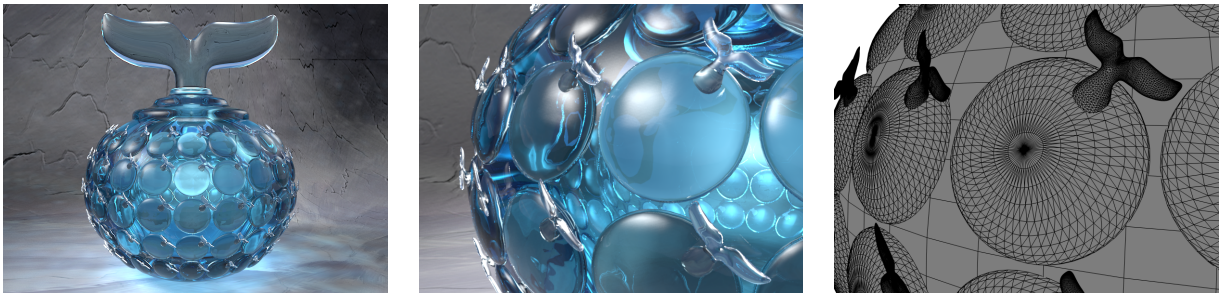
Shell Maps

Serban D. Porumbescu* Brian Budge* Louis Feng* Kenneth I. Joy*

Institute for Data Analysis and Visualization
Computer Science Department
University of California, Davis



(a) (b) (c)

Figure 1: Inspired by the pâte-de-verre techniques of glass sculpting, the whale's-tail vase is modeled by scaling the tail and placing it onto a glass "bead." The geometry of the tail and bead is textured around the base, and the whale's tail is used for the cap of the vessel. The full vase is shown in (a), a zoomed image in (b), and the wireframe detail of the model in (c).

Abstract

A shell map is a bijective mapping between shell space and texture space that can be used to generate small-scale features on surfaces using a variety of modeling techniques. The method is based upon the generation of an offset surface and the construction of a tetrahedral mesh that fills the space between the base surface and its offset. By identifying a corresponding tetrahedral mesh in texture space, the shell map can be implemented through a straightforward barycentric-coordinate map between corresponding tetrahedra. The generality of shell maps allows texture space to contain geometric objects, procedural volume textures, scalar fields, or other shell-mapped objects.

Keywords: texture mapping, displacement mapping, ray tracing, volumetric textures, near surface parameterization

1 Introduction

Today, the realism required in computer graphics applications necessitates fine detail on many surfaces in a model. Detail is commonly represented by two-dimensional textures [Blinn and Newell 1976], bump mapping [Blinn 1978], or displacement mapping [Cook et al. 1987] applied to objects, polygonal meshes or higher-level primitives. These methods cannot represent the fine-scale surface geometry that is an integral component of the appearance of many real-world materials and objects. While most textured surface representations work well for objects of simple topology, they are not practical for representing objects with fine-scale geometric detail. It is necessary to address volumetric methods for textures, which require a three-dimensional parameterization.

Therefore, we think of "thick surfaces" and methods to apply three-dimensional texture or geometric detail at a fine scale on these surfaces (Figure 1). We consider the volume of space between two surfaces, develop functions that identify this region with three-dimensional texture space, and develop techniques to model three-dimensional detail.

This paper introduces shell maps, a method that combines texture and geometry methods to represent fine-scale complex surface detail - extending the concepts of volumetric textures and displacement maps. A shell map is a bijective (one-to-one) mapping between shell space and texture space that can be used to generate fine-scale features on surfaces, see Figure 2. Given a base surface S, we generate an offset surface S_o that has the same structure as S. Shell space is the area between these two surfaces. Utilizing the identical structures of S and S_o, we tile shell space with prisms, each of which has a corresponding prism in texture space. Breaking these prisms into tetrahedra, we generate a direct correspondence between tetrahedra in shell space and tetrahedra in texture space. The shell map is then defined by the barycentric coordinates of the corresponding tetrahedra.

These maps allow considerable flexibility in the types of objects that can be placed in texture space. Texture space can contain geometric objects, procedural volume textures, scalar fields, or other objects that have been shell mapped. The mapping is bijective, allowing the use of both feed-forward rendering applications and ray-tracing applications. We illustrate the use of this mapping in feed-forward applications utilizing geometry as generalized displacement maps, and in ray-tracing applications with procedural textures.

*e-mail:{sdporumbescu, bcbudge, zfeng, kijoy}@ucdavis.edu

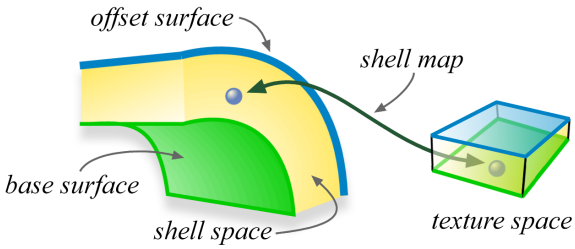


Figure 2: Shell space is the region between a base surface and an offset surface to the base. A shell map is a one-to-one function between texture space and shell space.

