Multiscale Vector Volumes

Lvdi Wang
Tsinghua University

Yizhou Yu
University of Illinois at Urbana-Champaign
The University of Hong Kong

Kun Zhou
Zhejiang University

Baining Guo
Microsoft Research Asia
Tsinghua University

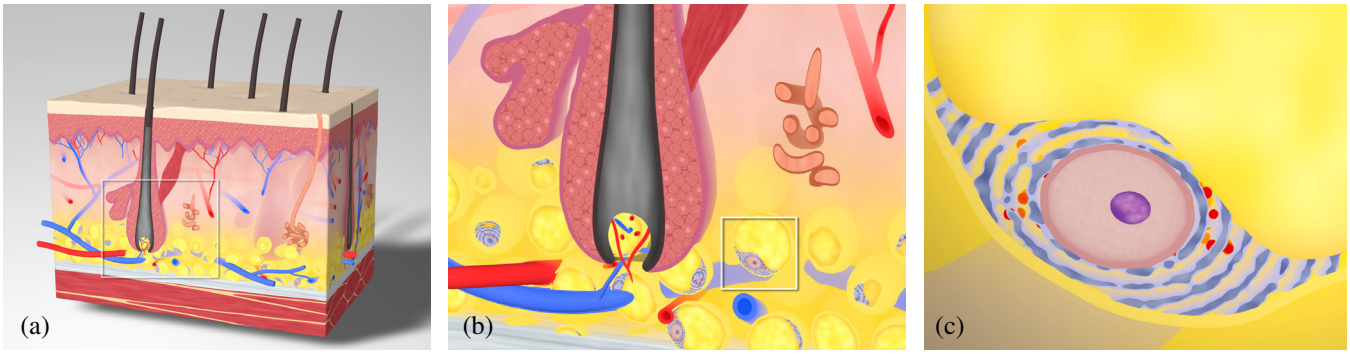

Figure 1: *A multiscale vector volume depicting a portion of human skin viewed at different scales. (b) Zoomed view of the selected area in (a). (c) Zoomed view of the selected area in (b). This model was created from scratch with our representation and consumes 6.8 MB storage.*

Abstract

We introduce *multiscale vector volumes*, a compact vector representation for volumetric objects with complex internal structures spanning a wide range of scales. With our representation, an object is decomposed into components and each component is modeled as an *SDF tree*, a novel data structure that uses multiple signed distance functions (SDFs) to further decompose the volumetric component into regions. Multiple signed distance functions collectively can represent non-manifold surfaces and deliver a powerful vector representation for complex volumetric features. We use *multiscale embedding* to combine object components at different scales into one complex volumetric object. As a result, regions with dramatically different scales and complexities can co-exist in an object. To facilitate volumetric object authoring and editing, we have also developed a scripting language and a GUI prototype. With the help of a recursively defined spatial indexing structure, our vector representation supports fast random access, and arbitrary cross sections of complex volumetric objects can be visualized in real time.

Keywords: volumetric modeling, multiscale representations

Links: DL PDF WEB

1 Introduction

Most natural organisms and materials, such as the human body, fruits and sedimentary rocks, have rich and complex volumetric structures, patterns and color variations. Constructing high-quality digital models for such natural organisms and materials is of vital importance because they exist everywhere, and literally everything we see is either a living being, a natural material or something made from these two. Since volumetric models allow us to visualize the internal structure of an object, they are valuable graphical contents that can be used in biomedical research, scientific visualization as well as educational and training activities.

Nevertheless, constructing high-quality digital models of natural organisms and materials with complex volumetric properties is an extremely challenging task. *First*, how can we compactly represent volumetric structures and patterns spanning a wide range of scales? Taking the human body as an example, it has a skeleton, organs and tissues at the macroscopic scale, cellular structures and neuronal fibers at an intermediate scale as well as proteins and genes at the molecular scale. *Second*, how can we represent high-frequency features in a resolution-independent way? Physical and appearance properties (e.g. color) change abruptly across different materials or volumetric components. Such discontinuities typically form thin surface sheets, which may join or split following an irregular pattern, giving rise to a non-manifold structure (Figure 2). To prevent visual artifacts when zooming into these high-frequency structures, a resolution-independent vector representation is desired. A volumetric object representation not only needs to depict complex multiscale structures, but also should be easy to use, which means it should be easy to create novel objects and easy to view existing objects in this representation. Thus, a volumetric object representation should have the following desired properties:

- *Expressiveness*: It should be able to represent volumetric objects with spatial structures spanning a wide range of scales and including complex non-manifold features. High-frequency features should remain sharp during magnification.
- *Ease of editing*: It should be easy and intuitive to create novel objects and edit existing ones using this representation.
- *Random access*: To be able to provide a timely response to user interactions, fast visualization is required, which further demands efficient random access to the volumetric content.
- *Compactness*: Given the limited memory on current graphics cards, the representation should be as compact as possible.

	Voxel	Procedural	VST [Wang et al. 2010]	DSs [Takayama et al. 2010]	**Our method**
Expressiveness	*Yes*	*Domain specific*	*Limited (two-colorable)*	*Limited (smooth variations)*	*Yes*
Ease of editing	*No*	*Non-intuitive*	*Limited*	*Yes*	*Yes*
Random access	*Very efficient*	*Very efficient*	*Efficient*	*Not efficient*	*Efficient*
Compactness	*No*	*Very compact*	*Compact*	*Compact*	*Compact*

Table 1: *Comparison among different volumetric modeling approaches.*

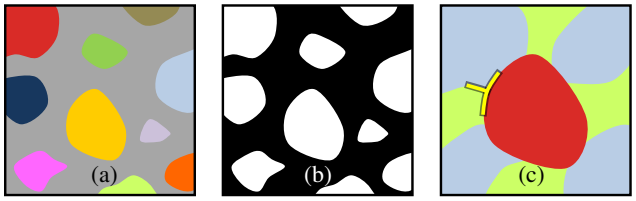

Figure 2: *The regions in (a) are two-colorable as they can be distinguished by a single binary mask (b). In (c) there are more than two regions adjacent to each other, therefore the regions are not two-colorable. The boundaries between regions contain non-manifold features (e.g. T-junctions as highlighted in yellow).*

In this paper, we introduce *multiscale vector volumes*, a compact vector representation for volumetric objects with complex internal structures spanning a wide range of scales.

