Thick Surfaces: Interactive Modeling of Topologically Complex Geometric Details

Jianbo Peng

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September 2004

Denis Zorin

To my wife, Xin, and my son, Patrick

Acknowledgments

This document would never have come to light without the contributions from a lot of people around me. Here I would like to give them my greatest appreciation for their generous help.

First and foremost, I am particularly grateful to my adviser, Professor Denis Zorin, for his guidance over these years. I was inspired by his broad knowledge and profound insight, and learned how to approach problems, analyze them and do research work. I also benefited tremendously from his passion for research and consistent hard work.

Special thanks are due to Professor Demetri Terzopoulos and Professor Ken Perlin, who both served as members and readers of my proposal and dissertation committees. They provided valuable comments not only on the research work but also about the writing and composition of this dissertation. I also want to thank Professor Chris Bregler and Professor Eitan Grinspun for serving in my dissertation committee.

I own a great deal to people in the CAT/MRL community and am very proud of having been a part of it for several years. It is the most friendly working environment I have ever experienced. Colleagues at CAT/MRL offered me great help in improving my work and made it much enjoyable. I also want to thank all my friends for their understanding and support. I was lucky to have shared an office with Henning

Biermann, a well-beloved figure at CAT/MRL. I want to thank him for helping me in more ways than I could mention here. Should he rest in peace since cancer took him away from us in 2002.

Finally, I appreciate the support of my parents, my brother and sister throughout the whole time. Most importantly, I sincerely acknowledge my wife, Xin and my son, Patrick for their unconditional support and the joy they gave me.

Abstract

Lots of objects in computer graphics applications are represented by surfaces. It works very well for objects of simple topology, but can get prohibitively expensive for objects with complex small-scale geometrical details.

Volumetric textures aligned with a surface can be used to add topologically complex geometric details to an object, while retaining an underlying simple surface structure. The simple surface structure provides great controllability on the overall shape of the model, and volumetric textures handle geometric details and topological changes efficiently.

Adding a volumetric texture to a surface requires more than a conventional two-dimensional parameterization: a part of the space surrounding the surface has to be parameterized. Another problem with using volumetric textures for adding geometric detail is the difficulty of the rendering of implicitly represented surfaces, especially when they are changed interactively.

We introduce *thick surfaces* to represent objects with topologically complex geometric details. A thick surface consists of three components. First, a base mesh of simple structure is used to approximate the overall shape of the object. Second, a layer of space along the base mesh is parameterized. We define the layer of space as a *shell*, which covers the geometric details of the object. Third, volumetric textures of

geometric details are mapped into the shell. The object is represented as the implicit surface encoded by the volumetric textures. Places without volumetric textures are filled with patches of the base mesh.

We present algorithms for constructing a shell around a surface and rendering a volumetric-textured surface. Mipmap technique for volumetric textures is explored as well. The gradient field of a generalized distance function is used to construct a non-self-intersecting shell, which has other properties desirable for volumetric texture mapping. The rendering algorithm is designed and implemented on NVIDIA GeForceFX video chips. Finally we demonstrate a number of interactive operations that these algorithms enable.

Contents

List of Figures

List of Appendices

Chapter 1

Introduction

1.1 Model representation

People are using more and more complex objects in computer graphics applications, such as special visual effects, video games, virtual environments, etc. These objects tend to have tremendous geometric and textural details. Most of them are represented by surfaces: either textured polygonal meshes or higher-order primitives, such as spline patches or subdivision surfaces. These common surface representations work well for objects of relatively simple topology and continuous geometric structure. For many types of objects however, the local geometry can be highly complex. Examples include fur, bark, cracked surfaces, grills, peeling paint, chain-link fences and others. In these cases, using meshes or patches to represent small-scale geometry of complex topology often leads to excessively large number of small faces or patches, which require a lot of resource to process and tend to be error-prone in simulation. The surface representation approach limits our ability to construct and process more than a few of such topologically complex objects on current personal computers. But if

we ignore the small-scale details, a complex surface often has a simple overall shape, which can be well represented by a mesh or a smooth surface.

Volumetric representation is preferable for objects with complex topology and geometric details. To obtain a volumetric representation of an object, a function is sampled at regular grid points in a volume containing the object. The sampled function values are stored and interpreted as an implicit representation of the object. The three-dimensional function can be the distance function to the object, the density, transparency of the object or other user-defined functions with a proper reconstruction process. For the distance function, the object is reconstructed as an ISO-surface where the function value is zero. Volumetric representation handles topological operations effectively, but requires a lot of storage spaces and computational resources. If the object has a very simple large-scale shape, most of the sampled function values do not provide helpful information to the geometric details of the object. The spaces and resources used to store and process them are thus a large waste. Another drawback of volumetric representation is aliasing, which is caused by sampling on regular grid, and results in artifacts on the reconstructed model. Features not aligned with the grid directions are subject to the aliasing problem.

The volumetric texture aligned with the surface is a combined volume and surface representation of three dimensional objects. It overcomes some of the problems we described above, and makes it possible to efficiently represent geometrically and topologically complex details in implicit form by encoding the surface as an ISO-surface in a layer. This idea was explored by a number of researchers in the past (see Chapter 2).

1.2 Thick surfaces

We introduce *thick surfaces*, a combined volume-surface representation for handling models with topologically complex geometric details (Figure 1.1), extending the idea of volumetric textures. A thick surface consists of three components. First, a base mesh of simple structure is used to approximate the overall shape of the object. Second, a layer of space along the base mesh is parameterized. We define the layer of space as a *shell*. The shell covers the geometric details of the object. Third, volumetric textures of geometric details are mapped into the shell. The object is represented as the implicit surface encoded by the volumetric textures.

This representation has important advantages over conventional surface representation for objects with complex details:

- It uses simple and efficient data structures (textures) to represent highly irregular geometry.
- Small features of high topological complexity can be easily introduced and modified.
- Image processing techniques can be used to modify small-scale geometry without topological constraints.
- Hierarchical representations can be naturally constructed using filtering on volumetric textures.
- One can easily use procedural modeling and simulation to produce complex effects near the surface.

We focus on algorithms central to the goal of using this approach in modeling applications. Specifically, in addition to surface parameterization required by 2D

Figure 1.1: A surface with fine-scale details added as volumetric textures. The small picture shows the texture detail. The volumetric textured model is further edited by boolean operations.

texturing, volumetric textures require parameterizing a region of space near a surface (a volume layer). Furthermore, as the surface deforms, the volume layer parameterization needs to be recomputed. Most of previous work on volumetric textures used techniques such as normal displacement, which results in self-intersections near concave features. We describe an algorithm for computing volume layer parameterizations with a number of desirable properties, which can be used to update the parameterization interactively.

While ISO-surfaces are convenient for many types of operations, they are much more difficult to render than conventional meshes. We describe algorithms for volumetric texture rendering that enable interactive manipulation of volumetric-textured objects. Our algorithm for rendering volumetric-textured surfaces extends direct slice-based ISO-surface rendering for volumes. We take advantage of the programmable graphics hardware to reduce the geometry requirements of the slice-based methods, which is crucial for interactive rendering of volumetric textures.

Chapter 2

Previous Work

2.1 Volumetric textures

The idea of volumetric textures goes back to the work by Kajiya and Kay [23], Perlin and Hoffert [33], both of which appeared at Siggraph at the same year. They introduced the idea of representing a pattern of 3D geometry in a reference volume. The reference volume is actually a 3D texture in which both a volume density value and a lighting model are stored at each point, and not tied to the geometry of any particular surface. It is tiled over an underlying surface much like a 2D texture. Rendering of such textured surface is accomplished by ray casting. Brightness from the lighting model and illumination intensity at each intersection point of a ray and 3D texture cells is attenuated and accumulated along the ray. Kajiya and Kay [23] used this model for fur rendering, while Perlin and Hoffert [33] applied it in modeling fire, woven cloth, as well as curly fur.

Our work was motivated by the work of F. Neyret and coworkers (e.g. [29, 30, 28]) as well as the recent work on fur rendering [25, 26]. In their work on volumetric

textures, Neyret and coworkers defined an offset vector for each vertex of a mesh, but they did not show how to generate these vectors. Texture coordinates are assigned to the two endpoints of the vector, and interpolated for other points on the line segments. An opacity value and a normal distribution function (NDF) are stored in each voxel of the texture. The NDFs, which are approximated using ellipsoids, defines the per voxel reflectance properties. The volumetric texture is filtered at different scales, forming a hierarchy similar to a mipmaped texture. The filtered volumetric texture data is regarded as a fully expanded octree, through which rays are traced in rendering. While tracing a ray into the octree, cells of the appropriate sizes are selected from different levels. Those selected cells are then illuminated and their opacity is accumulated to obtain the final rendered image.

Our geometry is to some extent similar to the slab representation used for modeling weathered stones [17] and volume sculpting [1]. Agarwala uses multi-resolution data structure in each slab to represent the solid material for sculpting [1]. Each data field is rendered as a box and invisible geometry is culled for interactive rendering. Final rendering is obtained by ray tracing. Slice rendering cannot be used because of the multi-resolution data structure.

The surface aligned slabs have two advantages over traditional volumetric representation. They have less aliasing and less storage coherency for layered objects. The disadvantage is that they are predefined [1] or first displaced in normal directions then manually corrected as in [17]. The slabs are not automatically generated from the input mesh, thus cannot be interactively deformed. We present a fully automatic method to construct layers around a surface. The simple structure of the layers gives us seamless transition between parametric and volumetric representations. Slice rendering can be applied for fast interactive rendering.

2.2 Stable medial axes

Our construction is closely related to the work in vision and medical imaging on using various types of medial axis approximation to analyze shape and extract surfaces from volumetric data (e.g. [34, 18]). The medial axis of a 2D domain is the set of the centers of the maximal inscribed circles of the domain. In case of 3D, the medial axis is similarly defined, with spheres in place of circles.

Choi and others did extensive work on medial axis on planar domain, citing [9, 10, 11, 12] for a few of his publications. They explained the mathematical theory of medial axis transform [9, 11] and developed algorithms for various applications [10], especially for computing 2D offset curve of a planar domain whose boundary is composed of rational curve segments [12]. Instead of computing intersections of each offset curve segments and finding the valid parts of the 2D offset curve, Choi and coworkers decompose the planar domain into many fundamental domains according the radius of the maximal inscribed circle and compute offset curves in each fundamental domain. A fundamental domain does not contain any locally maximal or minimal radius component. The medial axis of a fundamental domain is a simple curve segment and the offset curve can be easily computed. Results are then connected to form the offset curve of the original domain. Change in topology is taken care of in fundamental domains. This medial axis approach can be modified in each fundamental domain so that the topology of the offset curve does not change by using medial axis as the offset curve where the real offset curve segments are invalid. Although it can then be extended to 3D and used to construct an interior shell (refer to Chapter 4 for definition) of a surface, the computation of medial axis remains too expensive. Any deformation of the base surface will require re-computation of the

whole medial axis. That makes the medial axis approach unsuitable for interactive applications. Our generalized distance function (Chapter 4) is similar to some of the medialness functions used to construct stable medial axes, but we do not need any form of medial axes in our shell construction process and avoid the expensive computation cost.

Our work is closest to [40] which solves the Hamilton-Jacobi equations for the medialness function on a regular grid to recover a skeleton. In [40], Siddiqi and others introduced an algorithm for simulating the eikonal equation in medial axis tracking. They first sample the initial boundary into marker particles, then evolve these particles. The trajectories of the particles are governed by the vector field obtained from the gradient of the Euclidean distance function. For accuracy, they start with a dense sequence of marker particles and generate new particles when original ones drift apart in rare-faction regions. Special numerical technique is applied to estimate the gradient of the Euclidean distance function, because finite differences will lead to error near singularities (the medial axes), where the function is not differentiable. The gradient field of our generalized distance function is much easier to estimate than that of the Euclidean distance function, due to much less singularities and smaller gradient magnitude near medial axes.

Our generalized distance function leads the shell off or stops at the medial axes depending on the type of medial axes it meets, effectively trimming the unstable medial axes. Invalid offset surface and self intersection are avoided automatically. R-functions have been used to achieve similar effects ([36, 37, 31]), but with more complex computation based on a CSG representation of the surface.

2.3 Direct ISO-surface rendering

Indirect methods for rendering ISO-surfaces [27, 8] require significant preprocessing and results in large meshes. Westermann and Ertl presented a method to render a shaded ISO-surface directly from volumetric data using up-to-date graphics hardware in [43]. The volumetric data is used as a 3D texture without constructing an explicit polygonal mesh. Geometry is rendered by slices with material used as alpha. Two types of shading (with gradient as texture and without gradient) were implemented. In gradient-less case, they compute the differencing derivative in the light direction, which is the dot product of the gradient with the light direction vector. It is done in two passes, using positions as colors and rendering twice with texture coordinates shifted along the light direction. Rezk-Salama and others use multiple textures for interpolation of 2D textures [35]. It makes the technique in [43] single-pass, but requires high slice density. Variations of the idea of using slices to render displacement maps are explored in [24, 16, 38]. Kautz and Seidel described a clever technique for converting displacements along one direction to displacements along two other directions [24]. This allows one to choose slice direction closest to the perpendicular of the view direction and minimize the problem of viewing slices at grazing angles.

Our rendering technique extends the direct volume and ISO-surface rendering approach based on slicing and 3D textures (or collections of 2D textures), which is originated in [6, 45, 14]. Most closely our work is related to ISO-surface rendering and displacement map rendering described above [43, 35, 24, 16, 38]. For volumetric textures a slice-based approach is described in [28]. We apply an extension of these techniques to a collection of relatively small distorted volumes. Our technique has the crucial advantage of allowing a more flexible choice of the number of slices per

volumetric element, while minimizing visual artifacts associated with insufficient slice density. A more detailed comparison is given in Chapter 5.

2.4 Implicit surfaces and volume modeling

There is an extensive body of literature related to volume-based representations (see [5] for a list of references); some recent important work includes [7, 20]. Carr and others use polyharmonic radial basis functions (RBF) to reconstruct smooth manifold surfaces from point cloud data and to repair incomplete meshes [7]. An object's surface is defined implicitly as the zero set of an RBF fitted to the given point cloud data. Adaptively sampled distance fields (ADF) are introduced in [20] to effectively capture fine details on geometric objects. Interactive and procedural volume sculpting techniques [42, 1, 15] can be applied to our surface representation. Most work on volumetric modeling focuses on volumetric data in pure form, i.e. objects are represented as level sets of a function defined by volume samples. We concentrate on techniques which blend parametric and volumetric representations.

2.5 Structured mesh generation

Constructing a collection of layers aligned with a surface is a common problem in structured mesh generation. Mesh generation is a large and complex field aiming to build meshes suitable for a variety of numerical algorithms to solve PDEs (see, e.g. surveys [3, 21] and the book [41]). Such meshes often have to satisfy stringent requirements for the algorithms to achieve optimal or nearly-optimal convergence rates. Particularly, object-aligned grids are important for CFD problems [21].

Yezzi and Prince presented a PDE approach to compute point correspondences and grids within annular tissues [46]. They derive a vector field in the volume sandwiched between two simple surfaces from the solution of the Laplace equation. Correspondence trajectories are implicitly built using the vector field as the tangent vector field. Each trajectory connects two points on the two boundary surfaces respectively. To generate a structured grid in the sandwiched volume, the trajectories of a set of seed points are evenly segmented. The splitting points are used as the grid points and the trajectories as the grid lines in one direction. The result grid does not self-intersect, and follows the geometry nicely. This method applies to occasions when both boundary surfaces are well defined, and the stability and performance depend heavily on the PDE solvers used in each step. But we only have one boundary surface and need to generate the grid and the other boundary surface at the same time. Traditional grid generation algorithm does not trivially apply to our problem.

We aim to construct a shell aligned with the surface efficiently, maintaining non-degeneracy without explicitly minimizing a distortion measure. As explained in greater detail in Chapter 4, the desirable behavior of shells is different from the desired behavior of grids used for solving PDEs.

Cohen and coworkers construct an envelope shell around a surface for mesh simplification [13]. For some proposed thickness ϵ, they apply normal displacement for each vertex and limit the displacement magnitude to avoid self intersection. In the analytical method, a 'safe' ϵ value is found for each vertex. In the numerical approach, vertices walk along normals in small steps and stop when they cannot proceed without generating self intersections. An octree is used in both analytical and numerical methods to accelerate the computationally expensive intersection test at each step. The envelope shell serves the mesh simplification application well, which

only uses it to bound the error. Walking along normals tends to bring all vertices together in high curvature concave area, making the shell degenerate or generating highly distorted shell. Usually, smoothing in both walking direction and displacement magnitude is desired to avoid visual artifacts if the shell is to be used in 3D texture mapping.

Level set methods [39] present the main alternative to our approach in shell generation. However, it has several difficulties in our scenario. See Chapter 4 for a more detailed comparison of shells generated by different methods.

Hyperbolic grid generation methods are similar in nature to our method: the surface is propagated outwards. However the direction and speed of propagation is determined by solving a non-linear PDE, which is computationally expensive. Furthermore, the criteria used to formulate the PDE (volume preservation and orthogonality), are not necessarily the best for our application. Volume preservation, just like the marching method, tends to expand the shell excessively outwards in concave areas. We consider this property less appropriate for texturing application than compressing the shell as our method does.

Chapter 3

Representation

3.1 Shells

A parameterization of a layer of space around a surface is required to apply a volumetric texture to the surface. The parameterized shell can lie outside the surface, having the surface as the inner boundary. We refer this kind of shells as *exterior shells*. Similarly, we define *interior shells*, which lies inside the surface, and *envelope shells* with layers located on both sides of the surface (Figure 3.1).

Different shells are suitable for different applications. For example, exterior shells are preferred for modeling of fur and other objects with attached geometric details, while interior shells are better choices for modeling of cracks on tree trunks. If preserving the exact shape of the surface is not crucial, and the surface need not be retained as a shell layer, envelope shells allow us to avoid running into some difficult concave areas on a surface and achieve better shell quality. When a high curvature feature will result in tightly compressed exterior or interior shell, envelope shell can let the shell expand to the other side and achieve smaller thickness change and less

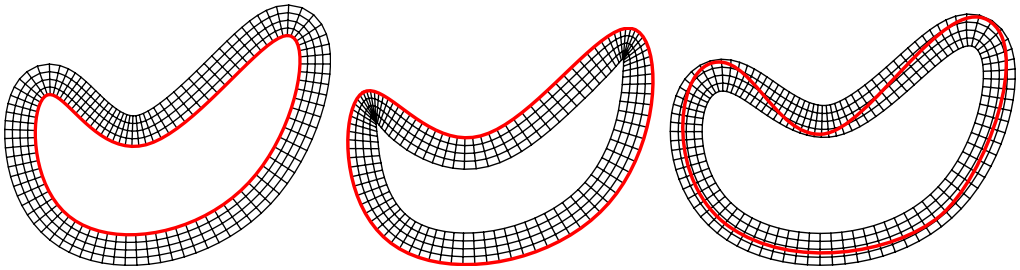

Figure 3.1: Three types of shells plotted in 2D. The base surfaces are shown in thick red. Exterior shell (left plot) lies outside the base surface; interior shell (middle plot) lies inside the base surface; and envelope shell (right plot) could be both side of the base surface.

distortion of the mapped texture. However, how to generate envelope shells is not as clear as the cases of exterior and interior shells.