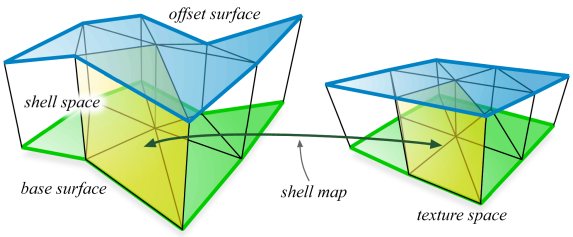


Figure 3: Prisms in shell space correspond to prisms in texture space.

Section 3 presents the construction of a shell map including offset surface generation, prism and tetrahedra correspondence, and barycentric coordinate mapping, all of which combine to allow the definition of the bijective map. Section 4 discusses modeling and rendering issues using the shell-mapping technique, and Section 6 details results of the use of the algorithm.

2 Related Work

Several research groups have focused on the problem of generating fine-scale detail on surfaces. Initial efforts to map planar textures [Blinn and Newell 1976; Blinn 1978] are now a standard component of all graphics processors and systems. Unfortunately, these methods do not map three-dimensional fine detail to surfaces, and recently the focus has shifted to volumetric methods.

Most volumetric methods define a shell volume about a surface and use mappings to transfer from texture space to shell space. Kajiya and Kay [Kajiya and Kay 1989] were the first to introduce volumetric textures. Their methods utilize volumetric data sampled on a regular grid, and trace rays through a shell volume on a surface. Rays that intersect the shell are transformed to texture space and traced through the sampled data grid. Neyret [Neyret 1998],who greatly expanded the types of objects used for volumetric textures, used a similar strategy. Once the rays are transformed to texture space, the problem reduces to an isosurface generation problem in volume rendering [Levoy 1988]. Perlin and Hoffert's hypertextures [Perlin and Hoffert 1989] are rendered similarly.

Cabral *et al.* [Cabral et al. 1994] and Westermann and Ertl [Westermann and Ertl 1998] have developed slice-based methods for volume rendering applications, and this method has been adapted by several researchers for volume texture methods in interactive applications. Meyer and Neyret [Meyer and Neyret 1998] utilized this approach to improve the speed of the ray-tracing approach of [Neyret 1998] and to utilize graphics hardware. Peng [Peng 2004] and Peng *et al.* [Peng et al. 2004] used the slice-based methods in an interactive system that can be used for modeling both surface models and volumetric textures.

Displacement mapping was introduced by Cook *et al.* [Cook et al. 1987] in their description of the REYES image architecture. This technique explicitly models surface displacement by height fields. Smits *et al.* [Smits et al. 2000] describe methods to make displacement mapping reasonable for ray tracing. Wang *et al.* [Wang et al. 2004] describe generalized displacement maps. Their method is a powerful precomputation method for adding displacement geometry and global illumination features onto surfaces.

Chen *et al.* [Chen et al. 2004] model an object as a shell layer and a homogeneous inner core. The computation of surface radiance from the shell layer is accomplished through shell texture functions (STF) which describe radiance fields based on precomputed fine-level light interactions. This enables their system to map fine-scale structures with advanced lighting features onto arbitrary surfaces.

We present a straightforward and powerful method for mapping three-dimensional regions containing textures or geometry into regions between surfaces and their offsets. The mapping is bijective which allows the use of applications that map shell-space points to texture space, and texture-space points to shell space. This method allows many object types to be placed directly into texture space, greatly expanding the detail that can be created on surface models.

3 Shell Maps

Shell maps provide a general technique to map a three-dimensional volume onto a surface. They provide a way to model a thick layer of "skin" containing complex topological details, and map these details to a surface. We refer to the original surface for which we construct a shell map as the *base surface* S. We assume that S is a triangulated mesh with texture coordinates assigned to each mesh vertex. *Shell space* is a bounded region between S and a constructed offset surface S_o. The shell map establishes a correspondence between points in *texture space* and points in *shell space*, as shown in Figure 2.

Triangles in S_o correspond one-to-one with triangles in S. By connecting the vertices of corresponding triangles, we form prisms in shell space. Each prism has a corresponding prism in texture space, generated by using the texture coordinates of the vertices of the shell-space prism (Figure 3). To achieve a consistent parameterization, we split the prisms into tetrahedra, using a technique that preserves mesh continuity. The mapping is established through the barycentric coordinates of corresponding tetrahedra.

Shell maps are very flexible. Texture space can contain geometric objects, procedural volume textures, scalar fields, or other shell mapped objects. Once the correspondence is established, a variety of existing rendering techniques can be used to render surfaces with shell maps. Shell mapped

objects can be used in a global illumination algorithms such as photon mapping [Jensen 2001] and can naturally interact with other objects by casting shadows and caustics. Hardware accelerated rendering is also possible because geometry in texture space can be transformed into object space and rendered directly.

3.1 Offset Surface Generation.

To enable the correspondence between shell space and texture space, we first generate an offset surface S_o with the following properties:

- S and S_o must have the same number of triangles and the same mesh connectivity. Each triangle $\mathcal{T} \in S$ must be associated with a unique triangle $\mathcal{T}_o \in S_o$. Each vertex $\mathbf{v} \in S$ must be associated with a unique vertex $\mathbf{v}_o \in S_o$.
- S_o should have no self intersections and should not intersect S.