The key idea is to decompose a complex object into individual volumetric components with respect to its multiscale structure, and preserve the boundary shape of each component in a resolution-independent way. This is achieved by hierarchically dividing an entire volumetric object into multiple non-overlapping regions using multiple continuous signed distance functions (SDFs). These continuous SDFs collectively deliver a powerful resolution-independent vector representation for complex region boundary surfaces that contain non-manifold structures and non-differentiable features such as creases and corners. The hierarchical decomposition organizes all resulting regions as leaf nodes of a binary tree, which we call an *SDF tree*. SDF trees support fast random access because signed distance functions can determine region memberships efficiently. We further introduce object embedding and instancing in our vector representation to compactly represent internal structures with a wide range of scales. Volumetric objects at smaller scales can be embedded into the regions of those at larger scales by linking together their respective SDF trees.

Such an explicit definition of individual volumetric components offers a few advantages. First, each region in a volumetric object can define its own color function. Changes made to one region do not affect others and the boundaries between regions remain sharp during magnification. Second, it allows a user to define a complex volumetric object by identifying representative components at each scale, modeling such components as standalone objects, and finally linking them together through embedding and/or instancing. It also greatly improves the reusability of existing volumetric components.

To facilitate volumetric object authoring and interactive editing, we have also provided both a scripting language and a GUI prototype. With the help of a recursively defined spatial indexing structure, we demonstrate that antialiased visualizations of arbitrary cross sections of a complex volumetric object can be generated in real time.

In Table 1 we compare our volumetric object representation against existing ones, such as voxel grids, procedural methods, vector solid textures (VST) [Wang et al. 2010], and diffusion surfaces (DSs) [Takayama et al. 2010], with respect to four criteria, including expressiveness, ease of editing, random-access performance and compactness. These criteria are consistent with the aforementioned desired properties of volumetric representations. In summary, our representation compares favorably in all of these criteria.

2 Related Work

Procedural noises are capable of generating solid textures as well as object boundaries [Perlin and Hoffert 1989] in a very compact form. However they are domain specific and it can be difficult to control the generation process precisely, and thus not sufficient to create generic volumetric objects with deterministic and semantically meaningful structures.

Implicit surfaces are an important geometric modeling tool. Ricci [1974] organizes multiple implicit surfaces into a CSG tree to construct complex solid geometries from simpler primitives. Frisken et al. [2000] develops adaptively-sampled distance fields (ADFs) which encode an SDF using an adaptive space partition scheme, such as an octree, so that geometric features with varying scales can be represented efficiently. Perry and Frisken [2006] extended ADFs to support multiple SDFs combined using Boolean operators within each octree cell to represent non-differentiable features such as corners. However, the above methods are designed for representing solids with a single region or multiple two-colorable regions. They are unable to represent objects with non-manifold structures. Differences among CSG, ADFs and our method will be further elaborated in Section 3.1.3.

Example-based methods allow the user to generate a solid texture that visually resembles one or more smaller exemplars [Kopf et al. 2007; Dong et al. 2008]. These methods are typically built upon the Markov Random Field assumption, which limits the result to patterns with relatively short-range correlations. Of particular interest among these methods is the multiscale texture synthesis algorithm developed in [Han et al. 2008], where an exemplar-graph is introduced to link together multiple 2D texture exemplars at different scales. The exemplar-graph is, however, neither volumetric nor resolution-independent. Takayama et al. [2008] provide a method for volumetric modeling by pasting solid texture exemplars repeatedly within a solid object with respect to a spatially-varying tensor field. Although their method grants the user with considerable controllability, the voxel-based nature makes it difficult to describe the global structures in an object in a resolution independent way. Wang et al. [2010] propose a resolution-independent representation for solid textures by decomposing a texture into individual regions whose boundaries correspond to the original sharp features and are represented by an SDF. Their method can only model two-colorable regions due to the use of a single SDF. In addition, the feature resolution is strictly limited by the SDF and region label pair structure, preventing their method from modeling general multiscale volumes.

Structured authoring of volumetric objects is pioneered by Cutler et al. [2002]. Their method allows the user to define layers in an object each of which can be assigned a different material. They

have also used multiple SDFs to define the layers. However, their method essentially organizes SDFs in a CSG tree using Boolean operators to create the boundary of a single solid object, while we organize the SDFs in a so-called SDF tree that allows for simultaneous modeling of multiple regions and a wide range of scales.

Takayama et al. [2010] present diffusion surfaces as a primitive to model featured colors in a volume. The color of any location is then computed by diffusion from nearby surfaces. Together with a sketch-based GUI, they have demonstrated that various volumetric objects can be created rapidly. By duplicating a diffusion surface into two slightly apart sheets, their method can also achieve the effect of blurred color transitions, which is an advantage over our method. However, diffusion surfaces are meshes, not a true vector representation, and it is also unclear how to add region-based high-frequency and multiscale details. In addition, their representation does not support fast random access as the color of every vertex on a cross section needs to be calculated in a separate rendering pass whenever the cross section is changed.