Exterior and interior Shells can be considered as generated by expanding its boundary surface and grow a skin on the surface. The expanding trajectory of each point on the surface can be used as grid lines, providing a natural parameterization of the shell (Figure 3.2). For some expanding methods, the trajectories are along a straight line, e.g. the widely used normal displacement method. Shells with straight grid lines are usually efficient in storage and processing, providing substantial advantage for modeling of complex objects.

To overcome the self-intersection problem of the normal displacement method, other methods are developed to expand the surface and construct the shell, which is to be discussed in more detail in Chapter 4. In general, these trajectories are not straight lines any more, which requires more sample points to represent.

For volumetric texture applications, it is enough and sometimes preferred to use simple shells for which the displacement away from the surface is along a straight

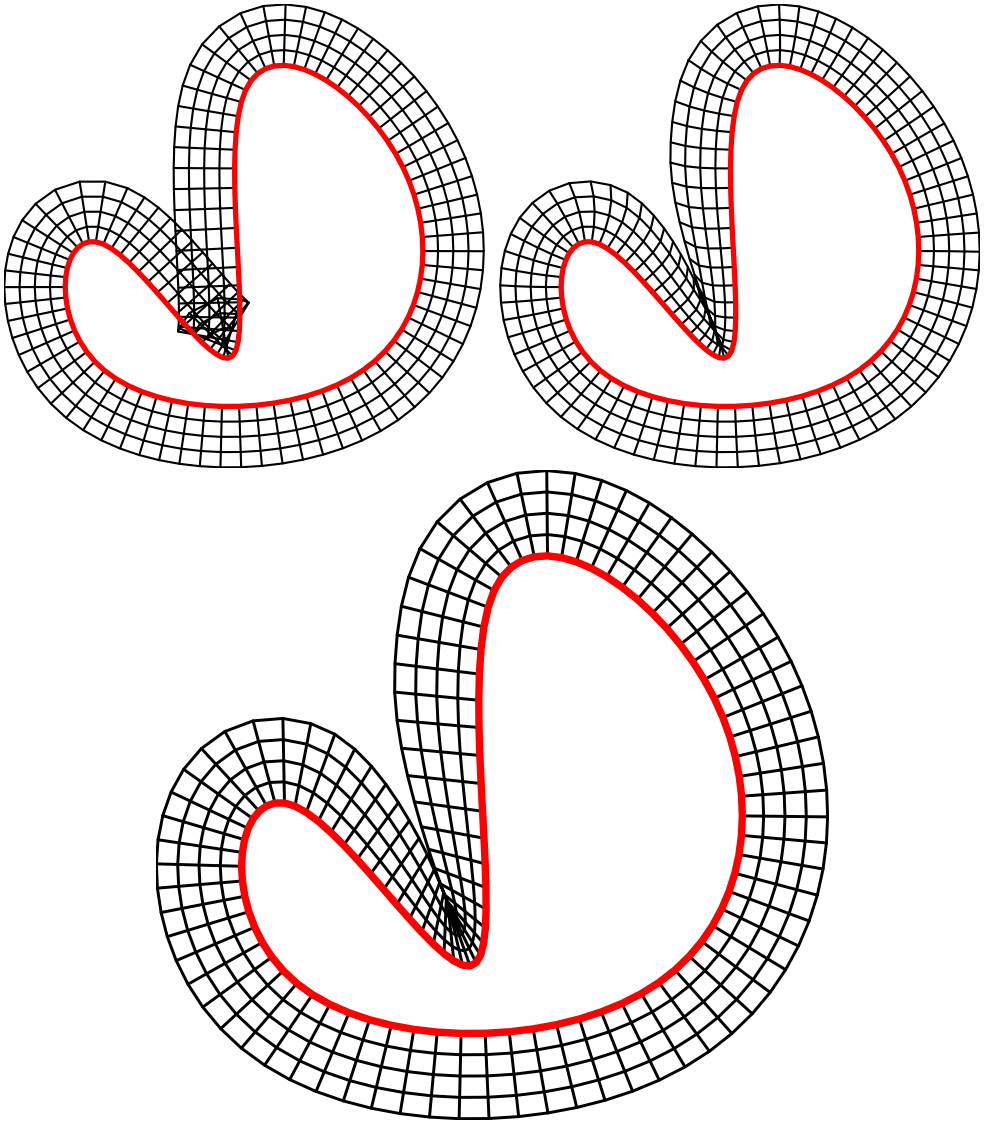

Figure 3.2: Shells generated and parameterized with straight and non-straight expanding trajectories. The upper left picture is a normal displacement shell. The upper right one shows the shell generated by our methods with the actual trajectories of the points on the surface. The lower one is also generated by our methods, but uses adjusted straight trajectories as grid lines.

line. The basic geometric information required to represent such a parameterized shell is minimal and does not substantially increase the memory requirements. To represent an exterior shell around a surface, we can specify the extending direction of the shell at each vertex and the thickness of the shell in this direction. Both the direction and the thickness can be encoded into a three-dimensional vector. In practice, slightly more information are stored along with this vector.

We refer to the initial surface for which we construct a shell as the *base surface*. We consider shells that are obtained by displacing points of the base surface along line segments defined at vertices, which we call *directors*. At each vertex of the surface, we store the director, shell thickness, the number of shell layers and texture coordinates. The shell in our setting consists of slabs corresponding to the faces of the mesh or individual patches. Each slab is a deformation of a prism.

3.2 Volumetric textures

The main additional storage incurred in our thick surface applications is the 3D textures associated with the surface. The alpha channel of the texture defines the effective surface implicitly as the ISO-surface corresponding to a fixed alpha value. The remaining texture channels are used to store the gradient of the alpha values. The number of layers in the shell corresponds to the number of pixels in the texture in the direction perpendicular to the surface. It can vary across the surface. In an extreme case, as shown in Figure 3.3, the number of layers can go down to one. If no holes need to be specified on the part of the surface where only one layer is used, the texture can be entirely omitted. By choosing appropriate alpha values on the textured parts of the surface, we can easily ensure smooth transition between the implicitly defined

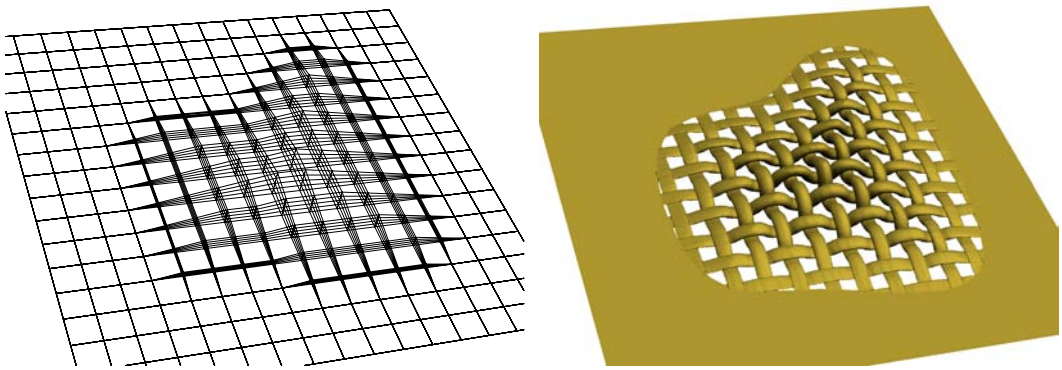

Figure 3.3: A surface with a partial volumetric texture attached. The left picture shows the structure of the shell.

part of the surface and the part that has a conventional polygonal representation.

In our implementation we use multi-resolution surfaces with subdivision connectivity for the base surface. Texture coordinates are specified only for vertices of the top level mesh in the multi-resolution hierarchy, and interpolated for vertices generated by subdivision. However, the basic techniques that we have developed can be applied to arbitrary meshes with 2D texture coordinates.

Chapter 4

Constructing Shells

Most previous works on volumetric textures used normal displacement to generate the shell around a surface. In concave area, self-intersection quickly develops even with very small displacement, resulting in noticeable artifacts in rendering. A few others did manual correction on the result of normal displacement when self-intersection is observed. This approach is obviously limited to simple objects, and does not work at all when the base surface is interactively deformed and the shell needs to be updated at each frame.

In this chapter we describe our basic algorithm for constructing shells around surfaces. Intuitively, one can think about the shell construction process as growing thick skin on the surface. Shells constructed by our method behave more or less like elastic compressible skin, which is considered a desirable property for the purpose of volumetric texturing.

Our algorithm is based on the observation of the gradient field of a generalized distance function, which we will discuss in detail. The most important property of the generalized distance function is that it enables us to avoid self-intersection

automatically without incurring heavy computational burden. We start with some desirable properties of a shell for our purpose.

To make a shell useful for volumetric texturing, a number of properties are desirable:

- The layers should not intersect. This requirement is motivated by the ”skin” metaphor which we believe to be natural for manipulating this type of surface representation in many cases.

- The layers should have the same connectivity. This is crucial for defining a vertex's volumetric texture coordinates (s, t, r). They can be obtained as follows in this case: (s, t) are given by the base surface parametrization which is assumed to be known, and r is incremented proportionally along the displacement director from the base surface.

- The shell should maintain prescribed thickness whenever possible. However, if thickness cannot be maintained due to geometric obstacles, valid shell with locally decreased thickness should be produced. This corresponds to the intuitive idea of elastic “sponge-like” skin. Note that volume preservation is somewhat undesirable as it is likely to result in fold formation.

- The shell should be close to the one obtained by normal displacement whenever possible.

- The shell at a point should depend only on the parts of the surface close to point. This property is important for modeling applications and efficient implementation.

Next, we describe our shell construction algorithm motivated by these requirements.

4.1 The basic algorithm

To describe our algorithm in detail we introduce some formal notation. We assume that our base surface is a mesh, or a higher order surface associated with a mesh (subdivision surface, spline surface etc.) without self-intersections. Formally, our goal of constructing a shell around the surface can be described as follows.

> Given a surface M embedded in $\mathbf{R}^3$, construct a one-to-one map $f(\mathbf{x}, t)$ from the direct product $M \times [0, 1]$ into $\mathbf{R}^3$. We call the length of the curve $t \to f(\mathbf{x}_0, t)$, where $\mathbf{x}_0$ is a fixed point on M and t is in $[0, 1]$, the *thickness* of the shell at $\mathbf{x}_0$ and denote it $h(\mathbf{x}_0)$.

4.1.1 Main ideas

Our algorithm is based on a simple idea: to construct a shell, we always need to find a corresponding point off the surface for every point on the surface. The corresponding points are further away from the surface when the desired shell thickness is larger. We can consider that as a process of moving every point on the surface along a certain path and stopping when the desired shell thickness is reached. In the places where this is impossible (the simplest example is the center of a sphere when we try to build an interior shell of a thickness exceeding the radius), the shell cannot be extended further. More formally, in the case of normal displacement, the direction of motion is defined by the gradient of the distance function, which points exactly along the

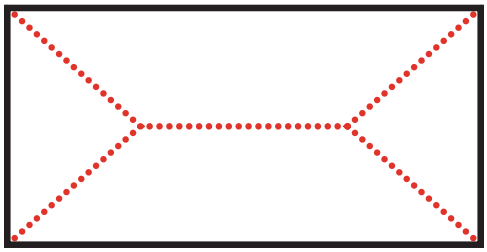

Figure 4.1: Medial axes of a box.

normal direction to the surface whenever it is defined. The lines of the gradient field of the distance function are just straight lines. However, they cannot be extended indefinitely: the distance function is singular at points on the medial axes. Extending beyond a medial axis results in self-intersections of the shell. Unfortunately the medial axis comes close to the surface at concavities and extends all the way to an object at sharp features, as in the example shown in Figure 4.1.

4.1.2 Extended distance function gradient

We consider this example in more detail. Suppose we are building an interior shell. In the process of moving away from the surface along the gradient, it is often still possible to move even we already hit the medial axis. E.g., if we start from the corner of the box, we just move along the branch of the medial axis. While the complete gradient of the distance function is not defined in the neighborhood of a medial axis, it is defined along the medial axis. If we define the gradient of the distance function on the medial axis to be the gradient along it whenever it is defined, we call this *extended distance function gradient*. While the magnitude of the regular distance function gradient is always one, the magnitude of this extended gradient is less than one on medial axes: the sharper the angle of the concavity, the smaller it is. In

general, the magnitude of the extended gradient at a point $\mathbf{x}$ on the medial axis equals $cos(\alpha)$, where α is the half angle between the two surface normals at the two closest surface points[1] from $\mathbf{x}$ (Figure 4.2). For the horizontal part of the medial axis of the box, it is identically zero. We note that these are exactly the points where no further motion is possible, because shell parts extended from two sides of the box run into each other. This shows that the magnitude of the gradient of the distance function along the medial axis can be used as a measure of how easy it is to move a particle located at that point of space away from the surface. If we are at a point off the medial axis, at which the gradient magnitude is one, there is an obvious direction to expand the shell locally. The magnitude of the gradient allows us to distinguish the points where the shell can grow quickly and where it needs to stop and start compressing.

4.1.3 The basic algorithm

These observations suggest the following simple abstract algorithm for constructing the director of a shell:

> *To obtain the director of a shell at point* $\mathbf{x}$*, first follow the extended gradient field* $g(\mathbf{x}) = \nabla_{\mathbf{x}} d(\mathbf{x}, M)$ *of the distance function, and solve the ODE:*
>
> $$\frac{\partial F(\mathbf{x}, t)}{\partial t} = hg(\mathbf{x}) \tag{4.1}$$
>
> *where* h *is the desired thickness, and* $F(\mathbf{x}, t)$ *is position along the integral line of the gradient field passing through* $\mathbf{x}$*. Then define* $f(\mathbf{x}, t)$ *by linear interpolation between* $\mathbf{x}$ *and* $F(\mathbf{x}, 1)$*.*

[1] If there are more than two closest surface points from the medial axis point $\mathbf{x}$, the extended gradient at $\mathbf{x}$ can be defined as zero, meaning you cannot move away from the surface any more once you hit the point $\mathbf{x}$.

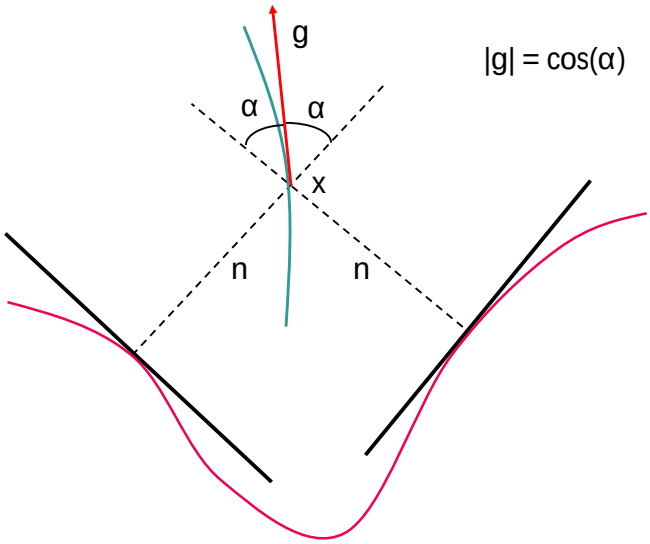

Figure 4.2: The extended gradient g at a medial axis point $\mathbf{x}$, where n is surface normal, and α is the half angle between the two normals.

Consider the trajectory $F(\mathbf{x}, t)$ of a surface point $\mathbf{x}$. Note that as long as the integral curve $F(\mathbf{x}, t)$ does not reach the medial axis, it remains a straight line with unit speed parametrization, as $\|g(\mathbf{x})\| = 1$. If it reaches the medial axis, the speed changes. If the point on the medial axis we have reached is an unstable medial axis branch extended to a local concave region, the gradient will remain high, and we will move away from the surface at a slightly slower speed. If the gradient is small, which indicates a more stable medial axis point, the shell starts compressing, and the total length of $f(\mathbf{x}_0, t)$ ends up being less than h. Note that this simple procedure ensures smooth transition between no compression and high compression of the shell in different areas. The endpoints of the trajectories, if they reach the medial axis, always end up at stable points with low gradient, effectively trimming the unstable medial axis branches. Figure 4.3 illustrates the shell generation process

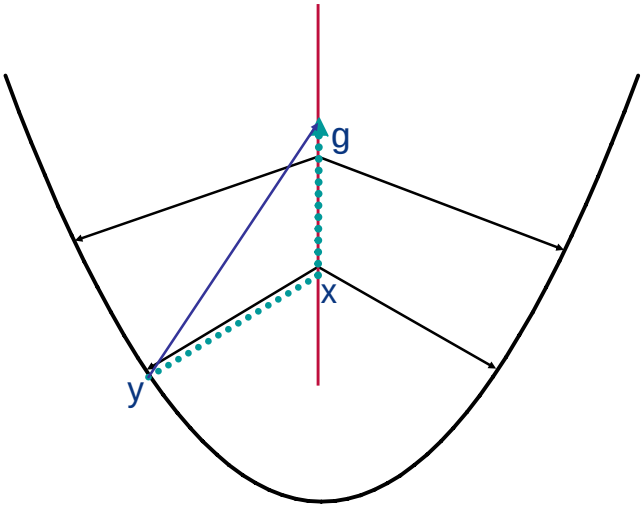

Figure 4.3: Moving along the extended distance function gradient. We start from a surface point $\mathbf{y}$ and move along the surface normal, which is the gradient of the Euclidean distance function. The moving trajectory is shown in dotted lines. Once we reaches a medial axis, denoted by $\mathbf{x}$, we turn to the extended distance function gradient g and move along the medial axis. The distance to the surface, indicated by the arrows pointing to the surface, increases along the actual moving trajectory.

by this algorithm. Once the shell reaches the thickness h in parameterization space, we connect the starting point and the end point and use the straight line segment as a virtual trajectory.