For a few objects, *e.g.*, spheres, ellipsoids, and cylinders, such offsets are trivial to generate. However, for general surfaces represented by triangle meshes, an offset surface is difficult to construct. Offset surface generation approaches can be found in the mesh generation field [Bernd and Plassmann 2000], in medical imaging [Yezzi and Prince 2002], and in computer-aided geometric design [Farin 1998]. The methods of Cohen *et al.* [Cohen et al. 1996], who generate an approximate offset surface for simplification of complex polygonal models, and Peng *et al.* [Peng et al. 2004], who utilize a point-to-surface distance function to generate displacements, generate an offset surface that satisfies the necessary properties for shell maps.

We employ the envelope generation algorithm of Cohen, *et al.* [Cohen et al. 1996] as it provides a nice balance between ease of implementation and generality. Their method preserves the topology of the generated offset surface, while maintaining a correspondence between vertices on the original surface and vertices on the offset.

First, S_o is generated by duplicating S. This establishes the basic correspondence between triangles and vertices in the two meshes. The vertices of S_o are assigned the same texture coordinates as the corresponding vertices of S. Each vertex $\mathbf{v}$ of S is assigned a "direction" vector $\vec{d}$, which is a linear combination of the normal vectors of the triangles of which $\mathbf{v}$ is a vertex. S_o is then displaced away from S by iteratively moving vertices $\mathbf{v}_o$ of S_o along the direction vectors associated with the corresponding vertex $\mathbf{v}$ of S. An attempt is first made to displace a vertex the full offset distance specified by the user. Using an octree, the algorithm efficiently detects whether the vertex move is valid or if it leads to self intersection. If the vertex displacement leads to self-intersection, the algorithm rejects the displacement and begins an adaptive stepping procedure along the direction vector at a fraction of the user-specified offset distance. This procedure terminates with a final vertex offset that ensures no self-intersection and no degenerate triangles. See Cohen *et al.* [Cohen et al. 1996] for additional details.

For fine-surface detail a "constant-distance" offset surface is best to keep the detail consistent over the surface. Cohen's algorithm generates an acceptable offset except in regions of extreme concavity, where the offset surface may deviate from the true offset. Arbitrary offset surfaces can be manually constructed and used to generate the shell map as long as they satisfy the two properties above. If detail distortion is desired, the offset surface can be manually constructed and may substantially deviate from the true surface offset. In this way, the offset surface S_o can be used as a modeling tool.

Note on offset self intersection. Preventing self intersection of the offset surface is necessary to create a bijective mapping (see Section 3.4). However, for applications with overlapping (*e.g.*, fur or hair) or self intersecting structures a bijective mapping is not always necessary. To guarantee that unintuitive overlapping geometry is not generated when ray tracing isosurfaces, bijectivity, and therefore non-overlapping adjacent tetrahedra should remain a constraint. It may be desirable to relax the constraint for non-adjacent tetrahedra (the ends of a horseshoe for example), to allow a natural "growing together" of surfaces. If overlapping surfaces are not a concern, naturally the bijectivity constraint can be lifted.

3.2 Prisms and Tetrahedra Construction

Given an offset surface S_o to S, we generate prisms in shell space by connecting the vertices of triangles in S with corresponding triangles in S_o. Each prism P is bounded by two corresponding triangles and three (generally non-planar) quadrilaterals.

Each P has a corresponding prism P_t in the (u, v, w) coordinate system of texture space. Assume that P is defined by the two triangles $\mathcal{T}$ and $\mathcal{T}_o$, where $\mathcal{T}$ has vertices $\mathbf{v}_1$, $\mathbf{v}_2$, and $\mathbf{v}_3$, and $\mathcal{T}_o$ has vertices $\mathbf{v}_{o,1}$, $\mathbf{v}_{o,2}$, and $\mathbf{v}_{o,3}$. Then P_t is defined by two texture space triangles $\mathcal{T}_t$ and $\mathcal{T}_{o,t}$, where $\mathcal{T}_t$ has vertices

$$(u_1, v_1, 0), (u_2, v_2, 0), \text{ and } (u_3, v_3, 0)$$

and $\mathcal{T}_{o,t}$ has vertices

$$(u_1, v_1, k), (u_2, v_2, k), \text{ and } (u_3, v_3, k),$$

where the (u_i, v_i) coordinates are the two-dimensional texture coordinates of the respective vertices of $\mathcal{T}$ in the base surface. The height, k, of texture space is computed to preserve scale between texture space and shell space as

$$k = a_t / a * h$$

where a_t and a are the average of the triangle edge lengths of the base surface in texture space and shell space respectively. The height, h, is the maximum offset height in shell space.

Texture-space prisms are right triangular prisms, but shell-space prisms are generally non-convex (They can contain non-planar quadrilateral faces). This lack of prism convexity precludes the use of generalized barycentric coordinate techniques [Warren et al. 2003] which require convex polyhedra as input. To create a robust mapping (Section 3.4) we split the prisms into three tetrahedra by triangulating the three quadrilateral faces, see Figure 4 (Note that different triangulations may lead to slightly different shell maps). By splitting corresponding prisms P and P_t in the same way, we establish a correspondence between tetrahedra in shell space, and tetrahedra in texture space, where each tetrahedron T in shell space has a unique corresponding tetrahedron T_t in texture space.

