in each region, we can build an undirected graph G, where each vertex corresponds to a region and an edge (v_i, v_j) means there exists at least one RBF of region i that affects region j or vice versa. Finding the smallest number of colors needed to color an arbitrary graph (a.k.a. the chromatic number) is NP-complete. But for all the textures we have tested, a greedy algorithm with the Welsh-Powell heuristic [1967] is good enough to find a solution with no more than 18 colors. The Welsh-Powell algorithm first sorts all the vertices in a graph according to their degrees and then processes the vertices in order and assigns the smallest numbered available color to each vertex.

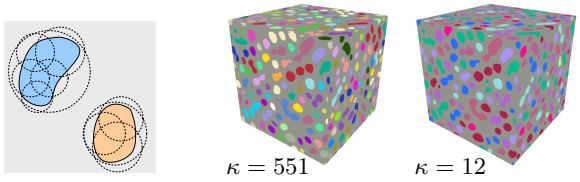

Figure 5: *Region relabeling. Left: the blue region and the yellow region can share the same label because their RBFs (dashed circles) do not affect each other. Middle and right: the colorized region labels before and after relabeling.*

5.3 Region Label-Pair Storage

The region labels, unlike the distance function, are constant within each region. There should be more compact forms than directly storing them at the regular grid nodes. However, due to the complex intrinsic structures in most solid textures, applying common spatial compression methods such as Run-Length Encoding or Octrees directly on region labels often gives rise to expensive storage overhead, not to mention the adverse effects on rendering performance.

As mentioned in Section 4, there exist at most two different region labels in each cell (a $2\times2\times2$ subgrid) of the regular grid $\mathcal{S}$. And if there are two different labels, the cell must straddle a zero isosurface of the signed distance function, i.e. some of the nodes in the cell are associated with a positive distance while the others are associated with a negative distance. Based on this fact, we chose to store *two* region labels, namely the label of the closest positive region (denoted as $\ell_\oplus$) and the label of the closest negative region (denoted as $\ell_\ominus$), inside each grid cell. To compute such closest regions, we initialize the region labels inside a cell using the region labels at the corners of the cell. Obviously in some cells, where no isosurfaces cut through, either $\ell_\oplus$ or $\ell_\ominus$ is *undefined*. We apply the fast marching algorithm [Sethian 1999] to propagate the initial $\ell_\oplus$ and $\ell_\ominus$ in two separate passes. Such propagation fills all undefined labels, as illustrated in Figure 6.

Note that although storing the label pairs seems to have doubled the storage requirement, the resulting spatial distribution of the label pairs actually makes them much easier to compress because they have much fewer transitions along any spatial direction. Indeed, we can build an octree data structure for the label pairs. We arrange the octree structure in a similar way as [Lefebvre et al. 2005]. For the textures in this paper, the use of octrees saves at least 70% (and at most 99.998%) of the original storage for region label pairs. Furthermore, we have observed about 5% improvement on the rendering performance due to reduced texture memory access.

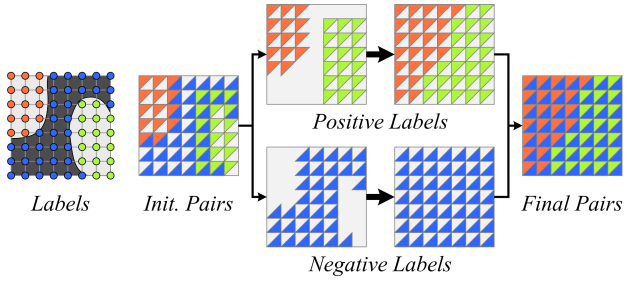

Figure 6: *A 2D illustration of the region label pairs. From left to right: the region labels on each grid node, the initial label pairs on each grid cell, the separated positive (top) and negative (bottom) labels, the labels after propagation, and the final label pairs.*

6 Rendering

As a form of solid textures, our vector representation supports efficient random access to the color at any given 3D location (texture coordinate), as requested by a pixel shader. Basically, to calculate the color of a pixel with texture coordinate x, the pixel shader needs to access three data structures: the quantized signed distance function D, the region label-pair octree P, and the set of RBFs, as described in Algorithm 1.