For the box example and sufficiently large h, this yields a shell completely filling the box as shown in Figure 4.4. Unfortunately, it is difficult to solve Equation 4.1, as the field is discontinuous, and we would have to compute the medial axis. To make the algorithm practical, we replace the distance function with a function we call L_p*-averaged distance function*.

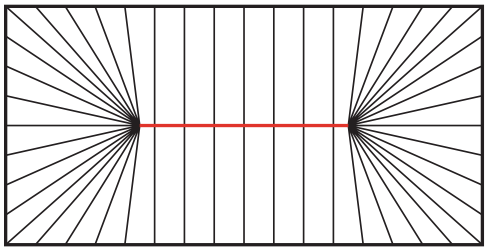

Figure 4.4: The shell with target thickness exceeding one half of the box size constructed using the gradient along the medial axis. The shell director lines are shown.

4.2 Averaged distance functions

4.2.1 Generalization of distance functions

The basis of our definition is the following simple observation. We can rewrite the distance function from a point $\mathbf{x}$ to a surface M as

$$\begin{aligned} d(\mathbf{x}, M) &= \inf_{\mathbf{y}\in M} |\mathbf{x}-\mathbf{y}| \\ &= \left(\sup_{\mathbf{y}\in M} |\mathbf{x}-\mathbf{y}|^{-1} \right)^{-1} \\ &= \left(\| |\mathbf{x}-\mathbf{y}|^{-1} \|_{L_\infty(M)} \right)^{-1} \end{aligned} \tag{4.2}$$

This definition lends itself to a natural generalization:

$$\begin{aligned} d_p(\mathbf{x}, M) &= \left(\| A^{-1} |\mathbf{x}-\mathbf{y}|^{-1} \|_{L_p(M)} \right)^{-1} \\ &= A^{1/p} \left(\int_M |\mathbf{x}-\mathbf{y}|^{-p} d\mathbf{y} \right)^{-1/p} \end{aligned} \tag{4.3}$$

where A is the area of the surface M. This normalization by the area is introduced to ensure that the gradient of this distance function is non-dimensional and close to magnitude 1 at infinity, which mimics the properties of the gradient of the Euclidean distance.

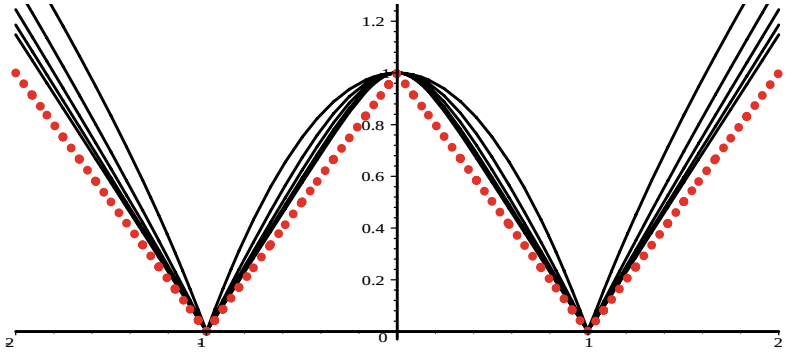

Figure 4.5: Averaged distance functions. The red dotted plot shows the standard distance function from a point on the line to the set of two points $\{-1, 1\}$. Other lines show the averaged distance functions for different values of p.

4.2.2 Gradient fields

Intuitively, one can expect the gradient direction field of this function to have similar properties to the gradient field of the distance function as p approaches ∞. Another intuitive interpretation of this distance function is the potential of field generated by particles on the surface raised to the power $-1/p$. In practice, we have observed that even for small values of p, the fields are quite similar. This is illustrated in Figure 4.5 and 4.6. The one-dimensional averaged distance functions are compared to the standard distance function in Figure 4.5, and fields of several values of p in a two-dimensional box are shown in Figure 4.6.

However, unlike the case of the standard distance function, the gradient of this function is well defined away from the surface, as the integration and differentiation

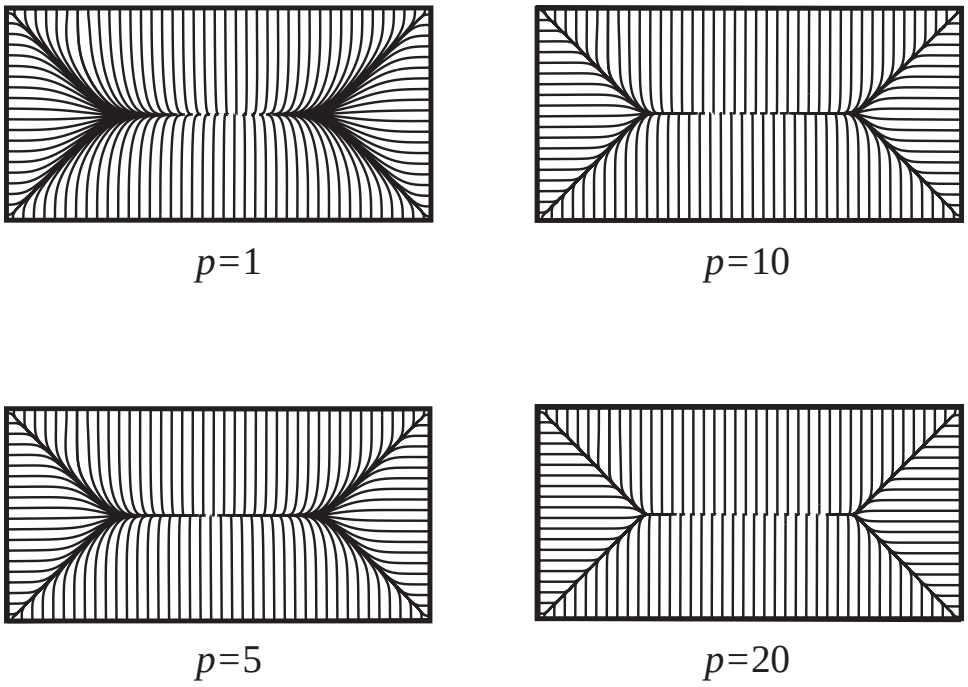

Figure 4.6: Field lines of the gradient field of the distance function for several values of p.

can be exchanged.

$$\begin{aligned} g_p(\mathbf{x}, M) &= \nabla_{\mathbf{x}} \left(d_p(\mathbf{x}, M) \right) \\ &= \frac{1}{A} (d_p(\mathbf{x}, M))^{p+1} \int_M (\mathbf{x} - \mathbf{y}) |\mathbf{x} - \mathbf{y}|^{-p-2} d\mathbf{y} \end{aligned} \tag{4.4}$$

4.3 The shell constructing algorithm

Using the averaged $d_p(\mathbf{x}, M)$ yields the analog of 4.1 in which the gradient has an explicit expression and the medial axis does not have to be computed explicitly.

$$\frac{\partial F(\mathbf{x}, t)}{\partial t} = h g_p(\mathbf{x}) \tag{4.5}$$

The L_p-averaged distance function shares a number of properties with the conventional distance function. It can be proved that in 3D for $p > 1$ (and $p > 0$ in 2D) the direction of the gradient g_p at points of a smooth surface coincides with the normal[2]. Furthermore, in all our experiments we have observed that the magnitude of the gradient remains close to one near the surface, and decays in the area close to the conventional medial axis. So our function defines a fuzzy medial axis, pruning away insignificant branches corresponding to concavities. The magnitude of the gradient field comes close to zero only in areas in which the shell genuinely cannot be expanded (see Figure 4.7 and 4.8 for the results of our two-dimensional experiments on deforming curves). The crucial difference of the averaged distance function from the conventional distance function is that the gradient field can be easily integrated.

[2]An interesting observation that $p = 1$ in 3D corresponds to the the electric field potential which makes it clear that this value cannot be used: e.g. the potential is constant inside a hollow uniformly charged sphere.

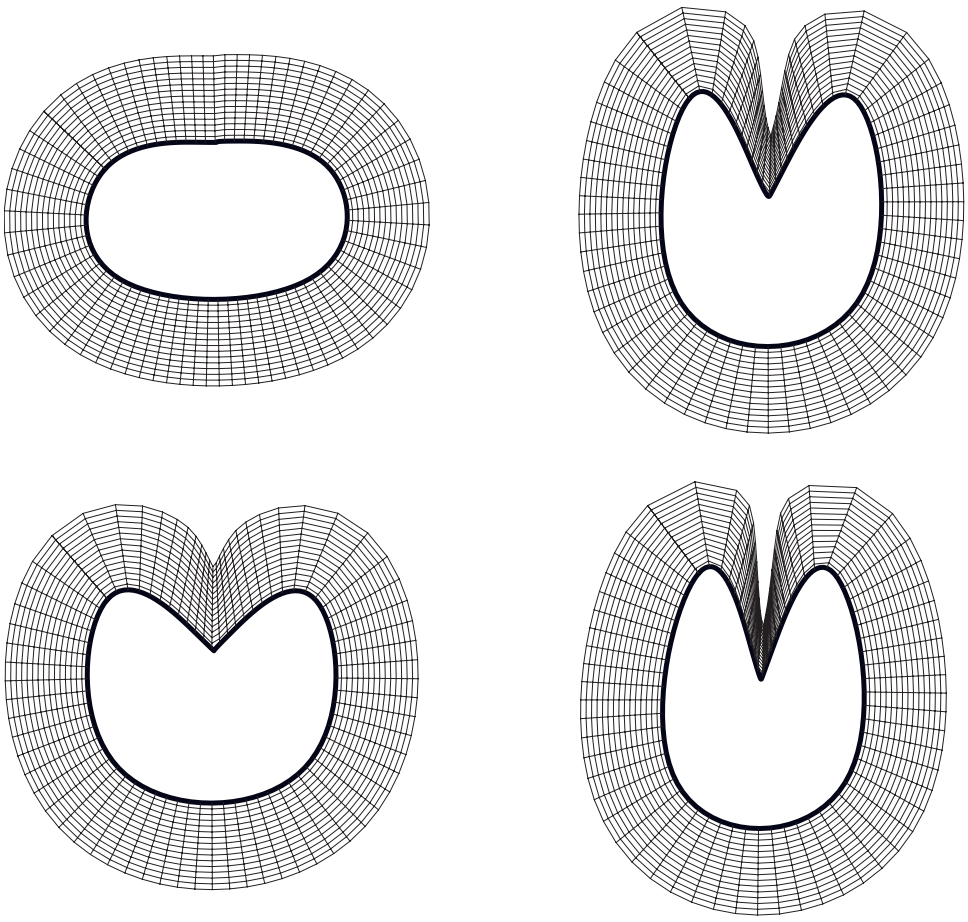

Figure 4.7: Self-adjusting shell behavior of an exterior shell in the concave region. With the angle up to 90 degrees, no compression is observed; after that the shell starts to compress.

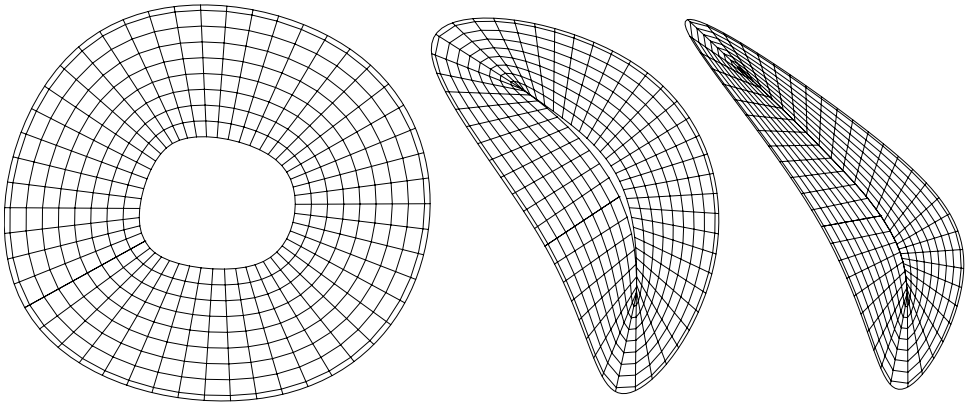

Figure 4.8: Interior shell. The first image shows interior shell for which prescribed thickness is achieved. As the object is deformed, the shell compresses to avoid folds (prescribed thickness remains the same).

4.3.1 Localization

The averaged distance function is supported over the whole surface. However, one can think by intuition that it does not make sense to take into account parts of the surface that are far away. If the distance between two surface patches is larger than double the target shell thickness, their generated shells will not intersect each other. Thus for every space point $\mathbf{x}$, we can only integrate over the parts of the surface that fit inside the ball of radius $2h$ and centered at $\mathbf{x}$. Here h is the target shell thickness. In our implementation, we extend the integration support to be a bit larger than $2h$ to ensure stability. This makes our calculation local. In addition to speeding up the calculation, this also makes the algorithm satisfy the locality condition, useful in geometric modeling applications: modifying a bounded region on a surface affects the surface only in a neighborhood of that region.

4.3.2 Boundaries

So far we have assumed that M does not have a boundary. Near the boundary, the averaged distance function is likely to yield shells with considerable distortion due to the fact that the gradient field has to make a 180-degree turn. The standard distance function handles this case well, but the averaged function gradient field turns to the outward direction. This problem is solved by adding artificial faces at the boundary. A single additional vertex is added for each boundary vertex. The direction to the new vertex is obtained by using a tangent direction across the boundary, and the distance is taken to be equal to the shell thickness. It should be noted that such extension is satisfactory if there are no other parts of the surface near the boundary. Otherwise, the extension can overlap a different surface part.

4.4 Numerical and performance considerations

There are two main difficulties in using the averaged distance function to construct shells: we need to solve the ODE which is stiff if the trajectory approaches the medial axis, and we need to compute the field gradient efficiently. While the ODE in most cases is well-behaved, the gradient direction and magnitude may change rapidly near the medial axis, resulting in a stiff system. In addition, the gradient, which is an integral over the surface, is expensive to evaluate. We have evaluated several solution techniques (variants of explicit and implicit Euler and Runge-Kutta methods) and obtained the best performance and stability using an adaptive explicit Euler method. This algorithm is given below in somewhat simplified form, where $\mathbf{x}_0$ is the starting point on the surface, $\mathbf{x}$ is the current position along the trajectory, g is the gradient of the averaged distance function at the point, h is the prescribed thickness, Δ is the

variable step size, and ϵ is the adaptivity threshold for the change in the direction.

$t = 0;$

$\mathbf{x} = \mathbf{x}_0;$

$g = Field(\mathbf{x}_0);$

while $t < h$

$\Delta = 2\Delta_0$**;**

do

$\Delta = \Delta/2;$

$\mathbf{x}_{new} = \mathbf{x} + \Delta * g$**;**

$g_{new} = Field(\mathbf{x}_{new});$

while angle between g and g_{new} is above ϵ**;**

$t+ = \Delta;$

$\mathbf{x} = \mathbf{x}_{new};$

$g = g_{new};$

end while

4.4.1 Computing integrals

The expense of computing the gradient can be considerable for an interactive application since it involves a surface integral.

The simplest approach to compute the integrals is to do point-wise summation over the surface. For each vertex on the mesh representing the base surface, we calculate the function to be integrated and weight the result value by a quarter of the area of the immediate neighborhood. Although we only integrate over a small part of the surface inside a ball near a given point, the integration still requires visiting all

vertices of the surface when implemented in a brute-force way.

We use the Barnes and Hut algorithm [2] to accelerate the computation of the integral. The vertices of the mesh and their weights are placed into the leaf nodes of an space-partitioning octree. A weighted centroid and a total weight are stored at each tree node. The centroid and weight of a non-leaf node are computed from all its children and the tree is built bottom-up. When computing the integral for a point of space, the tree is traversed top-down. For each node, we compute the ratio of the distance between our point and the centroid to the node size, and compare the ratio with a given threshold. If the ratio is below the threshold, the gradient field is computed using the centroid only and weighted by the weight of the node. This is considered a sufficiently close approximation. Otherwise, the gradient field for the node is computed recursively as the sum of the fields of its children. This algorithm takes very little effort to implement, and provided a substantial speedup.

4.4.2 An analytical approach

However, the point-wise summation approach does not work well in the case when the sampling is fixed and the surface may have very sharp angles. This is easy to understand if the sample points (vertices) are thought as point charges, replacing surface charge density. The approximate gradient field may "escape" between points when a surface region with high curvature is not sampled densely enough for numerical methods. This results in shell inversion. This "escape" problem can be avoided by integrating analytically over triangles of the surface mesh (quads can be split into triangles for this purpose). Fortunately, it is possible to analytically integrate $|\mathbf{x} - \mathbf{y}|^{-3}$ over a wedge, and a triangle can be represented as a complement of three different wedges in a plane, obtained by extending each triangle side in one direction (Fig-

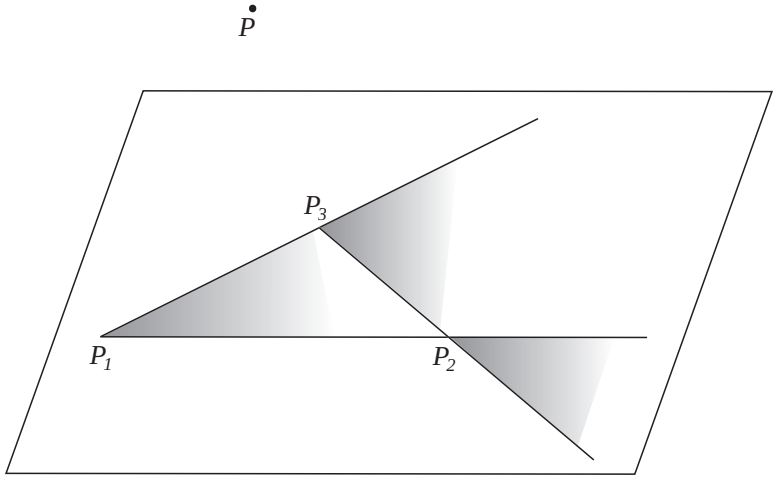

Figure 4.9: Decomposition of a triangle for integration. Integral on a triangle is decomposed into three integrals on wedges.

ure 4.9).

Over a single wedge, the integral of $|\mathbf{x}-\mathbf{y}|^{-3}$ can be computed explicitly. Without losing generality, assume that the origin is the vertex of the wedge and x coordinate axis is the starting edge. Then for $p = 3$, the integral in the averaged distance function is

$$\int_{\angle} |\mathbf{x} - \mathbf{y}|^{-3} d\mathbf{y} = \frac{2}{w} \arctan\left(\frac{w}{(|\mathbf{x}| - u)\cot\frac{\beta}{2} - v}\right) \tag{4.6}$$

where (u, v, w) is the coordinates of the point $\mathbf{x}$ and β is the counter-clockwise angle of the wedge $\angle$, as shown in Figure 4.10. This formula is then differentiated on u, v, and w for the calculation of gradient (refer to Appendix A for detail). Using this simple formulas the integral over the mesh can be evaluated *precisely* if desired.