3 Volumetric Object Representation

A multiscale vector volume has an explicit definition of the individual volumetric components within. Here a *component* may refer to either a physical material such as the stone chips inside a terrazzo, or a semantically meaningful concept such as a cell of an organ. Since the internal structures and contents of the objects we intend to model span a wide range of scales, we have developed two methods to describe them. In the following, we first introduce a novel data structure called SDF tree that hierarchically divides a volumetric object into non-overlapping regions with arbitrarily complex boundary shapes and defines the attributes of each region independently. Then we discuss how to represent a complete multiscale object by linking together multiple vector volumetric objects.

3.1 SDF Trees

To represent a volumetric object or component with a single scale, an intuitive way is to decompose it into multiple semantically meaningful but simpler regions, and represent the region boundaries and contents separately. We decide to use signed distance functions (SDFs) to represent region boundaries because they are implicit functions suitable for representing volumetric regions.

An important limitation of a single signed distance function is that it can only divide a volume into two-colorable regions [Wang et al. 2010]. This limitation can be overcome by increasing the number of SDFs. For example, two SDFs can already divide a volume into four-colorable regions. Thus, multiple SDFs have the capability of representing non-manifold structures, where more than two regions are simultaneously adjacent to each other.

Given n overlapping SDFs, their zero isosurfaces divide the volume into multiple physical regions, each of which can be uniquely identified by the *signs* of the n SDFs. We organize the SDFs into a binary tree called an *SDF tree*. Each intermediate node of this tree is associated with an SDF and thus denoted as an *SDF node*. Since the zero isosurface of an SDF partitions a space into two half spaces, an SDF node can also have two children. Having another SDF node as a child means one of the half spaces is further partitioned by the zero isosurface of that SDF. A leaf node is also called a *region node* as it corresponds to one of the physical regions in the volumetric object. In practice, we may need to merge multiple physical regions to define a semantically meaningful logical region. In an SDF tree, this is achieved by mapping multiple leaf nodes to the same logical region.

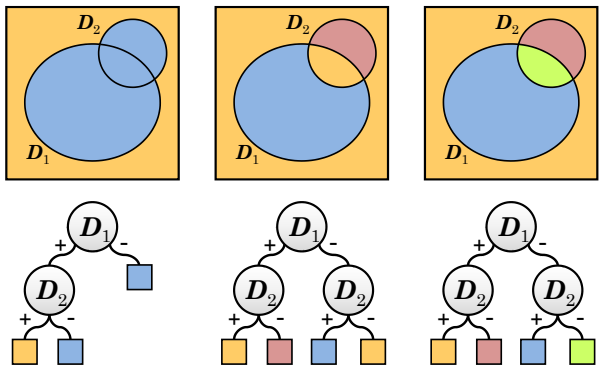

Figure 3: *By organizing two SDFs differently in an SDF tree, various region layouts can be achieved.*

Note that we allow the same SDF to appear in multiple SDF nodes as long as these nodes do not lie on the same tree path. As a result, the same set of SDFs can be organized differently in an SDF tree, giving rise to different region layouts. Figure 3 shows a 2D illustration. To determine the region membership of a 3D location p, one simply traverses the SDF tree by taking the child branch corresponding to the sign of the SDF associated with the current SDF node until a region node is reached.

3.1.1 SDF Nodes

SDF nodes need to be implemented to achieve both fast random access and compactness. Similar to [Wang et al. 2010], our SDFs are stored on voxel grids and evaluated using fast tricubic interpolation [Sigg and Hadwiger 2005] for efficient random access during rendering. Due to its voxel-based nature, the smallest feature details an SDF can represent are limited by the grid resolution. To compactly represent small or elongated regions, we enclose each SDF with an oriented bounding box (OBB) and allow this OBB to be scaled, rotated, and positioned anywhere in a volume. We use the term *SDF instance* to denote an SDF with an associated OBB whose affine transformation is represented by $\mathbf{M}$.

Sometimes the shape of a complex region can be assembled together from a number of parts, each of which is defined by its own signed distance function, such as the blood vessels in Figure 1 which consist of many similar branches. In such cases we can define a composite SDF in a tree node from a set of primitive SDF instances, $\{D_i, \mathbf{M}_i\}_{i=0}^{m}$ as follows:

$$\tilde{D}(p) = \min_i D_i(p \cdot \mathbf{M}_i), \quad p \in \mathbb{R}^3 \tag{1}$$

which essentially means that the shape defined by the composite SDF is the union of the shapes defined by the primitive SDF instances. Note that multiple instances can point to the same SDF.

3.1.2 Region Nodes

We define the color attribute inside each region as a 3D function. Once we know which region a 3D location p belongs to through SDF tree traversal, we can use the coordinates of p to evaluate the specific color function associated with that region to yield the final color at p. In our current implementation, the color function can be defined in one of the following three modes: solid color, solid textures, and radial basis functions. Note that a region can be declared as *empty*. Empty regions do not have color functions and can be used to define the exterior of a volumetric object.