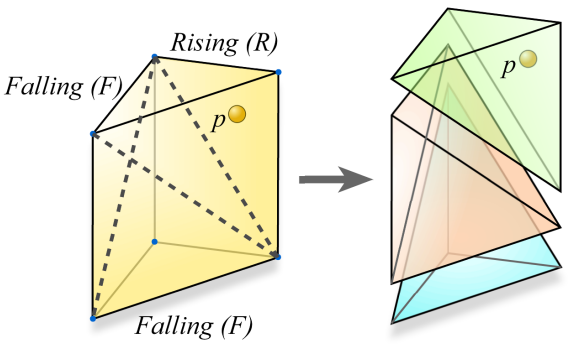


Figure 4: Prisms are split into three tetrahedra. Prisms can be split in six ways, depending on the direction of the triangulation of the quadrilateral faces. The six splits can be characterized with labels (FRR, RFR, RRF, RFF, FRF, and FFR) depending on the triangulation of the quadrilateral faces.

3.3 Maintaining Continuous Tetrahedral Meshes

To generate a continuous bijective mapping between shell space and texture space, it is important to define and maintain a consistent tetrahedral mesh. Each prism P can be split into three tetrahedra in six ways. Consider two neighboring prisms P_1 and P_2 that share a non-planar quadrilateral face F. It is possible to naively split P_1 and P_2 separately into tetrahedra such that they have different triangulations for F. This inconsistency must be avoided, as it will introduce a discontinuity into the mapping.

To construct a consistent tetrahedral mesh from the prism mesh, we use an algorithm similar to the rippling algorithm described by Erleben and Dohlmann [Erleben and Dohlmann 2004]. We label the edge that splits a quadrilateral of a prism as either rising (R) or falling (F), see Figure 4. There exist six possible configurations of the edges of a prism such that the prism can be tessellated into three tetrahedra: FRR, RFR, RRF, RFF, FRF, and FFR, each configuration having two Rs and one F, or two Fs and one R. Configurations FFF and RRR do not lead to valid tetrahedralizations. Consistent adjacent prisms should have different labels on the common quadrilateral. Tetrahedra patterns are assigned per prism by traversing the base triangle in a counter clockwise direction. To split a prism, we assign a base surface triangle a random pattern (*e.g.*, FFR) if none of its neighbors have been previously assigned a pattern. If a neighboring edge has already been assigned we choose the opposite value.

If an inconsistency arises during the split of a prism P, the conflict is resolved by flipping the triangulation of a quadrilateral in one of its neighbors. Two inconsistencies can be generated in this process:

- two adjacent prisms have identical labels on a shared quadrilateral face, or
- a prism is given a configuration with three Fs or three Rs, which implies that the prism cannot be split into three tetrahedra.

Given an inconsistent prism P, we resolve the inconsistency by flipping the triangulation of an adjacent quadrilateral in one of its neighbors P_1. If the change introduces a new inconsistency for P_1, a new neighbor P_2 of P_1, $P_2 \neq P_1$, is randomly chosen and the triangulation of the adjacent quadrilateral of P_2 is flipped. This "rippling" propagates the inconsistency away until all prisms are consistent. See Figure 5 for a pictorial description of this process. We use a depth-first search through adjacent prisms until all inconsistencies are resolved. In practice, the procedure converges rapidly, as there are six possible configurations for each prism. When a conflict arises only a few steps are needed to resolve the inconsistency. We do not have a proof that this rippling approach to resolve inconsistencies always converges or that there is always a solution. However, we have never encountered a case where the algorithm did not converge quickly.

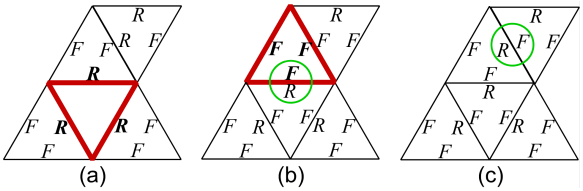


Figure 5: A two-dimensional description of the rippling algorithm. The red triangle represents a prism with inconsistencies in (a). No consistent label can be applied to this triangle. The inconsistencies are resolved by assigning a label (RFF) and flipping the label on an adjacent edge. This creates an inconsistency in the red triangle in (b). Flipping another edge in (c) resolves the inconsistencies.

Implementation Note. Once all prisms have been assigned tetrahedra patterns, actual tetrahedra can be extracted by using a lookup table for each of the six patterns. Our implementation uses a simpler approach based on the four patterns (FF, FR, RR, and RF) that can arise when looking at two edges of any of the six tetrahedra split cases. We construct each of the three tetrahedra in a prism by iterating counter clockwise around the base of the prism and looking at the current edge tag (which is either F or R) and the next counter clockwise edge tag (which is either F or R). This allows us to easily choose the four vertices of our tetrahedron for each of the three edges of the base triangle. We found this approach easier and less error prone than constructing the tetrahedralizations for each of the six cases separately.

3.4 Shell Mapping Function

Given a tetrahedron T in shell space, and its corresponding tetrahedron T_t in texture space, any point $\mathbf{p}$ in T can be associated with a unique point $\mathbf{p}_t$ in T_t using barycentric coordinates.[1] That is, if $B(T, \mathbf{p})$ defines the barycentric coordinates of $\mathbf{p}$ in T, and $\phi(T, \alpha)$ defines the point in T with barycentric coordinates α, then

$$\mathbf{p}_t = \phi(T_t, B(T, \mathbf{p})), \text{ and}$$
$$\mathbf{p} = \phi(T, B(T_t, \mathbf{p}_t))$$

establishing the bijective mapping between shell space and texture space.