Algorithm 1 Naïve Vector Solid Texture Fetch

procedure VECTEXFETCH(x) ▷ $x \in [0,1]^3$
  $d \leftarrow D(x)$ ▷ using tricubic interpolation
  $(\ell_\oplus, \ell_\ominus) \leftarrow P(x)$
  if $d > 0$ **then** $\ell \leftarrow \ell_\oplus$
  else $\ell \leftarrow \ell_\ominus$
  $color \leftarrow 0$
  for all RBF (w, c, r) in region ℓ **do**
    $color \leftarrow color + w\phi(\frac{\|x-c\|}{r})$
  return $color$

In the rest of this section we introduce several techniques to improve the performance and quality of Algorithm 1. For clarity we keep the description in an abstract way and leave the implementation details to Section 7.

6.1 RBF Spatial Indexing

To evaluate the color at a given location inside a region, a simple way would evaluate the value of every RBF (assigned to the region) at that location, which is expensive for regions with a relatively large number of RBFs. Since all the RBFs are compactly supported and their maximum radius is bounded, the number of RBFs that simultaneously affect a given location is usually quite low. Therefore, we set up a spatial indexing structure by dividing the texture volume into a sparse grid of size $d_s \times d_s \times d_s$. In each grid cell, we record the RBFs that potentially affect this cell.

The spatial indexing structure can be built quickly when loading the texture. Although it can boost the rendering performance significantly, it also consumes a small amount of additional storage. Therefore, the choice of d_s provides a space-speed tradeoff. In practice, we have found that $d_s = 8$ strikes a balance in most cases.

6.2 Region Boundary Softening

Since we segment the texture volume into regions and perform RBF fitting for each region separately, there exists a color contrast across region boundaries in final texture mapping results. However, a clear color contrast across region boundaries sometimes may look unnatural. Therefore, we make use of the precomputed region label pairs (Section 5.3) to perform boundary softening by blending colors from two adjacent regions.

The color contrast across a region boundary is caused by abrupt region membership changes across the boundary. Suppose the distance value $D(x)$ at a location x is positive. From the region label pair $(\ell_\oplus, \ell_\ominus)$ at x, we not only know the region label of x, $\ell_\oplus$, but also know the label of the closest neighboring region $\ell_\ominus$, and $|D(x)|$ is the shortest distance from x to the boundary between regions $\ell_\oplus$ and $\ell_\ominus$. Thus, during real-time solid texture mapping, we evaluate two colors (denoted as $color_\oplus$ and $color_\ominus$) on the fly at the same location using RBFs from region $\ell_\oplus$ and $\ell_\ominus$, respectively. The final color at any location x within a distance threshold δ from a region boundary (i.e. $|D(x)| < \delta$) is a linear blend between two such evaluated colors:

$$color \leftarrow \text{LERP}(color_\ominus, color_\oplus, (D(x) + \delta)/(2\delta)). \quad (5)$$

Note that unlike in 2D vector graphics, the meaning of *boundary softness* in a vector solid texture is two-fold. From a 3D point of view, it defines an intrinsic property of the solid texture that is similar to a 3D extension of the blur attribute in Diffusion Curves [Orzan et al. 2008]. The threshold in this context is denoted as δ_{3D} and is a user-adjustable parameter. From a 2D point of view, the softness also controls how the rasterized vector texture is antialiased in screen space.

To perform accurate antialiasing, we must carefully choose the threshold, denoted as δ_{2D} in this context, so that a pixel participates blending *if and only if* the projected region boundary cut through it. In practice, for each screen pixel we let $\delta_{2D} = 0.5\|\nabla d\|$, where d is the signed distance value at the pixel and ∇d is the *screen space* gradient of d. Hence, $|d| < \delta_{2D}$ means there exists a region boundary within the range of half the pixel width from the current pixel center, and we compute the color according to (5).

During rendering, the 3D and the 2D boundary softness thresholds are incorporated as $\delta = \max(\delta_{3D}, \delta_{2D})$.