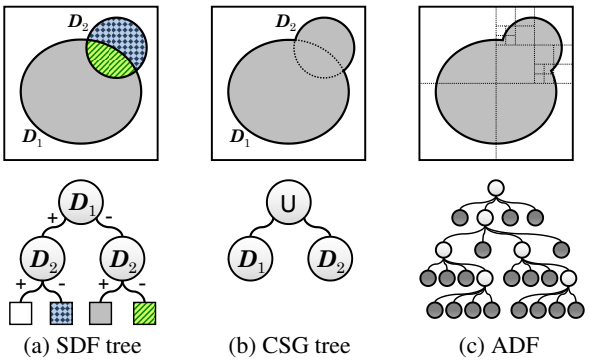

Figure 4: *The same shape (shaded) represented by (a) an SDF tree, (b) a CSG tree, and (c) an ADF respectively. Note that in addition to the shape boundary, our SDF tree also defines three internal regions.*

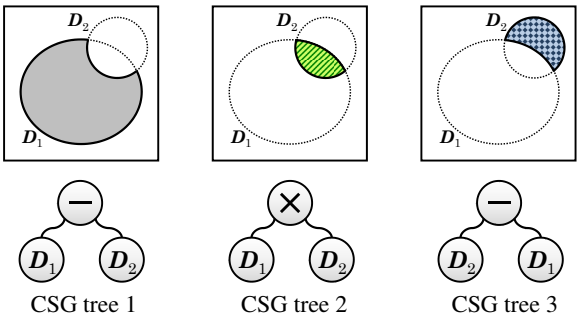

Figure 5: *A standalone CSG tree (or ADF) is needed to represent each region defined in Figure 4(a), while a single SDF tree is capable of representing all of them simultaneously.*

3.1.3 Comparison with Existing Methods

SDFs have been widely used in volumetric modeling. In this section we compare our vector representation with several SDF-related methods, especially those also involving a tree structure.

Constructive solid geometry. An SDF tree is a more powerful structure than constructive solid geometry (CSG) for implicit surfaces [Ricci 1974] even though the latter also has a hierarchical tree structure. In a CSG tree, individual implicit surfaces are associated with leaf nodes, which are merged bottom-up using Boolean operators to represent the single solid object associated with the root. In an SDF tree, individual implicit surfaces are associated with intermediate tree nodes, which recursively subdivide an inhomogeneous volume associated with the root into multiple simpler regions defined as the leaf nodes. Therefore, a CSG tree essentially defines *one single solid*, while an SDF tree is capable of defining *multiple mutually adjacent solid regions* which are not two-colorable. In fact, each region defined by an SDF tree can also be represented by a standalone CSG tree whose leaf nodes correspond to a subset of the signed distance functions in the SDF tree (Figure 5). Moreover, the time complexity for determining whether a point belongs to a certain region represented by a CSG tree is usually between $O(n)$ and $O(n^2)$ [Hable and Rossignac 2005], while in an SDF tree, the complexity is only $O(\log n)$, with n being the number of SDFs in both cases.

Adaptively-sampled distance fields. In a common implementation of ADFs, a volume is recursively subdivided by an octree where each cell contains a set of distance samples from which a partial implicit surface can be reconstructed. In order to represent sharp features more efficiently without excessive subdivision, Perry and Frisken [2006] allows an octree cell to contain more than one partial implicit surfaces whose Boolean combination results in one partial surface with sharp features. Nevertheless, similar to CSG, an ADF as a whole still defines one single solid that might consist of multiple two-colorable regions. It is unclear how ADFs can be extended to represent non-manifold structures, where *at least* three regions are mutually adjacent, which on the other hand can be conveniently represented with an SDF tree. Figure 4 shows the same shape represented by CSG, ADF, and our SDF tree respectively.

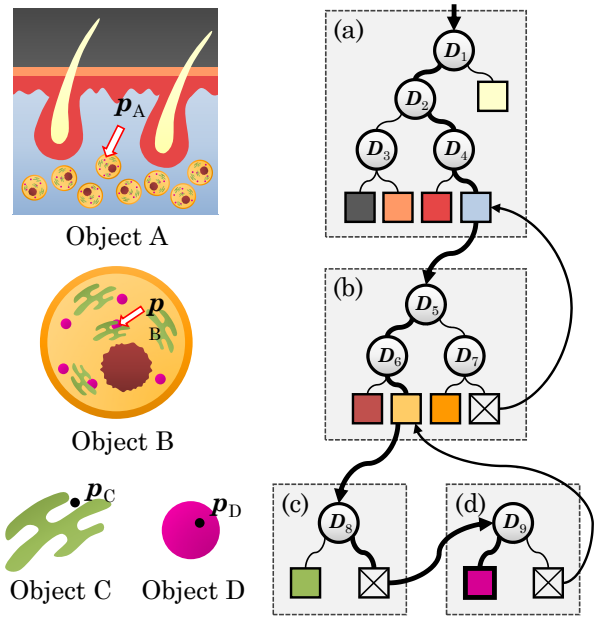

Figure 6: *Object embedding. Object A is constructed by recursive embedding of object B, C, and D. The SDF trees of the four objects are shown in (a)-(d) respectively. The bold arrows illustrate the traversal path to evaluate the color at location p_A, whose coordinates are transformed to p_B, p_C, p_D in each embedded instance's local frame respectively during the process. A crossed square indicates an empty region.*

Binary space partitioning. In the sense of region partitioning, an SDF tree can be considered as a generalization of a binary space partitioning (BSP) tree. A partition surface in a BSP tree is typically a plane while a partition surface in an SDF tree is the zero isosurface of an implicit function capable of defining an arbitrary shape.

3.2 Multiscale Embedding

Real-world volumetric objects often contain many volumetric components with dramatically different scales. Each component can also be considered as a standalone volumetric object and contain smaller-scale components recursively. We extend our volumetric representation to include *object embedding*, which allows one volumetric object to be embedded into a region of another object.

Without loss of generality, we assume that all vector volumetric objects are initially defined in a unit cuboid, $[0, 1]^3$. Similar to SDF instances, we define an *object instance* by associating a volumetric object with an affine transformation. To embed an instance of object A into region ℓ_B of another object B, we connect the SDF tree for A with the region node for ℓ_B in the SDF tree for B. When an SDF tree traversal reaches region ℓ_B, the 3D location p is first

transformed into the local frame for the instance of object A and then used to traverse the SDF tree of A to decide which region p belongs to. If however p falls into an empty region of A, its final color is determined by the color function of background region ℓ_B.