[1] If $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$, and $\mathbf{v}_4$ are the vertices of a tetrahedron T, and $\mathbf{p}$ is a point in T, then the barycentric coordinates $(\alpha_1, \alpha_2, \alpha_3, \alpha_4)$ of $\mathbf{p}$ in T are defined such that $\mathbf{p} = \alpha_1\mathbf{v}_1 + \alpha_2\mathbf{v}_2 + \alpha_3\mathbf{v}_3 + \alpha_4\mathbf{v}_4$, where $\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 = 1$.

Once the correspondence is established, we map points from shell space to texture space using a combination of point-in-tetrahedron queries and barycentric coordinates. Given a point $\mathbf{p}$ in shell space, we determine the prism P and tetrahedron T that contains $\mathbf{p}$. If α is the barycentric coordinate of $\mathbf{p}$ in T, then the texture-space point $\mathbf{p}_t$ that corresponds to $\mathbf{p}$ is given by $\mathbf{p}_t = \phi(T_t, \alpha)$, where T_t is the tetrahedron corresponding to T in texture space.

Similarly, we map points from texture space into shell space using a combination of point location queries and barycentric coordinates. However, since prisms in texture space are right triangular prisms, we cast this problem to a point location query in the plane by projecting $\mathbf{p}_t$ to the plane $w = 0$ in texture space, and search against the base of the prisms in two-dimensions.

Algorithms exist that can solve the point-in-triangle search in $O(\log N)$ time using $O(N)$ space [Kirkpatrick 1983; Sarnak and Tarjan 1986]. To reduce memory requirements, we use the point location algorithm of Guibas and Stolfi [Guibas and Stolfi 1985] with the modifications presented by Brown and Faigle [1997]. Brown and Faigle's modifications guarantee algorithm termination and provide a heuristic for worst case look ups of $O(\sqrt{N})$. Using this method to locate the base triangle, we compute the barycentric coordinates for $\mathbf{p}_t$ with respect to the three tetrahedra associated with the located prism, and this identifies the tetrahedron that contains $\mathbf{p}_t$.

4 Modeling and Rendering with Shell Maps

Geometric objects such as triangle meshes, subdivision surfaces, or even other shell mapped objects can be placed in texture space and mapped directly into shell space. Since shell space is represented by a tetrahedral mesh, we use the shell map to access the geometry stored in corresponding texture-space tetrahedra, map this geometry to shell space, and render it using the graphics pipeline.

We can use shell-mapped models in the design of geometry textures. We first use the shell-mapping technique to generate a polygonal model. This new model is then inserted into texture space, and a new shell map can cause this polygonal detail to be mapped to a new surface.

Procedural and geometry textures can be rendered using ray tracing. In previous methods [Kajiya and Kay 1989; Neyret 1998; Perlin and Hoffert 1989; Ebert et al. 2003], rendering is accomplished by transforming world-space ray segments to texture space and by marching rays through texture space. In our method, rays intersect either the offset surface or the base surface. The intersection determines the tetrahedron the ray enters, along with the entrance and exit point. The ray is marched in shell space, and points are transformed to texture space for density calculations and ray-surface intersection calculations. In this way, we effectively trace a curved ray in texture space, see Figure 7. This allows interaction between the element of the procedural texture, and results in a higher-quality solution than those that use linear marching in texture space [Wang et al. 2004; Neyret 1998].

The triangles of the base and offset surfaces contain links to their associated prism. Once a tetrahedra has been intersected, neighboring tetrahedra can be quickly accessed as the ray marches through shell space.

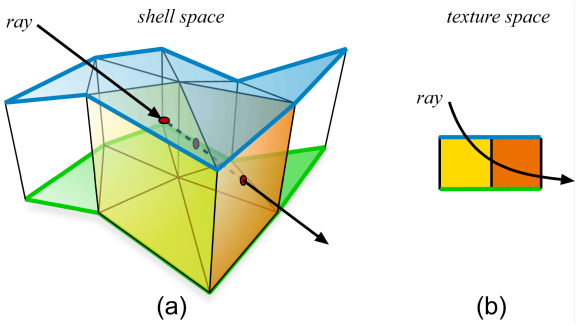


Figure 7: Rays intersect tetrahedra in shell space (a). Rays are marched in shell space, and points are transformed to texture space for density calculations. This method effectively traces curved rays through texture space (b).

Sampling is important in ray marching algorithms. If the step size is too large, sampling artifacts (aliasing) can occur. If the step size is too small, fewer aliasing artifacts occur, but rendering becomes computationally expensive. In our method, each object in texture space contains a predefined *base* step size – defined such that desired features will not be missed. We utilize a binary search procedure along the ray to find ray-surface intersections.