The computation of formula (4.6) and its gradient over the whole surface can get very expensive. Space partition techniques we used in the Barnes and Hut algorithm cannot be applied to triangles directly, because a triangle may not fit in a single node.

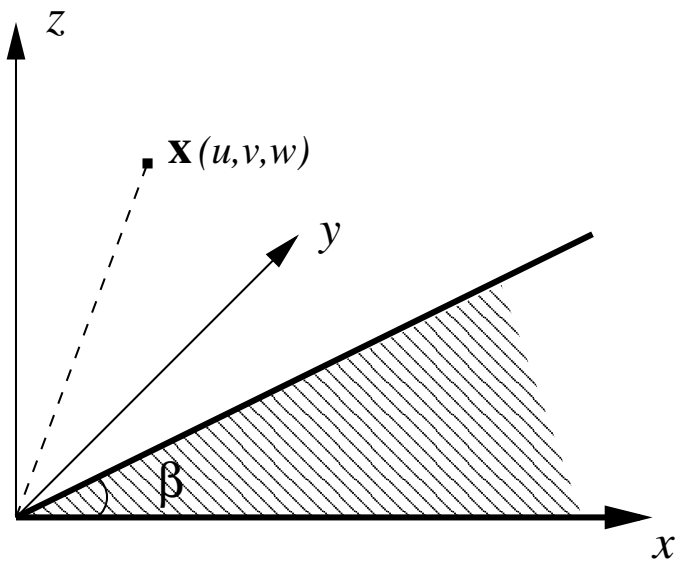

Figure 4.10: Analytical integration of the function $|\mathbf{x}-\mathbf{y}|^{-3}$ over a wedge. The result is shown in Equation (4.6).

We use an analogue of the Barnes and Hut algorithm to speed it up. For each surface patch, we compute the ratio of the distance between point $\mathbf{x}$ and the centroid of the patch to the size of the bounding box of the patch. If the ratio is below some threshold, the integral is computed using only the centroid and weighted by the area of the patch. Otherwise, the patch is subdivided when it has more than one faces; or the integral is computed using the analytic formula when the patch is a single face. The centroids and areas of patches can be pre-computed and organized in a tree. While computing the gradient field in this way is more expensive than the point-wise summation approach, this eliminates the need for refinement. In fact, using a coarser resolution version of the mesh yields good results.

4.5 Examples

Several examples of external and internal shells and textures are shown in Figure 4.11-4.13, Figure 5.7 and Figure 7.2-7.8. The timings for simpler objects were fractions of a second. For the bunny mesh shown in Figure 4.12, the external shell was generated in 1.8 sec on a 1GHz Pentium III, and the internal shell in 8 sec. The longer time for the internal mesh is due to refinement necessary to compute a valid shell inside the ears. The target thickness for the exterior shell was set at 10% of the bounding box size, and at 5% for the interior shell.

4.6 Limitations of the algorithm

Even with the optimizations described above, the algorithm is still relatively computationally expensive. Clearly, one expects better performance in cases when the exterior shell is constructed for a convex object, when simple normal displacement suffices. In this case our algorithm produces nearly identical results, but at much higher computational expense. Ideally, normal displacements should be used whenever possible, and our approach should be applied only in places where it is truly necessary. The problem is that identifying those trouble region of the normal displacement method is not trivial.

The resulting shell is not guaranteed to be one-to-one; this is essentially inevitable as we require directors to be straight. However, as shown in Figure 4.14 a rather large shell thickness needs to be prescribed for a special type of geometry for the failure to occur; for this figure the requested shell thickness was close to the size of the bounding box of the whole object.

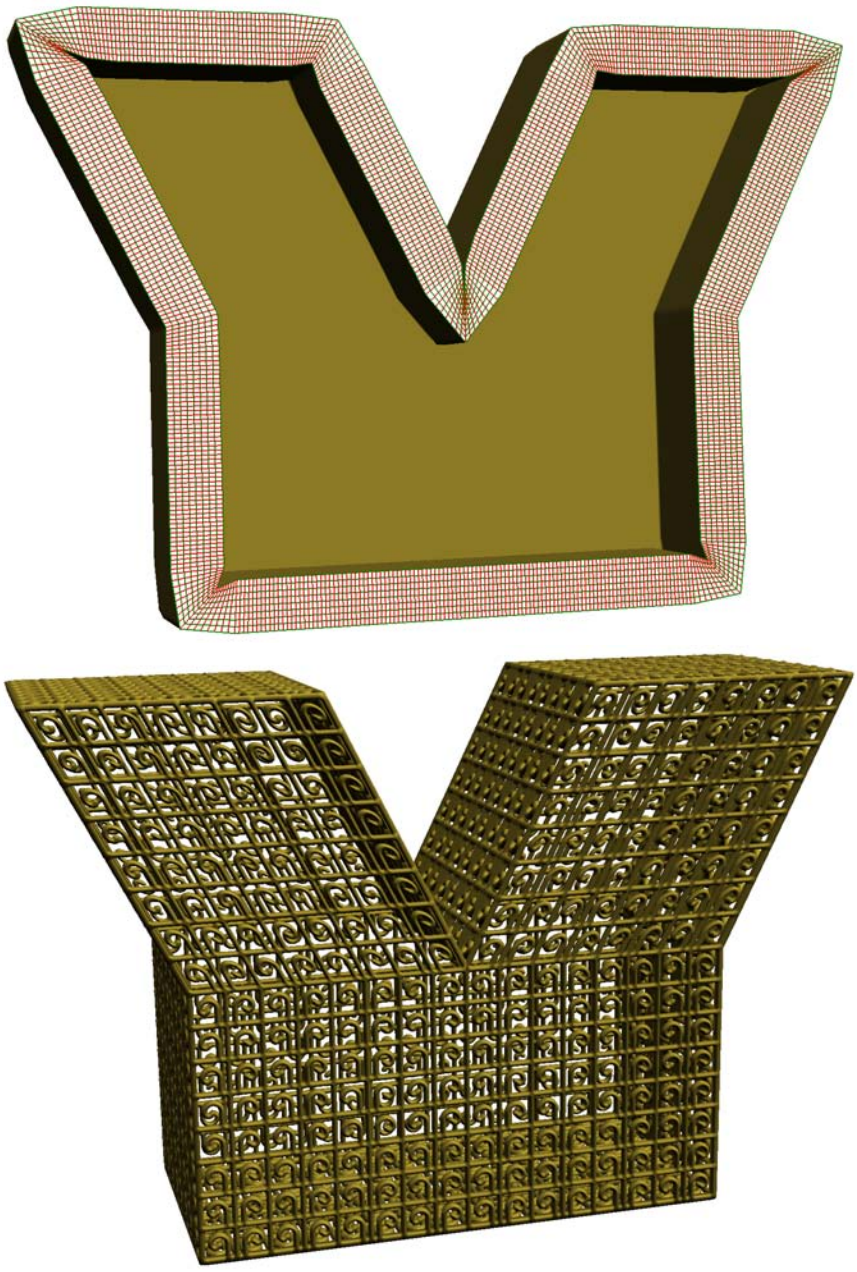

Figure 4.11: Shells for sharp features. Upper: cross-section of the shell for a shape with sharp corners; lower: same object with volumetric texture added.

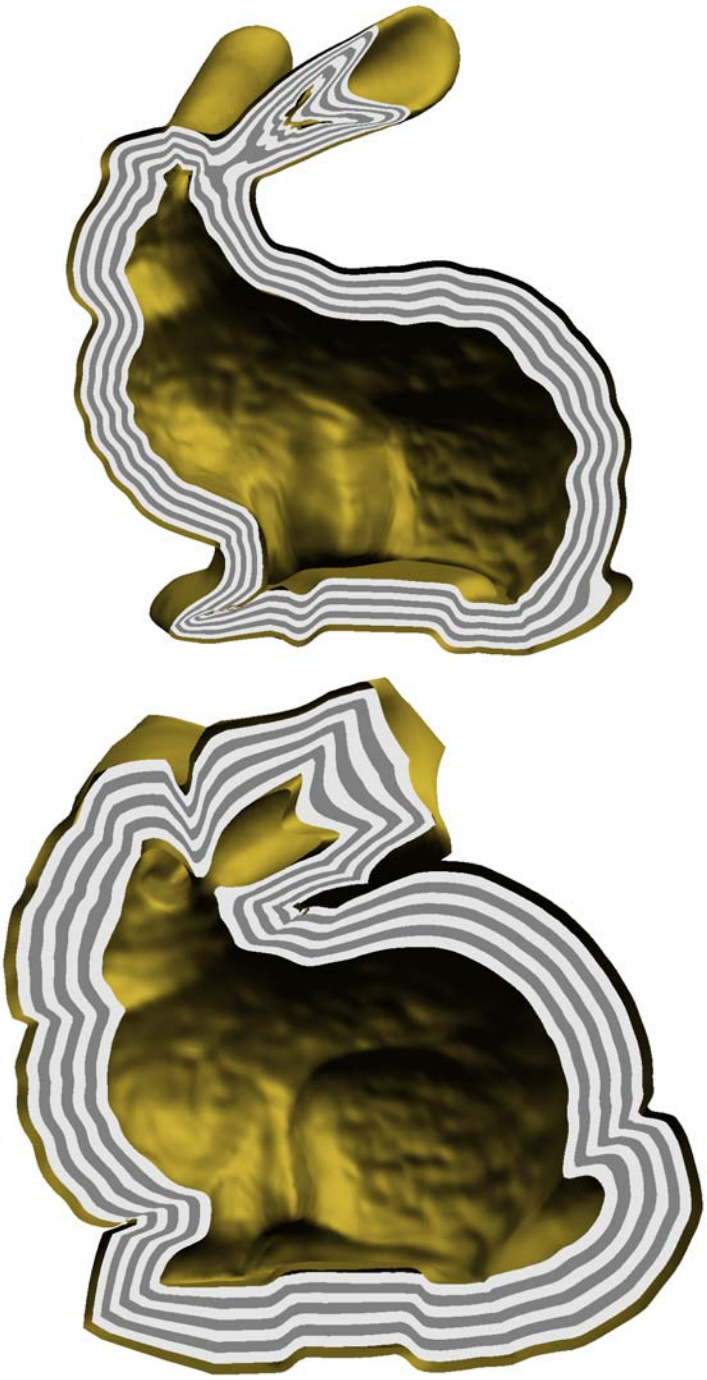

Figure 4.12: A cross-section of the interior and exterior shells for the bunny.

Figure 4.13: A kiosk and zoomed-in detail. Volume textures are used on the roof and walls.

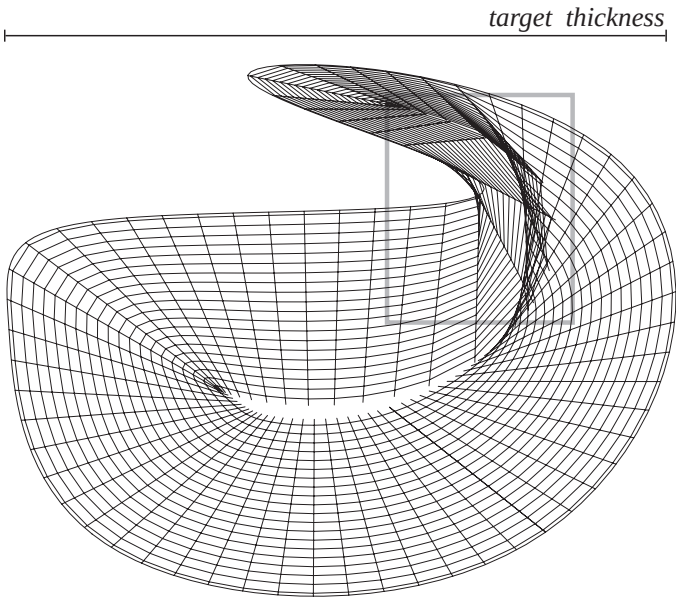

Figure 4.14: Folding for extreme shell thickness (prescribed thickness equal to the objects bounding box size, only 70% of the shell shown to show the fold clearly.)

4.7 Comparison

Figure 4.15 shows the shells generated by normal displacement and our method on a saddle mesh. Normal displacement method inevitably generates self-intersecting shell near the saddle point no matter which side of the mesh the shell goes. In the pictures, we only show the shell located at the upper side for clarity.

Level set methods, the fast marching method in particular, present the main alternative to our approach. However, the level set methods do not solve the problem of shell construction directly. Both the original surface and the constructed front are defined implicitly. The method does not readily provide any mapping from the original surface to the advancing front, and the topology of the front may change. This is in fact an advantage for many applications but makes shell construction difficult. An

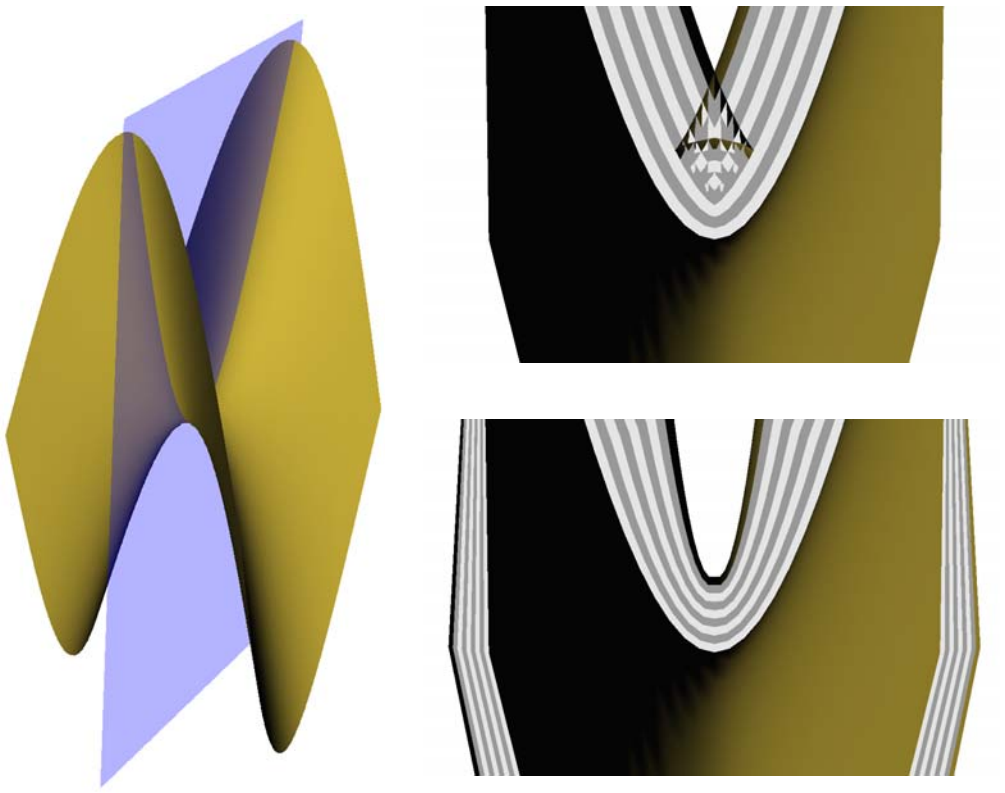

Figure 4.15: Comparison of the results for normal displacement (upper right) and our method (lower right) for a saddle. The left picture is the saddle mesh with the clipping plane shown in blue. Cross sections of the shells at the clipping plane are shown at right.

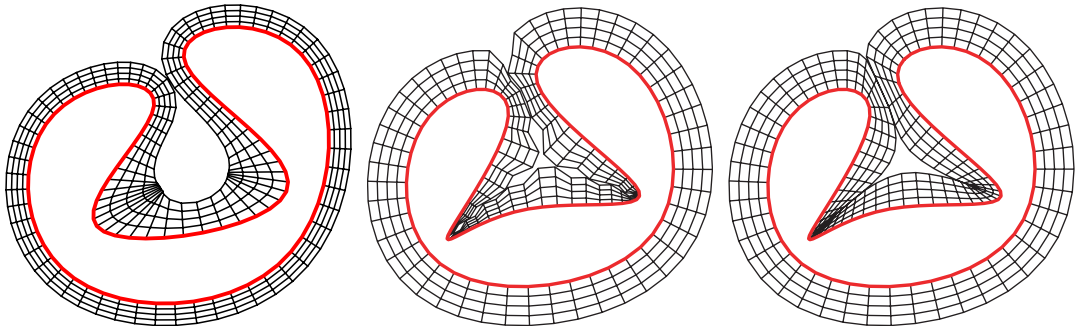

Figure 4.16: Exterior shells constructed by fast marching method, Cohen's method (envelope shell) and our method from left to right. Two dimensional shells are shown for clarity. The base curve is shown in thick red, all of which are the inner boundary of the shells.

additional step is required to establish the correspondence, as described in [39].

Another alternative is the envelope construction [13] which preserves the topology of the original surface. We have explored this approach and found that the thickness of such envelopes is very low in the regions of concavities, and the shape of the surface of the envelope tends to be undesirable in such areas.

Shells constructed by different methods are shown in Figure 4.16 for comparison. From the figure, we can see that our method generates nice shaped shell in concave area and avoids self intersection when two parts of the curve are close. In high curvature concave area, Cohen's method heavily compresses the shell while fast marching method tends to expand it. Cohen et al. only use the shell for error bounding, so the visual quality of the shell is less important. The shell by fast marching method is only half in thickness than the other two. Fast marching method does not work for the proposed thickness due to change of the topology of the marching front.