We can also embed multiple object instances into the same region. Since multiple instances may overlap in space, we use the embedding order to resolve the ambiguity in region membership. An illustration of object embedding and the corresponding traversal procedure is shown in Figure 6.

Many volumetric objects contain multiple similar but smaller-scale elements, such as cells in an organ. Modeling every individual element separately would be a tedious task and also requires a significant amount of storage. By embedding multiple instances that point to the same object but with different transformations, we considerably save both storage and modeling effort.

4 Volumetric Object Markup Language

We have discussed that SDFs can be used together to represent complex volumetric objects by partitioning the volume into regions that can be defined independently and recursively. We therefore introduce a descriptive language called *Volumetric Object Markup Language (VOML)* based on XML that allows a user to design the region configuration of a volumetric object.

Defining SDF Tree Structure. Listing 1 shows part of the VOML source code for the volumetric object in Figure 6. Basically, in a VOML source file, the region configuration is defined by organizing *SDF nodes* and *region nodes* into a binary tree. Each SDF node has two children nodes, corresponding to the subvolumes where the SDF is positive and negative, respectively. Either of the two nodes can map to a region by adding a region node as its child, or can be further subdivided by adding another SDF node as its child. A region node must be a leaf node in the tree and cannot have any children nodes.

```
<OBJECT name="skin">
  <SDF name="D1" file="1.sdf">
    <POSITIVE>
      <SDF name="D2" file="2.sdf">
        <POSITIVE>
          <SDF name="D3" file="3.sdf">
            <POSITIVE region="empty"/>
            <NEGATIVE region="epidermis"/>
        <NEGATIVE>
          <SDF name="D4" file="4.sdf">
            <POSITIVE region="dermis"/>
            <NEGATIVE region="hypodermis"/>
    <NEGATIVE region="hair"/>
```

Listing 1: *VOML code for Object A in Figure 6. The end-tags are omitted to save space.*

Once a volumetric object has been loaded from a file, a parser analyzes the region configuration defined in VOML and generates all necessary shader resources for rendering.

Region Content. Region content can be directly defined in the VOML source. For example, the following code defines two regions, filled by a solid color and a solid texture respectively.

```
<REGION name="color_reg" rgb="1 0.8 0.5"/>
<REGION name="texture_reg" bitmap="dermis_solid.png"/>
```

Regions that contain instances of other volumetric objects can be defined as:

```
<REGION name="dermis" rgb="0.8 0.3 0.3">
  <EMBEDDED object="fat_cell" instances="fat.txt"/>
  <EMBEDDED object="arteriole"/>
  <EMBEDDED object="venule"/>
</REGION>
```

The "instances" property points to an external text file containing the transformations of each instance. More details about VOML can be found in the supplemental materials.

5 Content Creation

Since our multiscale vector representation decomposes a volumetric object in a semantically meaningful way, this decomposition provides a natural guidance to the workflow one needs to follow when creating a digital volumetric object in our representation. The content creation process can be divided into a *planning* stage and a *creation* stage. During the planning stage, a user needs to rely on his/her prior knowledge of the object to be modeled to conceptually decompose the object in a hierarchical way. The following questions need to be answered.

- How many scales does the object have? Which representative components does the object have at each scale?
- What kind of features and regions does each component have? How are these regions organized? What is inside each region?

During the creation stage, the user needs to actually create the "building-blocks" of an object, such as SDF instances and region definitions, and then assemble them together into linked SDF trees.

In order to create SDFs for *highly structured* region boundaries, we have built a tool to convert a polygonal mesh to an SDF [Jones 1995]. To facilitate a continuous workflow, we have integrated this tool as a plug-in for the open-source 3D modeling software, Blender. In addition to SDFs converted from meshes, we have also implemented several procedural algorithms for efficiently synthesizing *stochastic* SDFs [Kopf et al. 2007] or adding randomness to existing SDFs [Perlin and Hoffert 1989].

As for region contents, there exist three options:

1. The user can assign a solid color, a solid texture, or a set of color radial basis functions to fill the region.
2. A region can be further divided by replacing the original region node in the SDF tree by a subtree containing more regions and SDF nodes.
3. A region can contain instances of other volumetric components. For example, the hypodermis region of skin contains many fat cells. A user could embed a volumetric object for the fat cell into the skin's hypodermis by object instancing.

For relatively simple volumetric objects, the user can construct SDF trees and perform object instancing by directly editing a VOML file. For more complex objects, we provide a GUI that allows the user to organize an intricate SDF tree by drag-and-dropping different nodes intuitively on the screen.

There exist various techniques to generate a spatial distribution of instances as well as the affine transformations associated with the instances. We have adopted three strategies for different scenarios: physical simulation for densely packed objects, adaptive blue-noise sampling [Wei 2008] for elements whose density and local properties vary according to a user-provided control map, and interactive placement of individual instances via the GUI we provide.

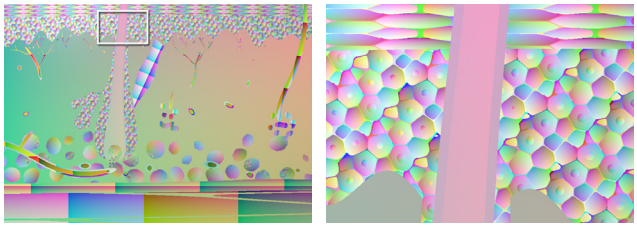

Figure 7: *The color in the above images represents the 3D coordinates of each pixel in the local frame of the deepest embedded object instance.*

Because our vector representation clearly defines the individual volumetric components of an object and organizes them into a hierarchical structure, the user is able to choose a highly flexible and iterative workflow during content creation. For example, a user could follow a top-down approach by first defining large-scale regions and then adding more details by subdivision or object embedding, or a bottom-up approach by first creating all low-level components and then organizing them into a complete multiscale object. At any time during the process, the user can also iteratively modify a single component without affecting other parts of the object. A live demo is given in the supplemental video.