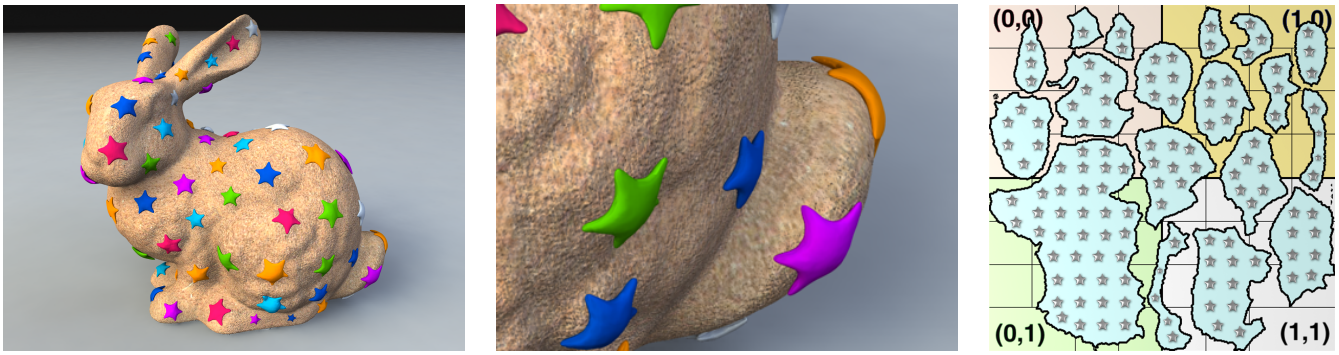
Shell mapping does not require regular grid sampled volumes and can evaluate procedural textures directly during rendering. This avoids many of the aliasing and interpolation problems encountered in previous volume texture mapping approaches [Chen et al. 2004; Peng et al. 2004; Wang et al. 2004; Neyret 1998], and also avoids the increase memory requirements of these systems.

5 Discussion

The main drawback of shell mapping is that it inherits the limitations of its two main building blocks: texture mapping and offset generation. Minimizing distortion in 2D texture mapping techniques is ongoing research. Shell mapping is essentially a texturing technique and many of the same issues (*e.g.*, shearing, warping, squishing and stretching in high curvature regions, and discontinuities across texture boundaries) may result. Distortions from 2D texture mapping can be observed in Figure 6 and in parts (a) and (b) of Figure 10.

Generating offset surfaces that are uniform over an entire surface is a challenging problem for surfaces with concave regions. In Figure 10, part (c), the surface pinches back on itself and results in a nonuniform displacement of the offset surface and squished geometry in the pinched area. Shell mapping is sensitive to offset height in addition to distortion induced by 2D texture mapping of the base surface. As shown in Figure 9 the distortion of the shell mapped cylinders increases as the height of the offset increases.

The quality of the resulting tetrahedral mesh in shell space is dependent on the quality of the original base surface triangulation and on the offset surface generation technique. In some situations (*e.g.*, long thin triangles with highly varying vertex normals) the combination of poor base surface triangulation and offset generation can lead to inverted tetrahedra and an inverted mapping in those tetrahedra. We have not experienced this problem in any of our experiments, but



(a) (b) (c)

Figure 6: In (a) a sand textured Stanford bunny is shell mapped with a star pattern. Zooming in on the tail in (b) clearly shows the mapped geometry and portions of its underside. The bunny's 2D texture map is shown in (c) with the 3D geometry to be mapped.

a possible solution is to use the determinant to detect tetrahedra inversion and reset the normals at the triangle's vertices with the average of the the vertex normals. Another possible solution is to mark those regions of texture space as unusable.

Shell mapping does not supplant displacement mapping or other texturing techniques, but complements them. Displacement mapping modifies a surface by moving vertices along associated normals, whereas shell mapping adds 3D geometric detail to the shell map region without modification of the original surface. This enables shell maps to easily create overhanging geometry, a task not easily accomplished by displacement mapping techniques.

6 Results

Figure 8 (a) and Figure 8 (b) illustrate a hypertexture function defining a weave adapted from [Ebert et al. 2003], that has been mapped to the shell space surrounding a sphere. In (b), the procedural geometry interacts with itself and with its surrounding environment, casting shadows on the "Cornell box." The model was ray traced using photon-mapping techniques, with full reflection. Figure 8 (c) shows the weave texture applied to the "Isis" data set.

We demonstrate the robustness of our technique on a benchmark data set, the Stanford bunny, in Figure 6. The zoomed view in part (c) of Figure 6 shows how the mapped geometry closely follows the curvature of the mesh and further inspection reveals views of portions of the underside of several starfish.

The whale's tail vase, shown in Figure 1, illustrates the shell mapping method as applied to a sculpted glass model. The final whale's-tail model consists of 1.17 million polygons and was shell mapped in approximately 24 seconds on an 2.5 gigahertz Apple G5 PowerPC with 2 GB of memory.

7 Conclusion

We have presented a powerful and flexible approach to mapping geometry and texture as three-dimensional detail on an object. A shell map is a bijective mapping between shell space and texture space that can be used to generate small-scale features on surfaces using a variety of rendering and modeling techniques. The method is based upon offset generation, prism identification, tetrahedral splitting, and point-in-tetrahedra methods, so that the barycentric coordinates of corresponding tetrahedra implement the mapping. We believe that shell maps have utility that extends well beyond the applications we have presented. The technique opens up new avenues of research in volumetric texture generation, geometry textures, and generalized displacement maps.