Finally we note that [46] uses a conceptually similar (although numerically quite

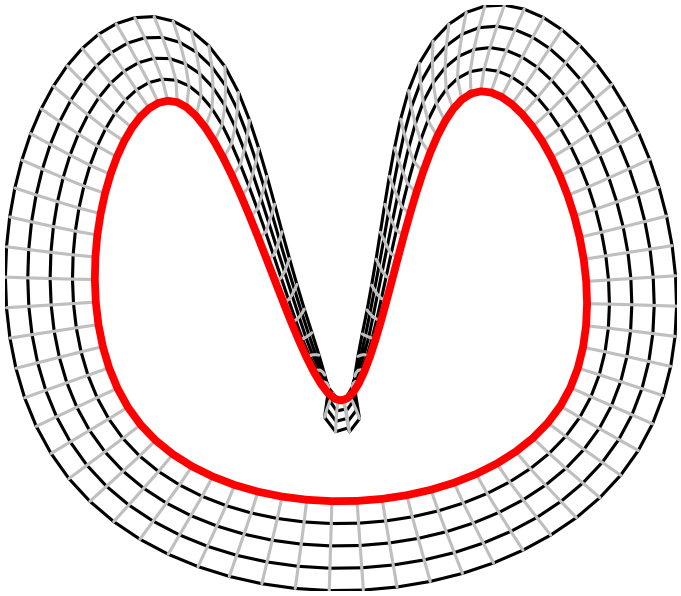

Figure 4.17: The shell generated by the lower bound value of p. In the tight concave region, the averaged distance function gradient field turns back into the surface, resulting in a flipped shell.

different) approach for constructing correspondences between surfaces, given that computing our integrals over the whole surface for $p = 1$ corresponds to solving the Laplace equation using Poisson formula. As we have pointed out, the value $p = 1$ does not work for constructing shells (Figure 4.17).

Chapter 5

Rendering

The marching cubes method [27, 8] is used in a lot of volume modeling applications for both ISO-surface extraction and volume rendering. It is simple to implement and sufficient to produce good images. However, marching cubes requires significant processing and results in large number of triangles. For a surface mapped with volumetric textures, the number of triangles produced by marching cubes easily grows into millions. While the surface is interactively changed, most of today's system cannot keep up processing and rendering such a large mesh. Ray tracing is another method to produce high quality rendering of volumetric-textured surfaces. It involves a lot of CPU bound computation, and is usually used for off-line rendering due to the high CPU usage and long rendering time. Interactive ray-tracing is often considered to be difficult for complex objects or scenes.

Slice-based rendering is another popular method in volume modeling applications. For an object that is represented implicitly as the ISO-surface of sample values in a volume, it renders slices through the volume and uses the sample values as alpha texture. Alpha test is enabled to trim the part of the slices that are out of the object.

This chapter presents our algorithm for rendering a surface represented by a volumetric texture. It is extended from the slice-based direct volume rendering techniques. This slice-based rendering algorithm maps perfectly to the current graphics hardware, and overcomes the major problem of conventional slice-based rendering, which is the normal discontinuities across slice boundaries.

This part of the work was done in collaboration with Daniel Kristjansson.

5.1 Slice-based direct ISO-surface rendering

Consider a single quadrilateral face of the surface. The part of the shell on the top of this surface is a hexahedron (Figure 5.1), with texture coordinates assigned to all vertices. Our surface inside the hexahedron is the ISO-surface of the function encoded in the alpha channel of the texture. We assume that the other channels contain the texture gradient. Recall that the normal to an ISO-surface defined by $\alpha(x, y, z) = const$ is $\nabla\alpha$; therefore the gradient texture can be used for shading. It should be noted that the gradient in the light direction can be computed on the fly as suggested in [43]. However, this approach works only if the gradient has unit magnitude (we need the unit normal for lighting), which places considerable restrictions on the way α can vary.

The idea of the slice-based direct volume rendering is to render polygons intersecting the 3D textured volume and assign texture space vertex positions as texture coordinates. The alpha test is used to discard the part of each slice that is outside the object, and the gradient texture is used for shading. One of the problems with this approach is that there is a normal discontinuity at the boundary of slice images (Figure 5.2). This means that many slices are needed to achieve smooth shading. A

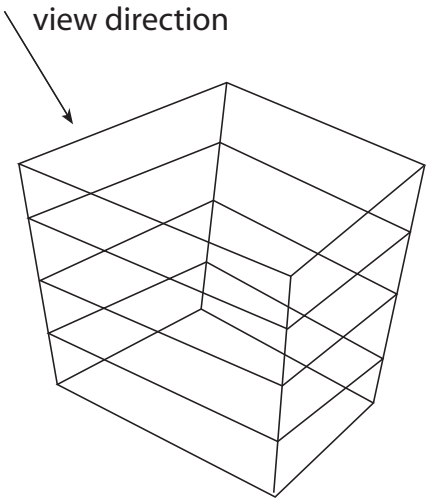

Figure 5.1: Slices in a single texture hexahedron. Slices are oriented in the direction closest to the view direction.

large number of slices is affordable when a single volume is rendered. In fact, several slices per 3D texture layer are often used; this still amounts to rendering only few thousand slices, as the volumetric texture size rarely exceeds 512 in any direction.

The situation is different for volumetric-textured surfaces; for a curved surface, a large number of polygons must be rendered for each surface layer, so rendering multiple slices per pixel is often not feasible.

Our algorithm addresses this problem by ensuring that the normal varies smoothly across slice boundaries, even if the distance between slices is considerable. This allows us to reduce the number of needed slices by a large factor.

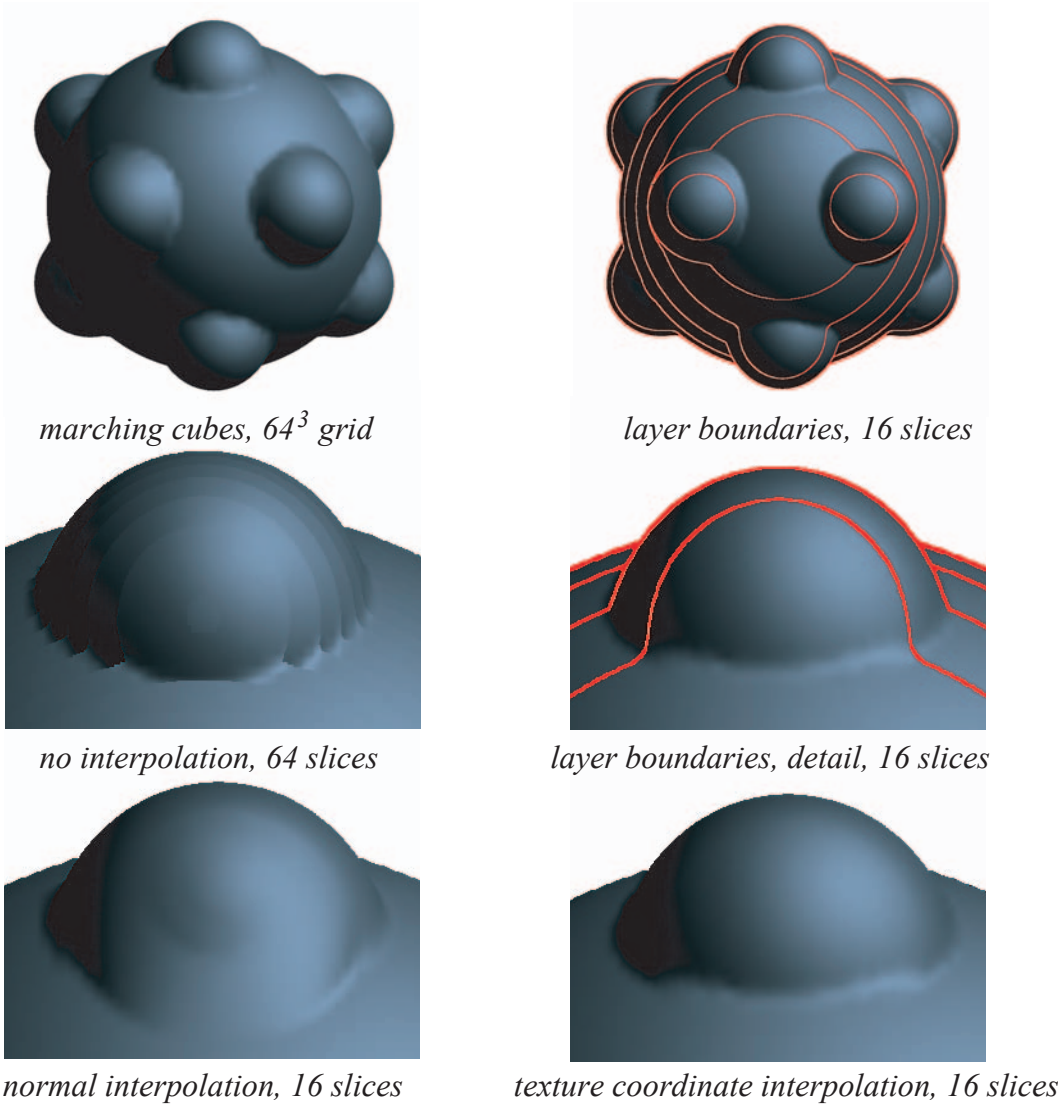

Figure 5.2: Comparison of quality for rendering methods for a single volumetric texture. The texture is computed as the distance field for a large sphere with smaller spheres attached at vertices of a regular icosahedron. The number of slices in slice-based methods is chosen so that the total time required to render an 800x800 image is approximately the same for each method.

5.2 The idea of the algorithm

The normal discontinuity at pixels near the slice boundaries for slice-based rendering method is resulted from the discontinuous texture coordinates, which jump from one slice to another at slice boundaries. In our algorithm, we eliminate this artifact by generating continuous texture coordinates across slice boundaries. Thus the problem is solved and we are able to use very few slices without artifacts.

The idea of the algorithm is illustrated in Figure 5.3. It is somewhat similar to the approach that was used in [43] for volume rendering of unstructured meshes.

Suppose that for each pixel, we know the texture coordinates t_1 corresponding to the first slice along the view direction, for which $\alpha \geq 0.5$ (i.e. the point of the slice is inside the object, represented by the upper black concrete dot), and the texture coordinates t_2 for the *last* slice along the view direction for which $\alpha < 0.5$ (the gray dot in Figure 5.3). In this case, we can approximate the point where $\alpha = 0.5$ by interpolating between two points, using α_1 and α_2 as weights. We interpolate the texture coordinates t_1 and t_2 using the same ratio, and look up the normal using the interpolated coordinates. The interpolated coordinates change smoothly as we pass the slice image boundary, and the normal obtained in this way is considerably more accurate. This process can be regarded as bump mapping for ISO-surfaces.

Conceptually, our algorithm proceeds in two passes. An actual implementation may be more efficient if the number of passes is increased as discussed below.

5.3 Algorithm details

Suppose the top face of the hexahedron is a quad $P = [p_0, p_1, p_2, p_3]$ and the bottom face is $Q = [q_0, q_1, q_2, q_3]$. We assume that the normals of any two opposite faces

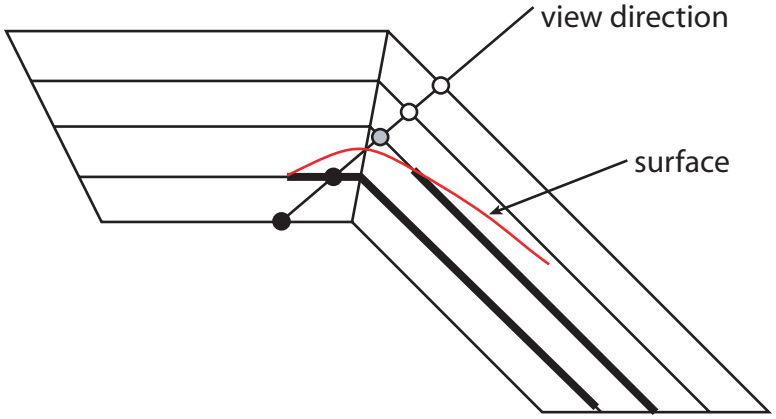

Figure 5.3: Slice interpolation. Our algorithm interpolates between the slice texture coordinates along the view direction of the last slice outside the surface and the first slice inside the surface.

are sufficiently close. Although nothing in the method relies on this assumption, for highly distorted hexahedron the quality of the result will be low. We assume that the view direction is close to the normals of both faces P and Q. We render a set of M slices S_m where $S_0 = P$ and $S_{M-1} = Q$. The vertices p_j^m ($j = 0 \ldots 3$) of the m-th slice are given by

$$p_j^m = \left(1 - \frac{m}{M}\right) p_j + \left(\frac{m}{M}\right) q_j$$

The texture coordinates are interpolated in a similar way. In addition to rendering the slices in the direction closest to the view direction, we also render the faces of the hexahedron to ensure correct texture coordinate interpolation as explained below. We assume that the top and bottom slices have alpha values below the threshold at all points; otherwise we render additional slices in front and back.

5.3.1 First pass

This pass is nearly identical to the simplest version of slice-based rendering. The difference is that the output of this pass is two textures. The first output texture T_1 contains the 3D texture coordinates of the closest fragments with $\alpha \geq 0.5$, and is of the same size as the output image. The second texture Z_1 of the same size has the depth values for rendered fragments.

The alpha test on this pass rejects the fragments with $\alpha < 0.5$ (outside the object); the standard depth test is performed.

5.3.2 Second pass

In this pass, we need to perform somewhat more complicated operations which require hardware programmability. The textures from the first pass are applied to the rendered fragments, using the view-port coordinates (x, y) as the texture coordinates. The fragment is discarded this time if $\alpha \geq 0.5$, i.e. it is inside the object. For fragments with $\alpha < 0.5$, we compare the fragment z value with the depth texture value $Z_1(x, y)$ (the z value for the closest fragment inside the ISO-surface). A fragment is discarded if it is behind the closest fragment inside the ISO-surface (Figure 5.4).

If the fragment passed all tests, we look up the alpha value α_1 for the current fragment, and the alpha value for the closest visible fragment at this point using T_1. $\alpha_1 = \alpha(T_1(x, y))$. The texture coordinates t_2 of the current fragment and texture coordinates $t_1 = T_1(x, y)$ of the visible fragment at the same pixel are interpolated to compute the new texture coordinates

$$t_3 = ((\alpha_1 - 0.5)t_2 - (\alpha_2 - 0.5)t_1)/(\alpha_1 - \alpha_2).$$

The resulting coordinates are used to look up the gradient for lighting. We note that

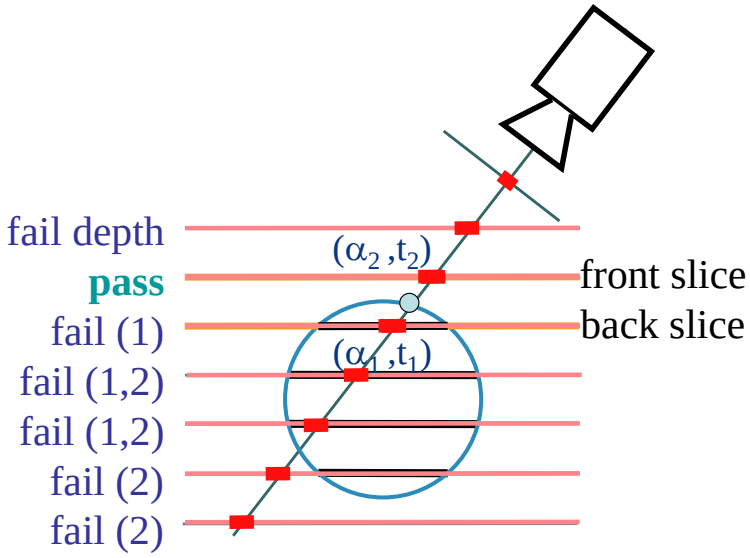

Figure 5.4: The second pass of our rendering algorithm. Here, besides the reversed depth test, the other two tests for a fragment are: (1) $\alpha < 0.5$; and (2) current depth is less than the depth value from the first pass. In the picture, t_1, t_2 are texture coordinates.

the gradient of α is computed in texture coordinates and needs to be transformed to eye coordinates. Because the hexahedron is a trilinearly distorted cube, the texture-to-eye coordinates change from texel to texel. Instead of recomputing the transformation, we compute it at the vertices and use trilinear interpolation in the interior. This approach works unless the distortion is high.

Most importantly, the depth test on this pass is reversed. This means that the fragment that is actually rendered, is the furthest fragment from the eye which is both outside the ISO-surface and passes the comparison with the depth value from the first pass.

Note that in the second step we assume that the texture coordinates from the first pass are taken from the same texture, so it makes sense to interpolate t_1 and t_2. This is ensured by rendering the boundaries between textures twice, with a small offset towards each of the textures.

Finally we observe that for smoothly varying textures or a number of slices equal to the texture dimension, normal interpolation can be used instead of texture coordinate interpolation.

5.4 Implementation

The algorithm was implemented using GeForceFX fragment programs.

The theoretical first pass produces two textures: the texture coordinates texture and the depth value texture. Although the depth buffer on the video card is of high precision, the video card does not give the full 24-bit depth values while binding a texture to the depth buffer due to the limitation of the GeForceFX driver. Only the 8 most significant bits are present in the output. This precision is insufficient for depth

tests in most graphics applications. If we use this low-precision depth texture in the second pass, we end up with comparing the high-precision depth values generated in the second pass with the 8-bit one from the first pass. The lack in precision causes the incorrect fragments to be picked by the reversed depth test and results in unacceptable artifacts. To work around this problem, we capture the depth values from the first pass in a separate pass, which writes the 24-bit depth values to a 32-bit RGBA buffer.

We also split the second pass into two: one to generate the 3D texture coordinate we need, followed by a final pass to render the image. The final pass uses the two 3D texture coordinates per pixel from the previous passes to look up the gradient, generates a normal and calculates the lighting. By using this extra pass we only need to perform the lighting and interpolation of texture coordinates for visible slices. The final lighting pass also does not require geometry, so it is relatively inexpensive in terms of AGP bus bandwidth and results in better performance (Figure 5.5).

5.5 Performance

The performance of the rendering algorithm is very satisfactory for interactive applications. Especially for large textures, our algorithm shows substantial performance improvement over other algorithms with similar rendering quality. We rendered a turbine blade model using 128 slices through a 512x512x512 texture (compressed to 134 MB) and the system exhibits real-time performance (Figure 5.6). With 512 slices, the quality is slightly greater, and the rendering time is still acceptable for interactive tasks.

On the other hand, when we try to stress geometric complexity, we run into performance limitations because of the large number of faces with volumetric textures

attached. For example, with the shirt in Figure 1.1 and Figure 7.2, we are limited to about 16 slices while still obtaining close to real-time performance (17fps with either normal or texture coordinate interpolation). The bunnies shown in Figure 5.7 are rendered in similar speed to the shirt.

The performance was observed on a Quadro 3000 clocked at the standard 400/850 Mhz for the core and memory respectively.