6 Rendering

Efficient rendering is critical for interactive content creation. Furthermore, rendering arbitrary cross sections of a volumetric object is an important way to visualize its internal content. Given a 3D location p, the rendering process traverses the SDF tree from the root until it reaches a region node. The color of p is then decided by evaluating the color function defined in that region. Appendix A shows simplified pseudo code for this process.

The coordinates p is transformed whenever the traversal enters a new embedded object (i.e. another SDF tree associated with an affine transformation), so that once the final region is reached, p may have been transformed multiple times, as shown in Figure 7.

Spatial Indexing for Embedded Objects. Embedded SDFs and objects have compact oriented bounding boxes to define their spatial extent. But these bounding boxes negatively impact random access performance since an additional step is required to decide which SDF/object a position belongs to. To overcome this problem, we build a spatial indexing structure for each region that contains embedded volumetric components or SDFs. The indexing structure simply divides the region into a uniform grid and records pointers to the objects and SDFs that overlap with each grid cell. Note that the indexing structure itself can be embedded and instanced along with its parent object, effectively making the entire volumetric object adaptively indexed.

Texture Packing. In our representation, a volumetric object may use many 3D textures. However, 3D textures cannot be dynamically indexed in the current graphics hardware. We have to pack all 3D textures of the same type into one large texture (similar to a texture "atlas" in 3D). Finding an efficient packing for a given set of textures can be considered as a 3D singular bin-packing problem where a set of boxes with different sizes are arranged in a way to minimize wasted space. Finding an optimal packing for a non-trivial input is NP-hard. We use a heuristic algorithm [Corcoran and Wainwright 1992] which strikes a balance between speed and packing efficiency.

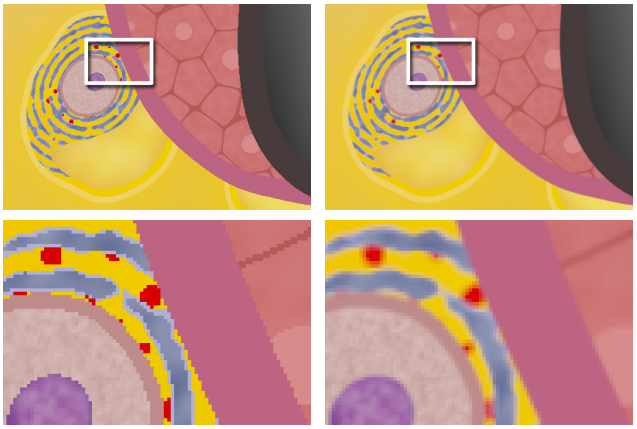

Figure 8: *Rendering without (left) and with (right) antialiasing.*

Antialiasing. Antialiasing is crucial for high-quality rendering of vector graphics. We have implemented antialiasing as a post-processing step in a similar way as in many deferred shading systems (e.g. [Shishkovtsov 2005]). In the first rendering pass, the shader writes the *minimum signed distance* and the *region ID* as well as the evaluated color of each pixel to the render targets. In the second pass, the *region ID* of each pixel is compared with those of its neighbor pixels to decide if this pixel is on a region boundary. If it is, the pixel shader calculates a weighted average of the colors in the 3×3 neighborhood. Otherwise the pixel color is left unchanged to avoid undesired blurring of textures within a region. This method is efficient and provides reasonably good quality (see Figure 8).

7 Results and Discussions

We have generated several volumetric objects of different types using our method, as shown in Figure 1 and 10. The overall authoring time ranges from a couple of minutes to several hours, most of which is spent on modeling the polygonal meshes for the SDFs. In addition to creating SDFs manually, we can also utilize existing data, such as scanned medical data. In Figure 10 (d), a medical scan of a human brain is segmented into regions, from which we can build an SDF tree with multiple SDFs automatically. Once all the SDFs and instancing transformations are known, it usually takes only minutes to design and construct a corresponding SDF tree.

We have tested our method on an NVidia GeForce GTX460. For the objects in this paper the rendering performances within our GUI prototype ranges from 60 to over 500 fps in an 800 by 600 window with antialiasing. In addition, we have also implemented a naïve offline renderer featuring ray-marching for translucent regions. The rendering of Figure 1 takes about two minutes at a resolution of 4096^2 with 4×4 supersampling.

Comparison with vector solid textures. Our work was partially inspired by vector solid textures [Wang et al. 2010]. Nevertheless, there exist significant differences between SDF trees and VSTs.

- SDF trees can represent non-manifold features, whereas VST cannot. The regions in a VST must be two-colorable due to the use of a single SDF. Although there is a two-scale result (Figure 9(g) in [Wang et al. 2010]), which has non-manifold features, the regions in each scale is still two-colorable and it cannot be applied to more general objects.