Acknowledgments

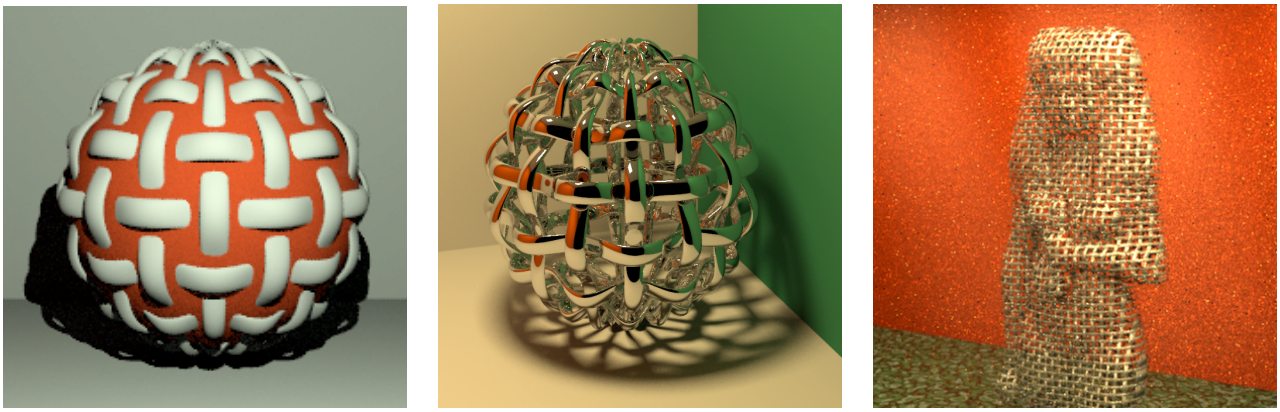
This work was supported by the National Science Foundation under contracts ACR 9982251 and ACR 0222909, and by Lawrence Livermore National Laboratory under contract B523818. We thank the Stanford Graphics Group and Cyberware for making benchmark data sets available, Bruno Levy for the LSCM texture map of the Stanford Bunny, and to Ken Musgrave for words of encouragement. Special thanks go out to Janine Bennett, Chris Co, Scott Dillard, Bao Qi Feng, Oliver Kreylos, Aaron Lefohn, John Owens, and Peter Shirley for helpful discussions, revisions, and suggestions related to this work, and to Nina Amenta for her invaluable help during the review and rebuttal process. We thank Shaun Ramsey for discussion on an early version of these ideas. Thank you to the anonymous reviewers for their useful comments and suggestions on how to make this paper better.

References

Bernd, M., and Plassmann, P. 2000. *Mesh Generation.* North Holland, Amsterdam, 291–332.

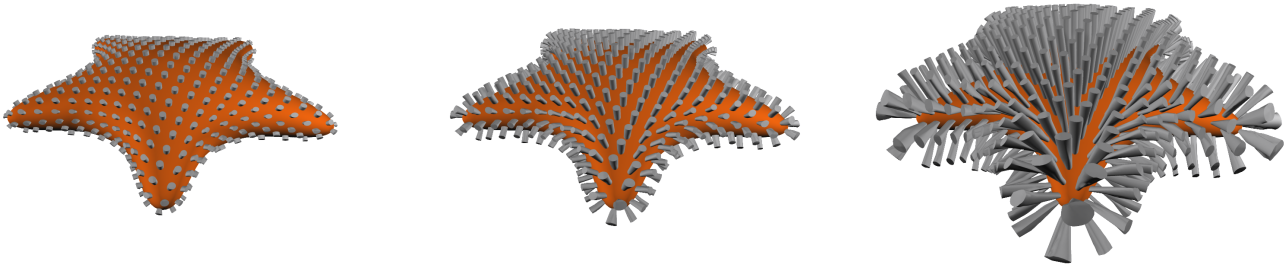
Blinn, J. F., and Newell, M. E. 1976. Texture and reflection in computer generated images. *ACM Communications 19*, 10, 542–547.

Blinn, J. F. 1978. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, ACM Press, 286–292.



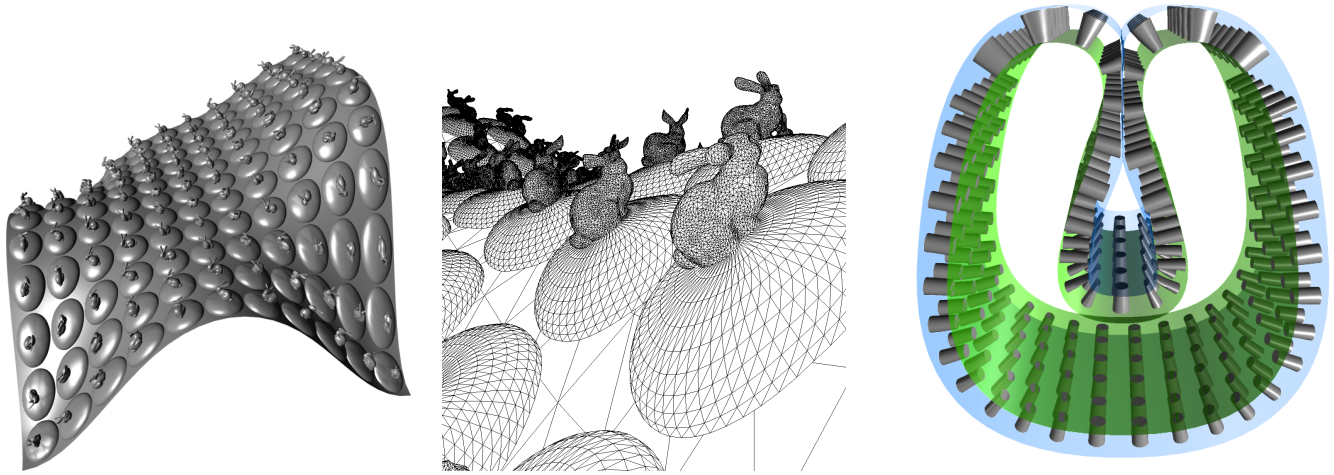
(a) (b) (c)

Figure 8: (a) A procedural texture of a weave applied to two different models. A sphere is shown in (a) with the weave texture. In (b), the procedural texture is applied to the shell space surrounding a sphere. Note that the reflective strands of the weave interact with each other, and with their environment. In (c), the weave is applied to an "Isis" model.