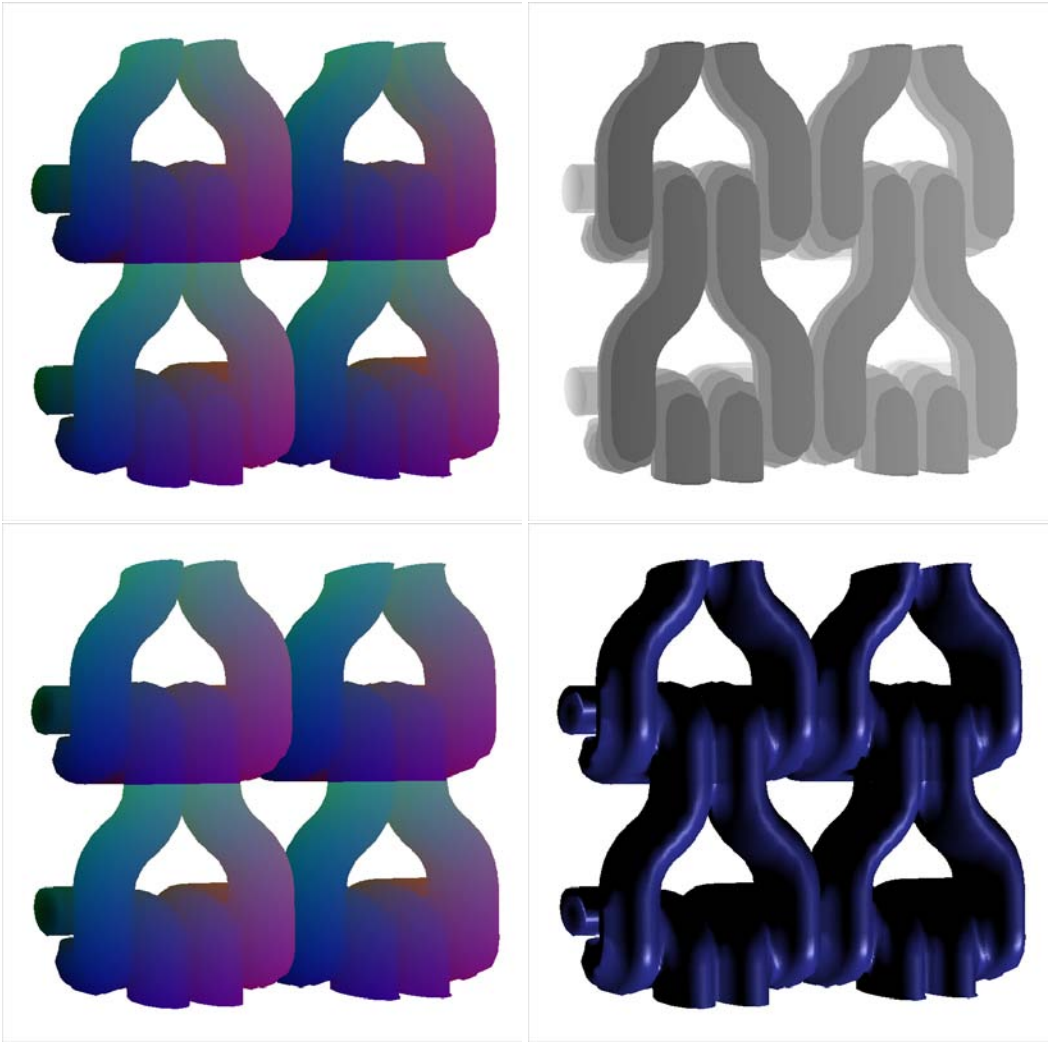

Figure 5.5: Steps of the rendering algorithm. From left to right: Texture coordinates and depth from the first pass on the first row; Interpolated texture coordinates of the second pass and final image on the second row.

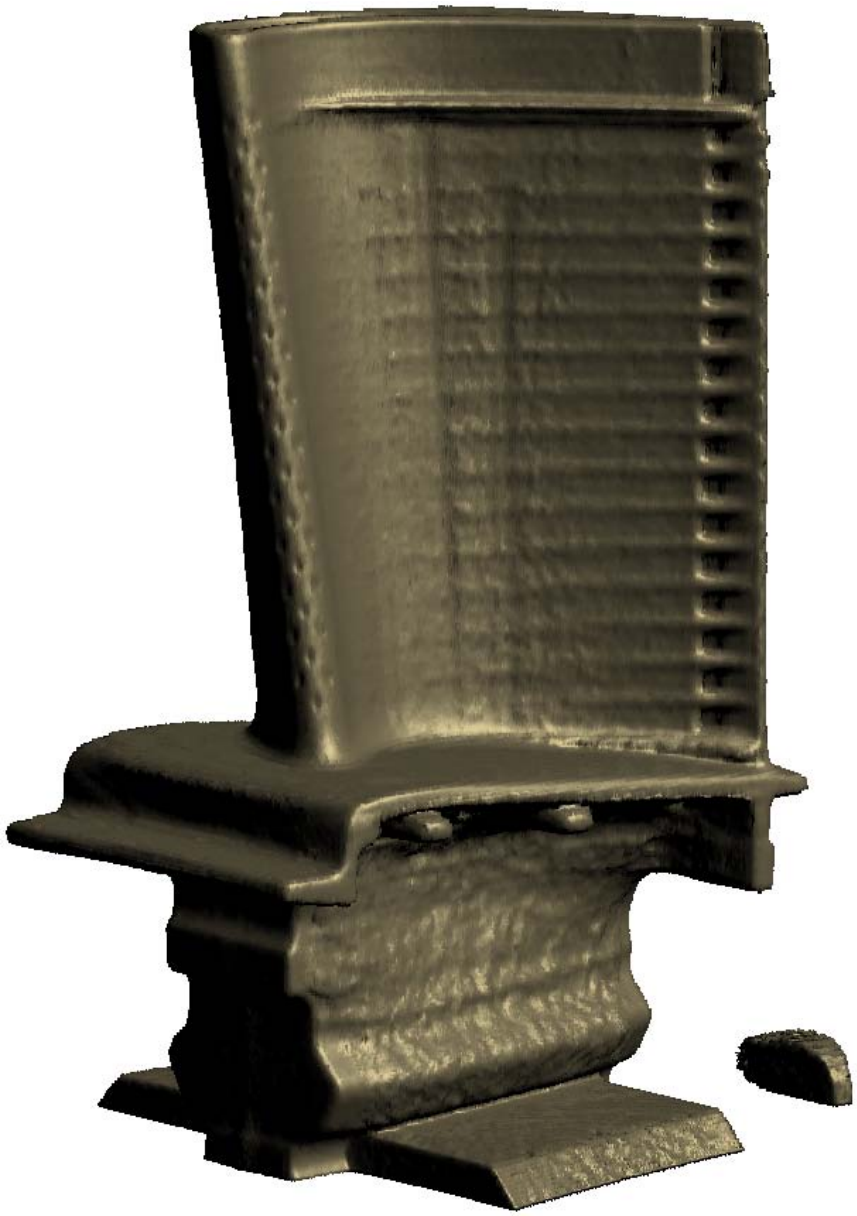

Figure 5.6: The turbine blade model rendered interactively by our algorithm. With compression, we fitted this excessively large texture in the video memory and rendered the model with real-time performance.

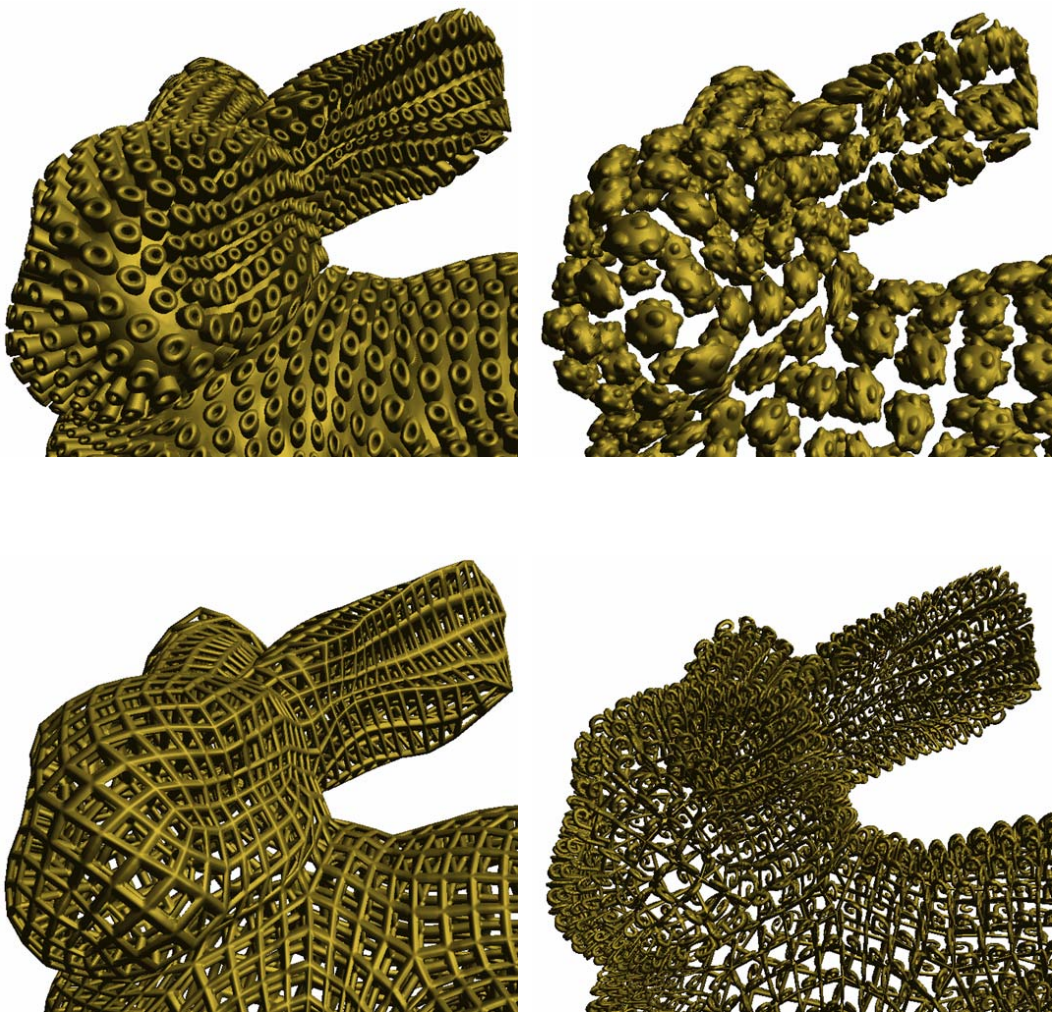

Figure 5.7: Several different volumetric textures applied on the bunny. Only the heads are shown to view the geometric details.

Chapter 6

Mipmaps

Mipmaping technique was introduced by Williams in 1983 [44] for filtering image textures. It filters an image texture into multiple levels of different resolutions. When a mipmaped image texture is rendered, one or more precomputed mipmap levels, whose pixel size is near the size of a screen pixel in the texture, are picked to retrieve the texture content. Mipmaping is an effective method to eliminate aliasing in texture mapping. It can also be applied to volumetric textures with proper definition of how to filter the voxel contents.

6.1 Volumetric textures

For the volumetric textures we discussed in this dissertation, each voxel contains a normal vector n and an alpha value α. We use the volumetric textures as *occupancy textures*, which means that an alpha value α specifies whether a voxel is inside the implicitly represented object or not. This is determined by a boundary value α_0. If $\alpha \geq \alpha_0$, the voxel is inside the object; otherwise it is outside. Usually the object only

contains part of the texture. On the other hand, one can define a volumetric texture to be a *transparency texture*, whose alpha value states how transparent a voxel is. The object represented by a transparency texture contains the whole texture, even some voxels may have their $\alpha = 0$ and be fully transparent.

6.2 Linear filtering

The classical and simplest approach to filter textures for mipmaping is linear interpolation. In the case of volumetric textures, α at voxel (i, j, k) in the filtered texture is set to be the averaged α of the 8 voxels whose indexes are in the range of $(2i \ldots 2i + 1, 2j \ldots 2j + 1, 2k \ldots 2k + 1)$ in the original texture. This method keeps both the occupancy and transparency interpretation valid for the filtered texture. However, it usually makes the alpha values in a filtered occupancy texture below the boundary value when the texture is sparsely occupied. This process shrinks or even vanishes the object gradually in mipmap levels of lower resolution (Figure 6.1).

6.3 Double mipmaping

6.3.1 Motivation

In the real world, there are a lot of objects similar to the one shown in Figure 6.1, which has a lot of holes. Examples include window screens, chain-link fence, etc. If you look at such a object from a large distance where it is impossible to see the structural details, the object will look like a semi-transparent mask and the objects behind it appear dimmer. The transparency of this imaginary mask depends on how much light transmits through the object, and can be approximated by the ratio of the

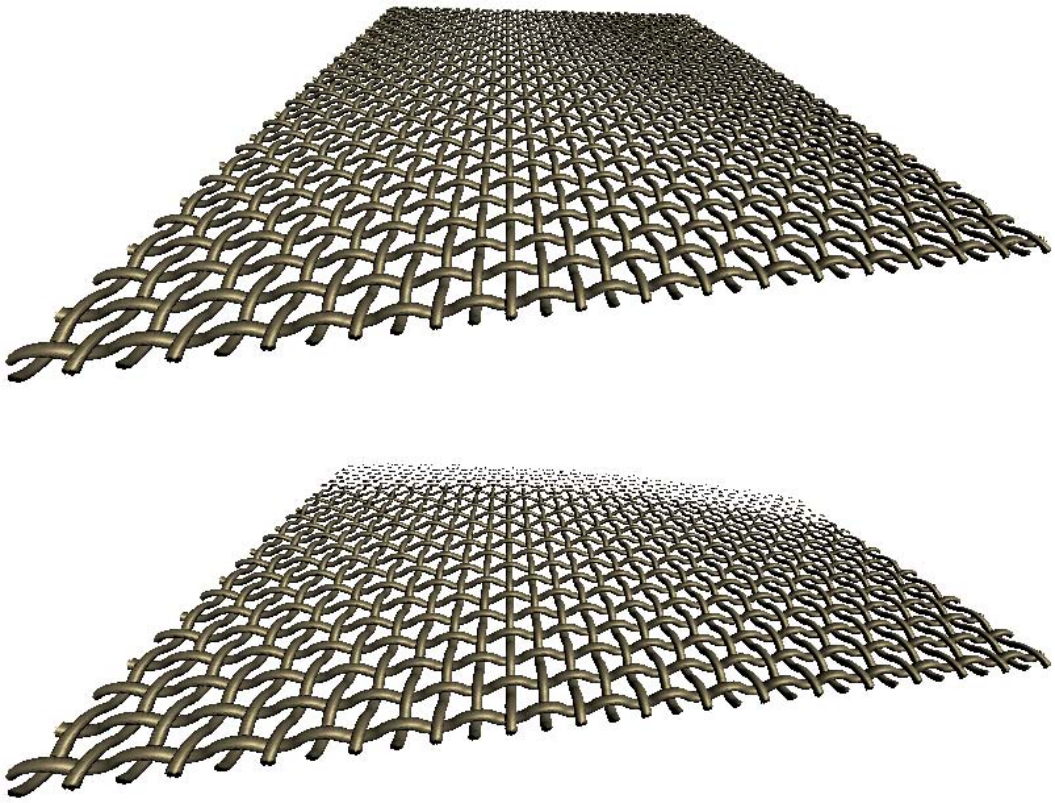

Figure 6.1: Simple linear interpolation filtering vanishes a volumetric texture in mipmaping. The upper picture does not use mipmap and aliasing shows in the far part. The lower picture uses mipmap built by simple linear interpolation filtering and the object becomes sparser while it is further away and finally disappears. α in the original texture is between 0 and 1, and the boundary value $\alpha_0 = 0.5$.

see-through part to the whole object.

This observation invokes a solution to the vanishing problem in volumetric texture mipmap. The original texture is an occupancy texture and represents a geometric object. It is rendered using alpha test as described in Chapter 5. We filter the texture to build a mipmap by the linear interpolation method discussed above. When the object starts vanishing or shrinking in the filtered texture at some level, the filtered textures can be interpreted as a transparency texture instead of an occupancy texture from that level. The transparency texture is then rendered using alpha blending, without alpha test. This double interpretation of the mipmap should imitate the real world observation closely.

6.3.2 Algorithm overview

The double-mipmaping approach uses two mipmaped textures, a high-resolution occupancy texture to represent the detailed geometry and a low-resolution transparency texture to generate the transparent effects. The object is rendered twice, first using the occupancy texture to render the part near to the view point, then the transparency texture for the further part. The two textures are carefully tailored to make the transition from geometry to transparency smooth. In the transparency texture, we encode the gradient using a reflectance model instead of a single vector to amend the low resolution.

6.3.3 Texture representation

In the case of occupancy textures, rendering a volumetric-textured surface by slices involves using alpha test to discard the part outside of the object. Furthermore, we

interpolate texture coordinates or normals to fix the normal discontinuity at slice boundaries as described in Chapter 5. Rendering a surface mapped with a volumetric transparency texture is simpler. The transparency is implemented by alpha blending. There is no alpha test involved, thus no sharp slice boundary or shading discontinuity. However, rendering a few mipmap levels as occupancy textures and the other levels as transparency textures is not feasible on current graphics hardware.

We use two mipmaped textures and render the object in two passes to accomplish this combined geometry-transparency mipmap approach. The first texture is an occupancy texture representing a geometric object. Its mipmap levels only contains the original full resolution texture and those filtered levels before the object starts shrinking or vanishing. The second texture is rendered as a transparency texture, whose mipmap contains the filtered levels left by the first texture. The two textures have one overlapping mipmap level to provide smooth transition between them. An extra empty level ($\alpha = 0$ for the whole level) is added to the end of the occupancy texture and the start of the transparency texture respectively. These two empty mipmap levels prevent the two textures from being applied to areas where they are not desired (Figure 6.2).