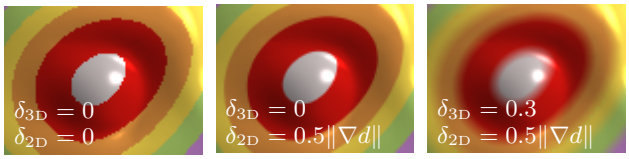

Figure 7: *Boundary softening with different thresholds.*

6.3 Mipmapping

Boundary softening alone cannot completely avoid aliasing in minification because more than two regions may fall into a single pixel. Our vector texture representation can also employ mipmapping in a similar way as bitmap textures do. To generate a mipmap for a vector solid texture, we first generate the texture pyramid for the original bitmap solid texture. Then each level of the pyramid is vectorized separately as described in Section 4, using $1/8$ as many as the RBFs in the finer level. Figure 8 shows several mipmap levels of a vector solid texture.

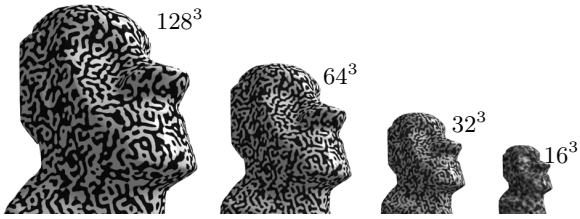

Figure 8: *Four mipmap levels of a vector solid texture.*

7 Results and Discussion

We have generated a number of vector solid textures that cover a wide range of texture types, as shown in Figure 12. Table 1 summarizes the storage and rendering performance of each texture. Generally, for a solid texture with three 8-bit color channels, our vector representation consumes $17\% \sim 26\%$ the storage of a bitmap version *with the same grid resolution*. This ratio is even lower for textures with more channels (e.g. displacement, appearance coefficients) since these additional channels are approximated with the same set of RBFs in our vector textures. Furthermore, to achieve a comparable quality after magnification, a bitmap solid texture with a much higher grid resolution is required. Note that the Kiwi example in Figure 12 (f) takes the most number of RBFs to get a good

Dataset	*Grid*	*# RBFs*	*Storage (MB)*		*Rendering (FPS)*
			Static	*Total*	
Fig. 1 (left)	128^3	5183	1.117	1.173	148.6
Fig. 1 (middle)	128^3	202	1.002	1.016	549.7
Fig. 1 (right)	128^3	2015	1.050	1.087	247.1
Fig. 10	96^3	5002	0.479	0.569	96.5
Fig. 12 (a)	128^3	1387	1.120	1.152	252.4
Fig. 12 (b)	128^3	69	1.497	1.548	146.4
Fig. 12 (c)	128^3	1060	1.074	1.100	335.5
Fig. 12 (d)	128^3	352	1.004	1.026	363.2
Fig. 12 (e)	64^3	19	0.126	0.135	741.1
Fig. 12 (f)	128^3	7137	1.199	1.275	140.8
Fig. 12 (g)	128^3+64^3	3893	1.290	1.369	150.5
Fig. 12 (h)	128^3	4735	1.141	1.230	144.0

Table 1: *Statistics of the vector solid textures in this paper. The total size includes both the static size and the size of the RBF spatial index. All the textures use an 8^3 grid for RBF spatial indexing. Figure 12 (g) is composited from two vector solid textures. The performance is measured by rendering a textured cube in an 800^2 window on an NVIDIA GeForce 8800 GTX.*

fitting result due to its complex structure and color variation. The original bitmap texture was created manually using a special tool developed by ourselves. It is difficult to generate such a texture using procedural or example-based synthesis methods.

Given an input bitmap solid texture with an accompanying signed distance function, the vectorization is fully automatic and takes about $1 \sim 20$ minutes depending on the input resolution and the number of RBFs.

We have implemented our rendering algorithm using Direct3D 9. Since current graphics hardware does not natively support a single-channel 4-bit texture format, we encode the $d \times d \times d$ distance function as a $d \times d \times (d/2)$ texture in `A4L4` format. The octree for region label pairs can be packed into an `A1R5G5B5` 3D texture where the 1-bit alpha value indicates whether a tree node is a leaf. For textures with only three color channels, the RBFs are stored in two 1D textures: an `A16B16G16R16F` texture for the center locations and radius values, and an `A8R8G8B8` texture for the weights and region labels, respectively. Finally, the RBF spatial indexing structure consists of another 32-bit 3D texture for the grid and a 16-bit 2D texture for the indirection table (similar to [Nehab and Hoppe 2008]). We use the `ddx`, `ddy` intrinsic functions to calculate the screen-space gradient of the signed distance function in the pixel shader. The tricubic interpolation of signed distance values can be accelerated significantly by using the method introduced by Sigg and Hadwiger [2005].