- The feature resolution in a VST is limited by the resolution of its SDF and region label pairs. On the other hand,

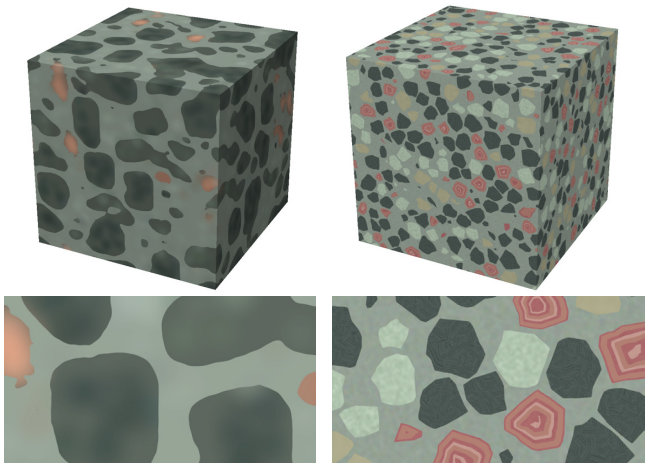

Figure 9: *Comparison between VST (left) and our vector representation (right). For a similar texture-like volume, VST consumes 1.17 MB while our representation delivers richer details but only needs 0.7 MB. Both are rendered at approx. 200 fps with antialiasing.*

our linked SDF trees can represent unlimited details at small scales through object instancing and embedding while maintaining very compact storage.

- VST uses a regularly-sampled region label-pair structure to identify individual regions. When there exist n SDFs, each label-pair would become a 2^n-sized label array, causing prohibitive storage overhead. In contrast, an SDF tree completely eliminates explicit region labeling and encodes region labels implicitly into different SDF tree paths.

Figure 9 demonstrates two texture-like volumetric objects created by VST and our method respectively. In comparison, our method provides richer details while consuming less storage and rendering more efficiently.

Limitations Currently each instance is only associated with an affine transformation and our representation does not support nonlinear transformations of object instances. In addition to transformations, it should be useful in certain scenarios to add other per-instance data such as color variations. Supporting blurred region boundaries would be challenging since the neighborhood information of a region is not directly accessible.

8 Conclusions

In this paper, we have introduced a compact multiscale vector representation for volumetric objects with complex internal structures spanning a wide range of scales. Our representation relies on a binary SDF tree to hierarchically decompose an entire volumetric object into multiple regions while maximally aligning sharp features with region boundaries. By linking together multiple SDF trees, a complex volumetric object can be constructed from simpler volumetric objects at different scales. SDF and object instancing further utilize the substantial repetition of similar elements or volumetric features in an object and reduce the storage cost significantly. To facilitate authoring and editing, we have also provided both a scripting language and a GUI prototype. Results show that volumetric objects with highly complex structures and resolution-independent features can be easily created from scratch using our method. These vector volumetric objects have compact storage and, with the help of a recursively defined spatial index, can be rendered in real time with antialiasing.

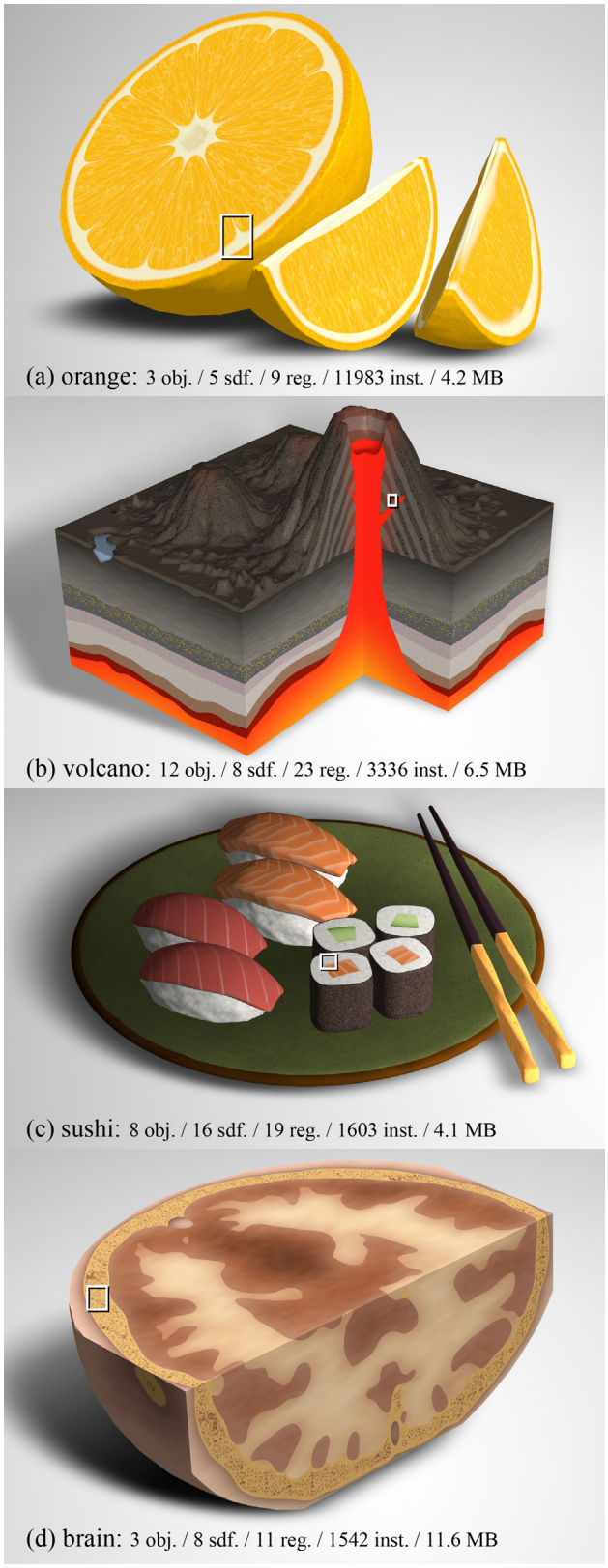

Figure 10: *Various volumetric objects created using our method. The first three are generated from scratch, while the last one is based on an existing segmentation result of captured medical data. The respective numbers of SDF cells, SDFs, regions, and instances, as well as storage are shown below each result.*