6.4 Reflectance model

6.4.1 Limitation of normal textures

Every voxel in our volumetric textures contains a normal vector n and an alpha value α. The alpha value indicates whether a voxel is inside the object or not, and is used by the renderer to discard pixels outside the object. The normal vector is used for

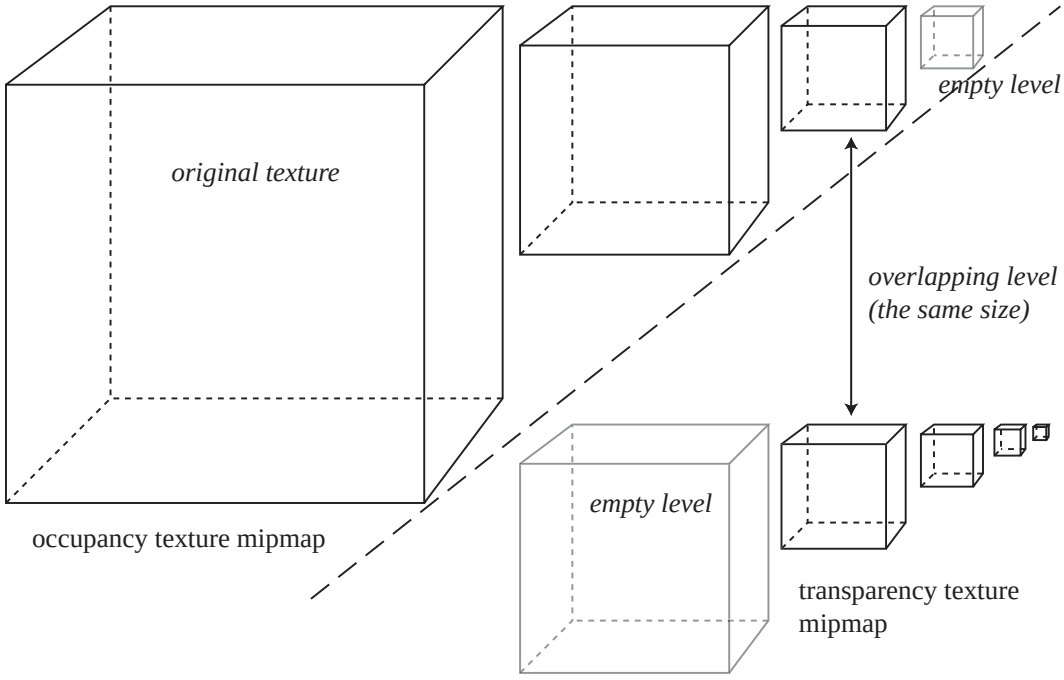

Figure 6.2: Structure of the double mipmaping. The occupancy texture mipmap contains the high resolution levels and the transparency texture mipmap contains the low resolution levels. They have one overlapping level of the same size. An empty level ($\alpha = 0$ for the whole level, shown as gray cubes in the picture) of the right size is placed in each mipmap.

shading the object. When an object represented by volumetric textures is sufficiently sampled, each voxel contains only a small piece of the object. If a voxel intersects the surface of the object, we can simplify the shading problem by assuming that the small piece of the surface is a plane. Thus the normal vector at each voxel provides enough information to reconstruct the local shape of the object. Combined with global material properties, we can apply the Phong illumination model for shading.

Problems arise while volumetric textures are substantially down-sampled. Voxels get larger in lower resolution texture levels and contain more of the object. When a piece of the object surface in a single voxel cannot be safely assumed to be a plane, one normal per voxel is not enough to describe the local shape of the object any more. Shading inconsistency can occur across the same object when different parts are mapped with different mipmap levels or the object is moving back and forth in a scene. This problem can be easily noticed when a volumetric texture contains two diffusive objects of the same shape, but one of them is smooth, and the other is rough with a lot of small geometrical details. Their normal textures will be dramatically different, and so are their appearances. After several down-sampling steps, both objects will become smooth, with their down-sampled normal textures being very close. Thus they will appear to be of the same brightness when rendered with the down-sampled texture. But a rough diffusive object tends to appear dimmer than a smooth one when both are illuminated with the same light. The normals in any small part of the rough object point to different directions. The light is then scattered around (Figure 6.3), which makes the object appear darker. This effect can only be correctly captured when the rough objects are rendered with a sufficiently high-resolution volumetric texture.

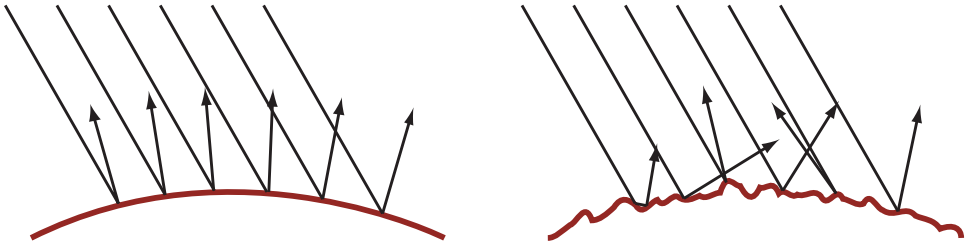

Figure 6.3: Lights reflecting on a smooth surface and a rough surface. On a rough surface, incoming lights are reflected to random directions. Some lights may be reflected several times between small features before leaving the surface and viewed by the camera.

6.4.2 Normal distribution model

The problem usually grows clearly noticeable with the transparency textures, which mainly consist of low resolution mipmap levels. Kajiya suggested to use a reflectance model to model details in distant view [22]. The reflectance can be used to represent geometrical details smaller than a voxel. Neyret presented a method using an ellipsoid as the geometrical primitive to encode the normal distribution in a voxel [30]. The ellipsoid is picked such that its normal distribution function (NDF) represents the real NDF in a voxel. During rendering, the Phong illumination model is integrated over the ellipsoid, using the normal of the ellipsoid as the normal of the represented object. In this representation, the position and scale of the ellipsoid does not affect the NDF. Thus, the oriented ellipsoid in each voxel has only six free parameters. Neyret used this scheme for efficient storage. Only are the two smallest ellipsoid axes stored in their volumetric texture. The third one is defined to be orthogonal to the two stored and has unit length. The object is rendered by ray tracing.

We adopt the method of encoding the NDF in a voxel by an ellipsoid in the

transparency texture. The current programmable graphics hardware lets people write vertex programs and fragment programs for the vertex and fragment processing units to customize their action in the rendering pipeline. This makes it possible to develop hardware-accelerated rendering algorithm for objects represented by volumetric-texture.

6.4.3 Ellipsoid representation and filtering

To fully utilize the power of the texture mapping hardware, we need to consider carefully on what to store in the volumetric textures. When the texture are retrieved in fragment programs, the voxel contents are linearly interpolated (supposed linear method is used, which is reasonable). For textures that contains colors and/or alpha values, even normal vectors, interpolation makes good sense given the proper post process if required (e.g. normalization in the case of normal textures). In the case of our transparency texture, the content of a voxel is some representation of an ellipsoid. We need to ensure that the representation is interpolatable, and the interpolated result still represents an ellipsoid. Preferably the ellipsoid representation can be filtered by linear interpolation.

The quadratic form of an ellipsoid can be represented by a 3×3 matrix Q (point M is on the ellipsoid if and only if $M^T Q M = 1$). The normal distribution function encoded by two ellipsoids is 'almost-additive' with respect to Q^{-1}, subject to an error of magnitude $O(\theta^2)$, where θ is the separation angle between the two ellipsoids (see [30] for details). We conclude that Q^{-1} is a good choice to store in the texture. It also indicates that the ellipsoid representation of the NDF can be filtered by interpolating Q^{-1}. while down-sampling the transparency texture, we weight the Q^{-1} of each ellipsoid by its opacity and sum them. The axes of the result ellipsoid are recovered

from the eigenvectors and eigenvalues of the summed matrix.

Although we can store only two axes of an ellipsoid and derived the third one and the quadratic form as Neyret did in [30], we prefer to store the full matrix Q^{-1} in the texture to lower the computational burden of the fragment processing unit, which is already heavily loaded in our rendering algorithm. The ellipsoid axes are scaled such that the longest one has unit length. This makes all 9 components of Q^{-1} in the range of $[-1, 1]$ (see Appendix B). We can stored them in textures without truncation after scaling and translating by 0.5 (i.e. $q' = 0.5 * q + 0.5$) and transforming them into the range of $[0, 1]$. This transformation can be easily reversed (i.e. $q = 2 * q' - 1$) in fragment programs. Furthermore, The normalization of the longest axis scales the ellipsoids to similar size, facilitating direct interpolation of the transparency texture by the texture mapping hardware.

6.4.4 Rendering of textures with NDF

The rendering method of the texture with NDF is a slightly modified version of the idea described by Neyret in [30]. We have implemented the algorithm as a fragment program on NVIDIA GeForceFX video card. The ellipsoid is first projected onto a plane perpendicular to the view direction. The quadratic form Q' of the projected ellipse is derived from the ellipsoid quadratic form Q and the view direction d. The ellipse axes are recovered from the eigenvalues and eigenvectors of Q' and nine uniformly distributed samples (3×3 regular grids) are generated (Figure 6.4). Since the length of the fragment program directly affects the rendering performance, we use only nine samples to compute the integral on an ellipsoid here. The integration is performed numerically by summing these samples. To further shorten the fragment program, we take the nine normalized normal vectors and average them directly. The

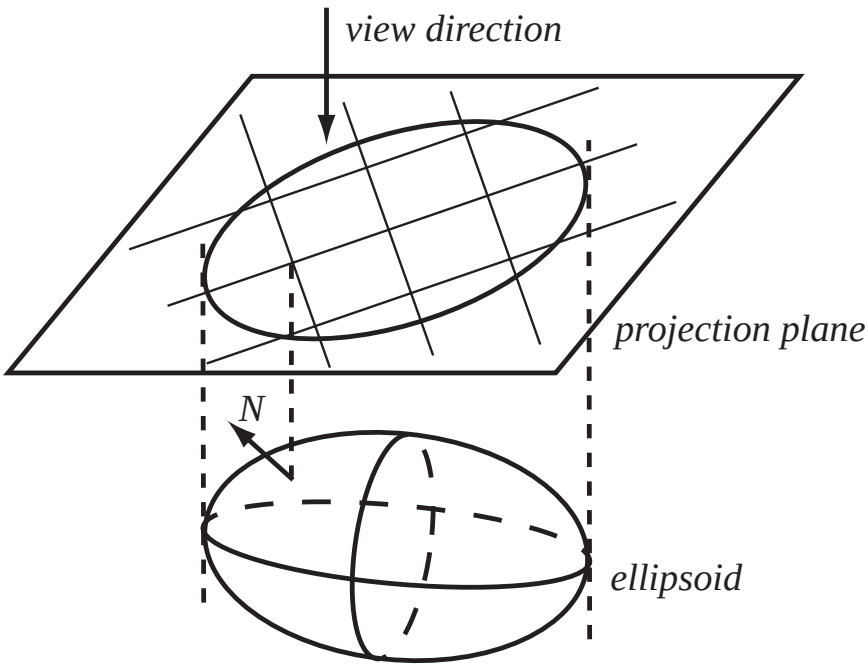

Figure 6.4: Rendering of NDF. The ellipsoid is projected onto the projection plane. Nine samples are picked in the projected ellipse and projected back onto the ellipsoid. Normals are sampled at these projected points.

averaged vector (not normalized) is then used as the normal vector at the current pixel to do shading. If the directions of the nine normal vectors vary a lot, which means the object is rough, the length of the averaged normal vector will be much less than 1 and the pixel appears dark. This is consistent with the real world observation. In our experiments, it is also a close approximation to the result obtained by shading on all nine locations and averaging. It shortens the compiled fragment program by 25%.

6.5 Double mipmap rendering

A surface with double mipmaped volumetric textures is rendered in two passes, with the occupancy textures and the transparency textures respectively.

6.5.1 First pass

The surface is first rendered exactly as described in Chapter 5 with the mipmaped occupancy texture. This pass only renders the part of the object with clearly visible geometric details. Slices through the texture are rendered with alpha test enabled. The correct mipmap levels are picked by the rasterization hardware during rendering. If the object or part of it is near the camera and appears large in the scene, high-resolution levels are used and the object is rendered in full geometric detail. If the object is far and occupies only a small number of pixels, filtered levels in the mipmap are used. Due to the existence of an empty level, which is also the lowest-resolution level of the mipmap (see Figure 6.2), far part of the object may be completely clipped by the alpha test.

6.5.2 Second pass

The second pass renders the object with the transparency texture. Alpha blending is applied in this pass. In contrast to the occupancy texture used in the first pass, the empty level in the mipmaped transparency texture is the level of the highest resolution. The part of the object near the viewer will be rendered with $\alpha = 0$, thus completely discarded by alpha blending. The two overlapping levels of the two textures are of the same size and applied to the same part of the rendered object, forming a *transitional zone*. When the linear interpolating filter is used to retrieve the texture

contents, the empty levels are interpolated with other levels in the two textures respectively. This makes the retrieved alpha from the occupancy texture goes to zero gradually from near to far in the transitional zone, while the alpha from the transparency texture fades into full effect. A simple surface with a double mipmaped volumetric texture is rendered in Figure 6.5. The transition from the occupancy texture to the transparency texture is smooth as shown in the picture.

The second pass use alpha blending to achieve transparent effects. This requires we render faces from far to near. One can use the order-independent transparency technique developed by Everitt in [19]. However, we only use blending to render the part of objects that is far from the viewer in a typical scene. The near part is rendered with the occupancy texture and does not have this requirement. Due to the layer structure of our representation, we can go easy on this problem without ruining the results. In the second pass, we render the object layer by layer in an inside-to-outside order. For objects that have no intuitive definition of inside and outside, such as a terrain model, we render the layers bottom-to-up. At the same time, we enable alpha test with $\alpha = 0$ to discard empty areas and let it to be seen through.

Another problem with the transparency texture is how many slices to render. For the occupancy texture, increasing the number of slices rendered improves the rendering quality. But in case of the transparency texture, the number of slices rendered has direct effects on the results: the more slices, the less transparent the object appears. According to our interpretation of the transparency texture, the slice number of each mipmap level should be the same as the dimension of the texture at that level. This can only be implemented by separating each level into a different texture and render one pass for each texture. But this will dramatically increase the rendering time just for the far part of the object. We pick the slice number to be the averaged texture

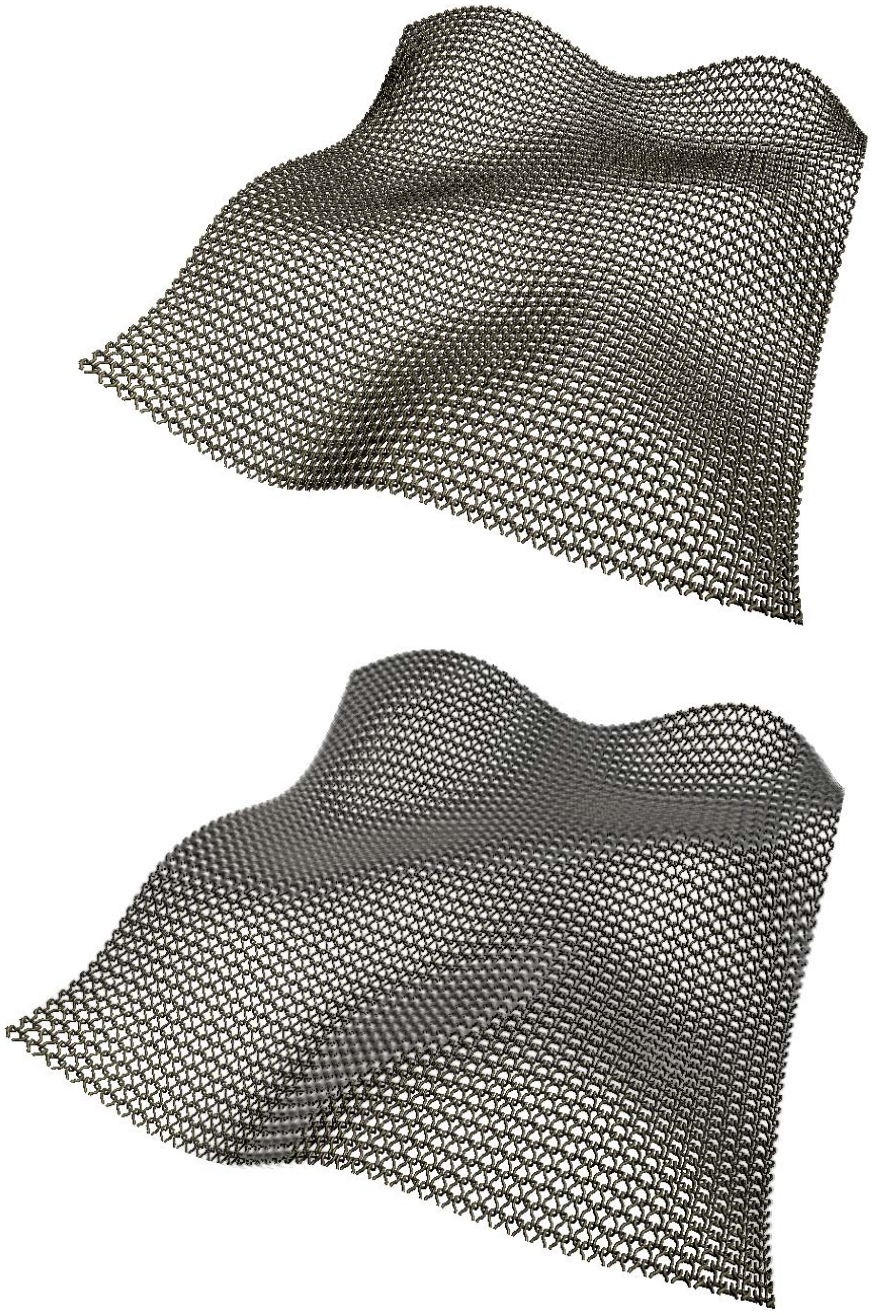

Figure 6.5: A surface rendered with unmipmaped and double mipmaped volumetric textures. Double mipmaping provides smooth transition from full geometric details to transparent effects.

dimensions of the overlapping level and the next level in the transparency texture mipmap. This is a trade-off by experiments. For the overlapping level, the result is compensated by the occupancy texture, and it is acceptable for other levels.

6.6 Alpha in transparency textures

6.6.1 Directional transparency

The alpha channel of the occupancy texture contains samples of a function, which defines the object implicitly. These alpha values are down sampled and interpreted as transparency when we go down the mipmap levels. Some objects display different transparency in different directions, e.g. a bunch of hollow tubes. Since we need three textures to store the quadratic form of the ellipsoids, we have three alpha channels that are freely available for the simulation of the directional transparency.

In the construction of a transparency texture in a double mipmap, we consider the scalar alpha channel in the occupancy texture as three channels with the same value. The three channels represent the directional transparency of the object in the three axis directions of the parameterization space. Each channel is then filtered separately into the transparency texture.

With consideration of the directional property, there are two cases to consider in filtering as shown in Figure 6.6.

The alpha values in the picture represent the voxel transparency in the upwards direction. α_1 and α_2 are the alpha value of the two voxels, and $\alpha = 1$ means the voxel is opaque. Denote α to be the new alpha value when the two voxels are filtered into one voxel.