7.1 Texture Composition

In the field of solid modeling, implicit representations have the unique strength to generate new geometric models by performing algebraic or Boolean operations on two or more existing primitives. Similarly, in our vector texture representation we can define the signed distance function as the result of operations on multiple distance functions. Figure 9 enumerates several possible operations.

One example is to generate "self-similar" textures: during rendering, we replace the signed distance value, $D(x)$, at 3D location x by the following formula:

$$D'(x) = D(x) + \sum_{i=1}^{k} a_i D(b_i x).$$

In this way, complex fractal-like boundaries can be generated with no additional storage and little performance overhead, as shown in Figure 1 (right), where $k=2, a_1=0.5, a_2=0.3, b_1=4, b_2=16$.

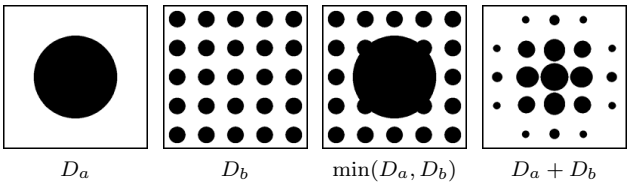

Figure 9: *Combining two signed distance functions D_a and D_b to define more complex region boundaries.*

Another possibility is to generate a hybrid texture from two different textures, as demonstrated in Figure 12 (g). In this case, the region label at a 3D location is determined by two signed distance functions.

7.2 Real-Time Texture Manipulation

Based on our vector texture representation, we have implemented several simple but interesting interactive tools that allow the user to edit certain aspects of the texture in real time.

Parametric Warp We provide a stroke-based tool that emulates the Liquify filter in Adobe Photoshop. By sketching on the surface of a mesh, the user can modify (e.g. "bloat" or "pucker") the texture coordinates of the mesh vertices within a given distance to the stroke to create effects such as spatially-varying texton size. Although this tool can also be applied to bitmap solid textures, the resulting distortion can cause severe blurring, which is not the case for vector solid textures (see Figure 12 (e)).

Local Boundary Softness As described in Section 6.2, the boundary softness of a vector solid texture is a user-adjustable global parameter. It is also possible to decide the softness based on local variables. For example, in Figure 10, the softness at a pixel is determined by the depth in the camera space to create a depth-of-field effect.

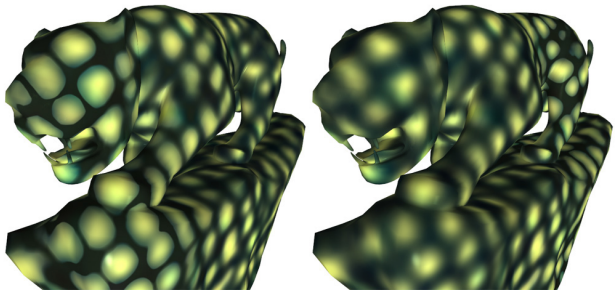

Figure 10: *The boundary softness at each pixel is determined by the camera-space depth. The "focal plane" is set to the head (left) and hip (right) of the tiger model, respectively.*

7.3 Limitations

Due to the nature of signed distance functions, sharp features (i.e. region boundaries) in the input bitmap texture must be identifiable by a single binary mask, or in terms of the graph theory, the regions should be 2-colorable. Although this is true for a wide range of solid textures, there exist exceptions, e.g. Figure 11 (left), that violate this requirement. A possible solution is to use multiple binary masks (hence multiple signed distance functions) to identify different regions, as shown in Figure 11 (right). Future research could be performed to find an algorithm that automatically generates multiple binary feature masks (if needed) for a given color texture.