(a) (b) (c)

Figure 9: Cylinders mapped onto a starfish model. Note the increase in geometric distortion as the offset height is increased from (a) one percent, (b) five percent, and (c) 10 percent length of the diagonal of the starfish's bounding box.



(a) (b) (c)

Figure 10: A simplified model of the Stanford bunny (approximately 7,000 triangles) and a sphere cap are shell mapped onto a patch. Notice how the mapped geometry in (a) closely follows the curvature of the patch and the distortion, in the form of stretching, from the 2D texture mapping on the right. A zoomed image of the wireframe model is shown in (b). In (c), the green base surface pinches back on itself and the resulting blue offset surface is restricted in height in the complement of the object. The nonuniform height of the offset results in squished cylinders.

Brown, P. J. C., and Faigle, C. T. 1997. A robust efficient algorithm for point location in triangulations. Tech. rep., Cambridge University, February.

Cabral, B., Cam, N., and Foran, J. 1994. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS '94: Proceedings of the 1994 Symposium on Volume Visualization*, ACM Press, 91–98.

Chen, Y., Tong, X., Wang, J., Lin, S., Guo, B., and Shum, H.-Y. 2004. Shell texture functions. *ACM Transactions on Graphics 23*, 3 (August), 343–353.

Cohen, J., Varshney, A., Manocha, D., Turk, G., Weber, H., Agarwal, P., Brooks, F., and Wright, W. 1996. Simplification envelopes. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, ACM Press, New Orleans, LA, 119–128.

Cook, R. L., Carpenter, L., and Catmull, E. 1987. The REYES image rendering architecture. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, M. C. Stone, Ed., 95–102.

de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. 2000. *Computational Geometry Algorithms and Applications*, 2 ed. Springer.

Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., and Worley, S. 2003. *Texturing & Modeling A Procedural Approach*, 3 ed. Morgan Kaufmann Publishers.

Erleben, K., and Dohlmann, H. 2004. The thin shell tetrahedral mesh. In *The 13'th Danish Conference on Patteren Recognition and Image Processing*, S. I. Olsen, Ed., 94–102.

Farin, G. 1998. *Curves and Surfaces for Computer Aided Geometric Design*, 5 ed. Academic Press, Boston.

Guibas, L., and Stolfi, J. 1985. Primitives for the manipulation of general subdivisions and the computation of Voronoi. *ACM Transactions on Graphics 4*, 2, 74–123.

Jensen, H. W. 2001. *Realistic Image Synthesis Using Photon Mapping*. AK Peters.

Kajiya, J. T., and Kay, T. L. 1989. Rendering fur with three dimensional textures. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, vol. 23, 271–280.

Kirkpatrick, D. G. 1983. Optimal search in planar subdivisions. *SIAM Journal on Computing 12*, 28–35.

Levoy, M. 1988. Display of surfaces from volume data. *IEEE Comput. Graph. Appl. 8*, 3, 29–37.

Meyer, A., and Neyret, F. 1998. Interactive volumetric textures. In *Rendering Techniques '98*, Springer-Verlag Wien New York, G. Drettakis and N. Max, Eds., Eurographics, 157–168.

Neyret, F. 1998. Modeling, animating, and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics 4*, 1, 55–70.

Peng, J., Kristjansson, D., and Zorin, D. 2004. Interactive modeling of topologically complex geometric detail. *ACM Transactions on Graphics 23*, 3 (August), 635–643.

Peng, J. 2004. *Thick Surfaces: Interactive Modeling of Topologically Complex Geometric Details*. PhD thesis, New York University.

Perlin, K., and Hoffert, E. M. 1989. Hypertexture. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, ACM Press, 253–262.

Sarnak, N., and Tarjan, R. E. 1986. Planar point location using persistent search trees. *Communications of the ACM 29*, 669–679.

Smits, B. E., Shirley, P., and Stark, M. M. 2000. Direct ray tracing of displacement mapped triangles. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, Springer-Verlag, 307–318.

Wang, X., Tong, X., Lin, S., Hu, S., Guo, B., and Shum, H.-Y. 2004. Generalized displacement maps. In *Eurographics Symposium on Rendering*, H. W. Jensen and A. Keller, Eds.

Warren, J., Schaefer, S., Hirani, A. N., and Desbrun, M. 2003. Barycentric coordinates for convex sets. Tech. rep., Rice University.

Westermann, R., and Ertl, T. 1998. Efficiently using graphics hardware in volume rendering applications. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, ACM Press, 169–177.

Yezzi, A. J., and Prince, J. L. 2002. A PDE approach for thickness, correspondence, and gridding of annular tissues. In *ECCV '02: Proceedings of the 7th European Conference on Computer Vision-Part IV*, Springer-Verlag, 575–589.