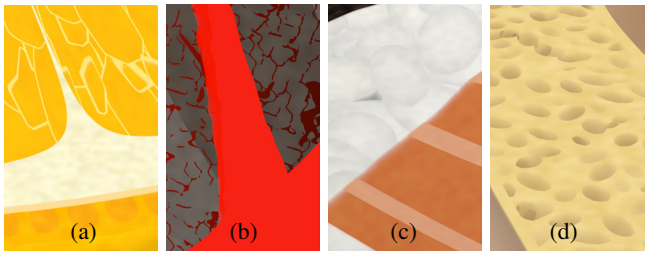

Figure 11: *Zoomed views of the volumetric objects in Figure 10. Note the ubiquitous non-manifold structures.*

In our current implementation, a polygonal mesh is used to define the outmost boundary surface where the color of a volumetric object should be evaluated. An intriguing idea is to get rid of any mesh and decide the evaluation locations by directly ray-casting [Hadwiger et al. 2005] the implicit region boundaries in our representation.

Acknowledgments

We would like to thank all the reviewers for their valuable feedbacks. Yugang Liu provided the brain model in Figure 10(d). Yizhou Yu was partially supported by NSF (IIS 09-14631) and The University of Hong Kong. Kun Zhou was partially supported by NSFC (No. 60825201) and the 973 program of China (No. 2009CB320801).

References

CORCORAN, A. L., AND WAINWRIGHT, R. L. 1992. A genetic algorithm for packing in three dimensions. In *Proceedings of ACM/SIGAPP Symposium on Applied Computing*, 1021–1030.

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

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

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.

HABLE, J., AND ROSSIGNAC, J. 2005. Blister: GPU-based rendering of Boolean combinations of free-form triangulated shapes. *ACM Trans. Graph. 24*, 3, 1024–1031.

HADWIGER, M., SIGG, C., SCHARSACH, H., BÜHLER, K., AND GROSS, M. 2005. Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum 24*, 3, 303–312.

HAN, C., RISSER, E., RAMAMOORTHI, R., AND GRINSPUN, E. 2008. Multiscale texture synthesis. *ACM Trans. Graph. 27*, 3, 51:1–51:8.

JONES, M. W. 1995. 3D distance from a point to a triangle. Tech. Rep. CSR-5-95, University of Wales Swansea.

KOPF, J., FU, C.-W., COHEN-OR, D., DEUSSEN, O., LISCHINSKI, D., AND WONG, T.-T. 2007. Solid texture synthesis from 2D exemplars. *ACM Trans. Graph. 26*, 3, 2:1–2:9.

PERLIN, K., AND HOFFERT, E. M. 1989. Hypertexture. In *Proceedings of SIGGRAPH '89*, 253–262.

PERRY, R. N., AND FRISKEN, S. F. 2006. Method for generating a two-dimensional distance field within a cell associated with a corner of a two-dimensional object. *US Patent No. US 7,034,830*.

RICCI, A. 1974. A constructive geometry for computer graphics. *Computer Journal 16*, 2, 157–160.

SHISHKOVTSOV, O. 2005. Deferred shading in S.T.A.L.K.E.R. In *GPU Gems 2*. ch. 9, 143–166.

SIGG, C., AND HADWIGER, M. 2005. Fast third-order texture filtering. In *GPU Gems 2*. ch. 20, 313–329.

TAKAYAMA, K., OKABE, M., IJIRI, T., AND IGARASHI, T. 2008. Lapped solid textures: filling a model with anisotropic textures. *ACM Trans. Graph. 27*, 5, 53:1–53:9.

TAKAYAMA, K., SORKINE, O., NEALEN, A., AND IGARASHI, T. 2010. Volumetric modeling with diffusion surfaces. *ACM Trans. Graph. 29*, 5, 180:1–180:8.

WANG, L., ZHOU, K., YU, Y., AND GUO, B. 2010. Vector solid textures. *ACM Trans. Graph. 29*, 4, 86:1–86:8.

WEI, L.-Y. 2008. Parallel Poisson disk sampling. *ACM Trans. Graph. 27*, 3, 20:1–20:9.

A Pseudo-HLSL Code for Rendering

```
float4 SDFTreeTraversal(float3 p)
{
   int      i     = 0;
   Instance inst  = g_Instances[i]; // Object instance
   SDFNode  node  = g_SDFTree[inst.rootNodeID];
   float4   color = float4(0,0,0,0);
   float3   oldP  = p;

   while (inst.isEmpty == false) {
      if (IsCoordValid(p) == false) {
         // p not inside current instance, try next
         inst = g_Instances[++i];
         node = g_SDFTree[inst.rootNodeID];
         p = Transform(oldP, inst.transformID);
         continue;
      }
      if (g_SDF[node.sdfID].Sample(p) > 0)
         node = g_SDFTree[node.positiveChild];
      else
         node = g_SDFTree[node.negativeChild];

      if (node.isLeaf) {
         Region region = g_Regions[node.regionID];
         if (region.type == REGTYPE_NORMAL)
            return EvaluateColor(region, p);
         else if (region.type == REGTYPE_EMBEDDED) {
            // Region contains embedded objects
            color = EvaluateColor(region, p);
            i = region.firstInstID;
            inst = g_Instances[i];
            node = g_SDFTree[inst.rootNodeID];
            oldP = p; // Cache p in current region
            p = Transform(p, inst.transformID);
         }
         else if (region.type == REGTYPE_EMPTY) {
            // Empty region, try next instance
            inst = g_Instances[++i];
            node = g_SDFTree[inst.rootNodeID];
            p = Transform(oldP, inst.transformID);
         }
      } // end if (node.isLeaf)
   } // end while
   return color;
}
```