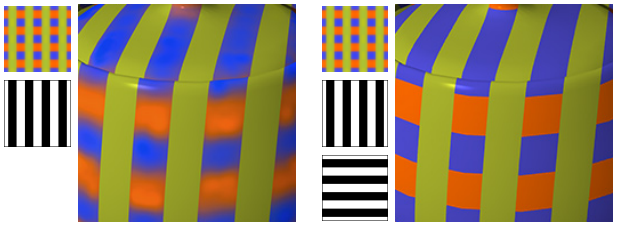

Figure 11: *In order to achieve a good vectorization of the upper left texture, two binary masks need to be used.*

Another limitation of our method is that boundary softness cannot be set to an arbitrary value. The upper limit of distance threshold δ is determined by the spatial distribution of region label pairs. If δ is too large, a pixel color could be blended from incorrectly evaluated region colors and artifacts may appear.

Since RBF-based color fitting is a lossy procedure, certain high-frequency details in the original bitmap solid texture could be smoothed out. It is worthwhile to investigate how to represent such details in vectorization results.

8 Conclusion

We have introduced a compact random-access vector representation for solid textures. It delivers high-quality rendering results by preserving both the sharp features and the smooth color variations of a solid texture. With a spatial indexing structure, our representation enables efficient color evaluation during real-time solid texture mapping. Due to its resolution-independent nature, our representation supports several interesting applications such as feature-preserving texture composition and parametric warping. We have developed effective techniques to generate vector solid textures from either 2D exemplars or existing bitmap solid textures. One interesting direction for future work is to develop a user-friendly interface for designing vector solid textures from scratch.

Acknowledgements

We would like to thank Johannes Kopf and colleagues for sharing their data, Yue Dong and Minmin Gong for providing helpful suggestions, and the anonymous reviewers for their insightful comments. Some models in this paper are provided by AIM@SHAPE shape repository and Evermotion. Kun Zhou was partially supported by NSFC (No. 60825201), the 973 program of China (No. 2009CB320801) and NVIDIA.

References

CHANG, H.-H., AND HONG, Y. 1998. Vectorization of hand-drawn image using piecewise cubic bézier curves fitting. *Pattern recognition 31*, 11, 1747–1755.

COHEN-STEINER, D., ALLIEZ, P., AND DESBRUN, M. 2004. Variational shape approximation. *ACM Trans. Graph. 23*, 3, 905–914.

DONG, Y., LEFEBVRE, S., TONG, X., AND DRETTAKIS, G. 2008. Lazy solid texture synthesis. *Computer Graphics Forum 27*, 4, 1165–1174.

EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND WORLEY, S. 1994. *Texturing and Modeling: A Procedural Approach*. Academic Press.

EFROS, A., AND LEUNG, T. 1999. Texture synthesis by non-parametric sampling. In *ICCV '99*, 1033–1038.

FRISKEN, S. F., PERRY, R. N., ROCKWOOD, A. P., AND JONES, T. R. 2000. Adaptively sampled distance fields: a general representation of shape for computer graphics. In *Proceedings of SIGGRAPH 2000*, 249–254.

GHAZANFARPOUR, D., AND DISCHLER, J.-M. 1995. Spectral analysis for automatic 3-D texture generation. *Computers and Graphics 19*, 3, 413–422.

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

HEEGER, D., AND BERGEN, J. 1995. Pyramid-based texture analysis/synthesis. In *Proceedings of SIGGRAPH '95*, 229–238.

HERTZMANN, A., JACOBS, C. E., OLIVER, N., CURLESS, B., AND SALESIN, D. H. 2001. Image analogies. In *Proceedings of SIGGRAPH 2001*, 327–340.

HILAIRE, X., AND TOMBRE, K. 2006. Robust and accurate vectorization of line drawings. *IEEE Trans. Pattern Anal. Mach. Intell. 28*, 6, 890–904.

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

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, Article 2.

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

LAGAE, A., LEFEBVRE, S., DRETTAKIS, G., AND DUTRÉ, P. 2009. Procedural noise using sparse Gabor convolution. *ACM Trans. Graph. 28*, 3, Article 54.

LAI, Y.-K., HU, S.-M., AND MARTIN, R. 2009. Automatic and topology-preserving gradient mesh generation for image vectorization. *ACM Trans. Graph. 28*, 3, Article 85.

LECOT, G., AND LEVY, B. 2006. Ardeco: Automatic region detection and conversion. In *EGSR 2006*, 349–360.

LEFEBVRE, S., AND HOPPE, H. 2006. Appearance-space texture synthesis. *ACM Trans. Graph. 25*, 3, 541–548.

LEFEBVRE, S., HORNUS, S., AND NEYRET, F. 2005. Octree textures on the GPU. In *GPU Gems 2*. ch. 37.

NEHAB, D., AND HOPPE, H. 2008. Random-access rendering of general vector graphics. *ACM Trans. Graph. 27*, 5, Article 135.

ORZAN, A., BOUSSEAU, A., WINNEMÖLLER, H., BARLA, P., THOLLOT, J., AND SALESIN, D. 2008. Diffusion curves: a vector representation for smooth-shaded images. *ACM Trans. Graph. 27*, 3, Article 92.

PARILOV, E., AND ZORIN, D. 2008. Real-time rendering of textures with feature curves. *ACM Trans. Graph. 27*, 1, Article 3.

PRICE, B., AND BARRETT, W. 2006. Object-based vectorization for interactive image editing. *The Visual Computer 22*, 9 (sep), 661–670.

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

RAMANARAYANAN, G., BALA, K., AND WALTER, B. 2004. Feature-based textures. In *EGSR 2004*, 65–73.

Figure 12: *Different vector solid textures rendered in real time. The texture in (c) is created by compositing multiple scales of the same distance function. The texture in (g) is created by compositing two different vector solid textures. The lower right half of (d) shows the underlying region boundaries without softening.*

SEN, P. 2004. Silhouette maps for improved texture magnification. In *Graphics Hardware 2004*, 65–73.

SETHIAN, J. 1999. *Level Set Methods and Fast Marching Methods*. Cambridge University Press.

SIGG, C., AND HADWIGER, M. 2005. Fast third-order texture filtering. In *GPU Gems 2*. ch. 20.

SUN, J., LIANG, L., WEN, F., AND SHUM, H.-Y. 2007. Image vectorization using optimized gradient meshes. *ACM Trans. Graph. 26*, 3, Article 11.

TAKAYAMA, K., OKABE, M., IJIRI, T., AND IGARASHI, T. 2008. Lapped solid textures: filling a model with anisotropic textures. *ACM Trans. Graph. 27*, 3, Article 53.

TARINI, M., AND CIGNONI, P. 2005. Pinchmaps: textures with customizable discontinuities. *Computer Graphics Forum 24*, 3, 557–568.

TUMBLIN, J., AND CHOUDHURY, P. 2004. Bixels: Picture samples with sharp embedded boundaries. In *EGSR 2004*, 186–196.

WEI, L.-Y. 2001. *Texture Synthesis by Fixed Neighborhood Searching*. PhD thesis, Stanford University.

WELSH, D., AND POWELL, M. 1967. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal 10*, 1, 85–86.

WORLEY, S. 1996. A cellular texture basis function. In *Proceedings of SIGGRAPH '96*, 291–294.

WYVILL, G., MCPHEETERS, C., AND WYVILL, B. 1986. Data structure for soft objects. *The Visual Computer 2*, 4, 227–234.

XIA, T., LIAO, B., AND YU, Y. 2009. Patch-based image vectorization with automatic curvilinear feature alignment. *ACM Trans. Graph. 28*, 5, Article 115.

ZHOU, K., REN, Z., LIN, S., BAO, H., GUO, B., AND SHUM, H.-Y. 2008. Real-time smoke rendering using compensated ray marching. *ACM Trans. Graph. 27*, 3, Article 36.

ZHU, C., BYRD, R., LU, P., AND NOCEDAL, J. 1997. L-BFGS-B: Fortran subroutines for large-scale bound constrained optimization. *ACM Trans. Math. Softw. 23*, 4, 550–560.

ZOU, J. J., AND YAN, H. 2001. Cartoon image vectorization based on shape subdivision. In *Proceedings of Computer Graphics International 2001*, 225–231.