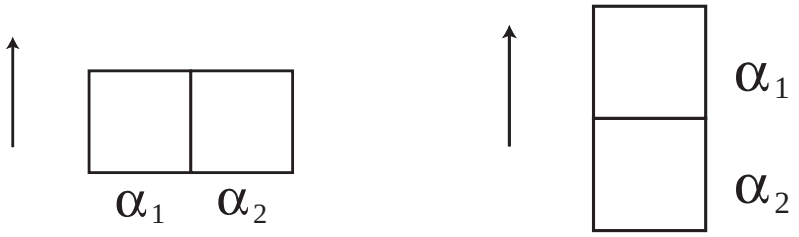

Figure 6.6: Relative positions of voxels in filtering of directional alpha channels. The alpha values here represent the transparency in the upwards direction as shown by the arrows.

In the first case, two voxels are side by side. The filtered upwards alpha value α is the average of α_1 and α_2: $\alpha = (\alpha_1 + \alpha_2)/2$.

In the second case, it gets a little more complex when two voxels lie in a row along the direction of the represented transparency. The correlation of the two voxels should be considered to obtain the correct result. If the correlation coefficient $C = 0$, two voxels are independent and the combined alpha $\alpha = 1 - (1 - \alpha_1)(1 - \alpha_2)$. (The probability that light goes through each voxel is independent.) If $C = 1$, the opaque parts of the two voxels overlap and $\alpha = \max(\alpha_1, \alpha_2)$ (Figure 6.7). While both α_1 and α_2 are in the open interval $(0, 1)$, α is a strict decreasing function of the correlation C.

6.6.2 Alpha blending

Standard hardware alpha blending in rendering treats all voxels as not correlated. That makes the filtered texture look more opaque than the original when correlation exists, no matter we store the alpha as scalar values or multi-component vectors. In practice, correlation needs to be considered in rendering of filtered volumetric tex-

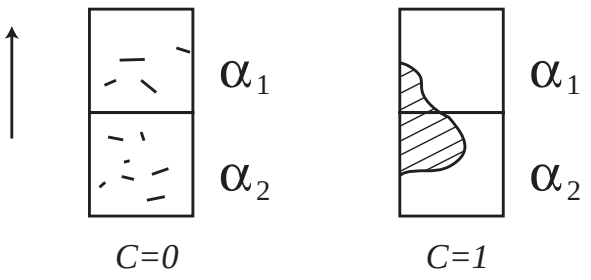

Figure 6.7: Correlation of voxels in filtering of directional alpha channels when voxels lie in a row along the direction of represented transparency. C is the correlation coefficient of the two voxels.

tures. It posts a difficulty with current hardware implementation of the rendering pipeline. The fragment program does not have read access to the destination frame buffer, thus unable to accomplish the desired blending function. Software rendering is able to do it but will not be suitable whenever interactivity is expected for a complex scene.

Chapter 7

Applications

In this chapter we describe a variety of operations that we have implemented in our modeling system. The modeling system uses the shell generation and rendering algorithms explained in previous chapters. These high performance algorithms enable us to process and manipulate a lot of complex models in real-time, which is hard to achieve by other methods.

7.1 Deformations

The base surface of a volumetric-textured object can be deformed with the volumetric texture moving along with it. In our modeling system, the base surface is represented as a multi-resolution subdivision surface and deformed by moving its control points. At each deformation step, the base surface is subdivided and the shell is recomputed on the subdivided surface to generate smooth action of the textured object. We take advantage of the locality of the field that defines the shell, and recompute only the part of the shell within the field influence distance from the modified surface part.

This can be done at interactive rates (Figure 7.2 and 7.3).

To further speed up the shell updating process, we implemented an interpolation coefficient t ($0 \leq t \leq d$) in our system, where d is the subdivision depth of the base surface. If $t = 0$, all shell directors are computed using the algorithm described in Chapter 4. In the case of $t > 0$, we only compute a fraction of the shell directors using our shell generation algorithm and generate others by bilinear interpolation. Specifically, for each top-level quad face, which has $(2^d + 1) \times (2^d + 1)$ vertices on subdivision level d, we compute the shell directors on a $(2^{d-t}+1)\times(2^{d-t}+1)$ grid and interpolate for other vertices (Figure 7.1). When the multi-resolution base surface does not have detail on any subdivision level or the magnitude of the level details is relatively small compared to the size of the face on that level, the interpolation method work very well and provide noticeable performance advantage.

Other method can also be used to implement the deformation. We note that if a volume deformer is used to modify the surface, the same volume deformation can be applied to the shell and no interactive re-computation of the shell is necessary. However, for significant deformation the shell is best recomputed to ensure consistent nominal thickness.

7.2 Moving geometry along the surface

Image editing operations can be easily applied to volumetric textures in a straight forward fashion: one can ignore the third dimension (such as translating, cropping operations, etc) or extend the operations to the third dimension symmetrically (such as blurring operation and a lot of other filters). These editing operations result in changes of the implicitly geometry defined the volumetric textures (Figure 7.4).

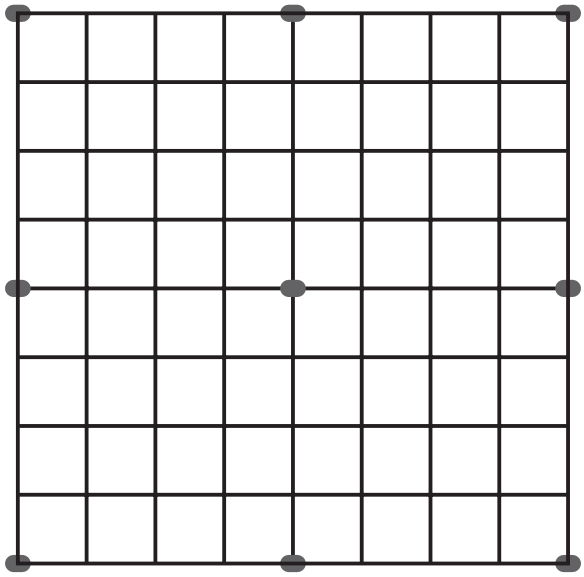

Figure 7.1: Interpolation in shell director computation. In the case of subdivision depth $d = 3$ and interpolation coefficient $t = 2$, each top-level quad face is parameterized on a domain shown above, where only the shell directors at the marked vertices are computed. All other directors are generated by bilinear interpolation.

However, mapping a texture onto a surface in three dimension space causes distortion of the texture. When some of these extended image editing operations (e.g. translating) are implemented, geometric distortion of the texture mapping should be taken into account. The problem is identical to the one addressed in [4]. The target area, to which the texture is moved, and the source area are re-parameterized on a common planar domain with a distortion-minimizing parametrization. The common parameterization is used to re-sample the source texture over the target geometry. The same approach can be used for the translating operation of volumetric textures.

We implemented the translating operation on volumetric-textured object as a proof of concepts in our system. The user can specify a region of arbitrary shape and fill the region with a volumetric texture. This piece of texture can be moved around on the object. The trace of the texture is filled with the base surface.

7.3 Boolean operations and carving

One of the advantages of volumetric geometry representations is that boolean operations become relatively simple (Figure 7.5). In the case of volumetric textures the situation is complicated by the fact that the transformation from world coordinates to texture coordinates is nonlinear. However, it is still relatively simple to compute a boolean operation between a regular non-distorted volumetric object and the volumetric-textured surface. This requires re-sampling the volumetric object over the shell grid, which is straightforward.

We implemented two boolean operations: cutting holes on a textured object and combining two volumetric textures. Cutting operation uses the same method to define an arbitrary shape as we did for the translating operation. The part of textures that

falls into the specified region is set to zero to accomplish the cutting operation. In the combining operation, we utilize the multiple texture units on the video card and the programmability of the pixel engine. Two textures are mapped to the same region of the object. The fragment program picks the texture with the larger α value at each pixel, resulting in a union operation of the two textures. This implementation make it very convenient to move one texture in the other, which requires no computation but a simple translation of one set of texture coordinates. We used this method to paste a horse texture into a terrain full of trees (represented by a texture), and move the horse around freely while the trees are growing.

Applying a boolean operation to two volumetric-textured surfaces is much more difficult. It involves re-sampling an ISO-surface represented by a distorted grid into another distorted grid. That is not a trivial problem.

7.4 Animated textures

Removing details from the underlying geometric representation and placing them into 3D textures makes some animations much simpler to execute. One example is the boiling man (Figure 7.6). The texture is procedurally animated to show the bubbles. During the animation, the bubbles rise from the bottom and burst on the surface, when small pieces of irregular shape fly around. This would require a change in the topology of the underlying geometry if we were using conventional geometric modeling. Volumetric textures handle the changes of topology without any additional cost, providing real time performance for the animation.

Other examples of texture animation include the waving flag (Figure 7.7) and the growing trees (Figure 7.8). The structural texture on the flag is constructed after an

example appeared in Neyret's work. They produced that animation with off-line renderer, which required 10 minutes per frame. We are able to render the flag animation in real-time in our modeling system. It is still a significant improvement in speed after discounted by the clock rate of our faster CPU.

In all these animated examples, Our fast shell generation algorithm enables us to animate the base mesh and the texture at the same time.

Figure 7.2: Two simple objects with small-scale geometry added.

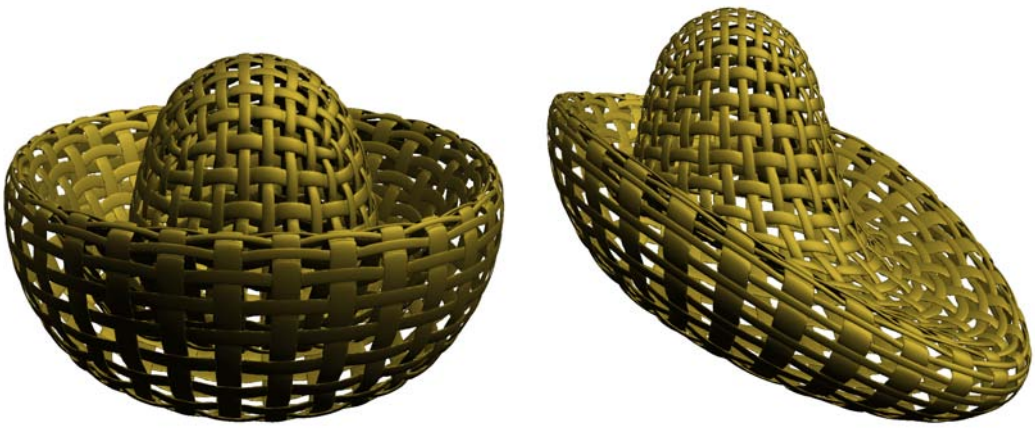

Figure 7.3: Editing operations: deforming a volume-textured surface by modifying the base surface.

Figure 7.4: Editing operations: moving a geometric texture on a surface.

Figure 7.5: Editing operations: cutting a hole on the chain-mail shirt.

Figure 7.6: An animated bubbling texture applied on the mannequin head. The base surface is deformed at the same time while the texture is animated.

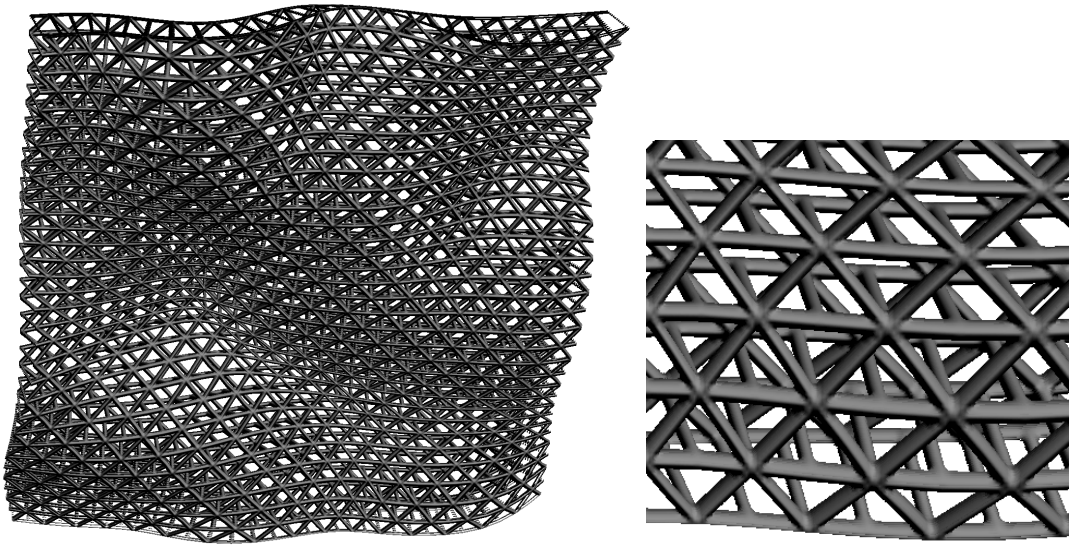

Figure 7.7: A structural texture on a deforming plane and a partial zoom-in view. This model is created after a similar one appeared in Neyret's work.

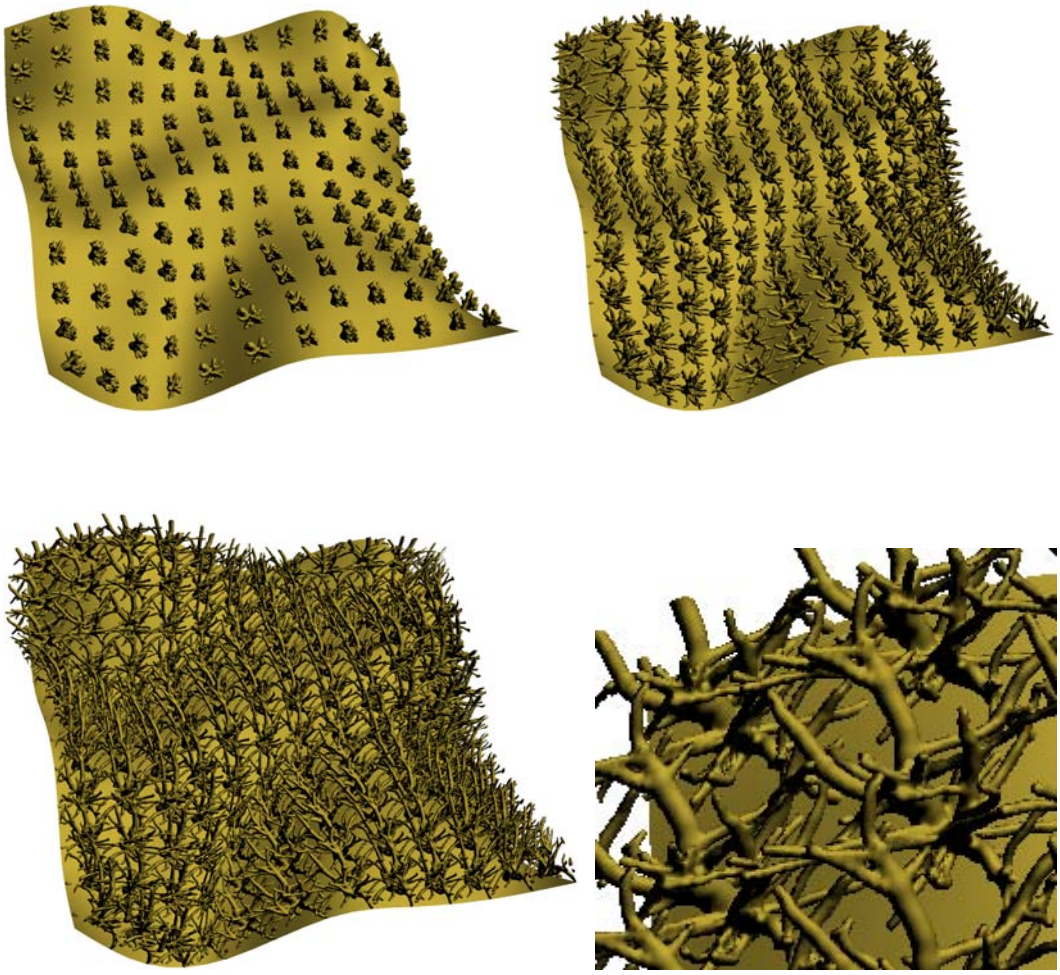

Figure 7.8: Three stages of a growing bush texture with a zoomed-in view.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

We presented methods for constructing and rendering surfaces with small-scale geometry represented by volumetric textures. These methods enable a variety of interactive modeling operations.

The shell construction algorithm is based on the L_p-averaged distance function, which is generalized from the standard distance function. The gradient field of the L_p-averaged distance function is well defined away from the surface. The shell around a surface can be obtained by integrating the gradient field directly using an adaptive explicit Euler method. In case that insufficient sampling causes difficulties for numerical methods, we developed the analytic formulas to compute the field. This analytic approach makes it possible to use a coarser base mesh for computation of the field and might lower the computation cost in some cases. The shell construction algorithm is expected to be useful for a variety of applications using volumetric textures, such as fur rendering, weathering simulation, etc.

Our rendering algorithm for volumetric-textured objects extends the slice-based volume rendering algorithm. The algorithm renders the object in multiple passes. For each pixel, we find the visible point in the object on one slice and the outside point on another slice immediately before the visible point. The texture coordinate for the pixel is then computed by interpolating that of the two points according to the two alpha values. The interpolated texture coordinates is continuous between slices. This algorithm effectively eliminates the visible slice boundary artifacts in standard slice-based volume rendering. It enables us to reduce the number of slices rendered without noticeably degrading the result. We do not need to render as many slices as the size of the texture to show off all the details of the object. The rendering algorithm can also be used for standard volume rendering applications to improve the ISO-surface rendering quality.

The mipmaping method we developed for volumetric textures produces rendering results similar to the real world observations. It is easy to construct hierarchies for volumetric textures by low-pass filtering and sub-sampling. However it is well known that low-pass filtering is not the best way to simplify implicit geometry, as undesirable topological artifacts may appear. Our current approach as shown in Chapter 6 uses transparency to amend the lost of topological information in sub-sampling. It is a reasonable approach for a lot of objects we want to represent. For example, you will not see any threads on a window screen while looking from far away. But the scene behind the screen will look darker, just like you are looking through a gray semi-transparent window. We use two textures of different resolutions to represent geometrical and transparent appearance of a texture. Explicit normal and alpha information are stored in the high resolution geometrical texture. In the low resolution transparency texture, normals are encoded by a distribution function at each texel to

avoid artifacts due to insufficient sampling density. The transition between these two textures is implemented smoothly using mipmaping hardware.

8.2 Future work

As we have mentioned, the shell construction algorithm that we have proposed has considerable potential for acceleration. Ideally normal displacement should be used whenever possible, with our algorithm applied only in complex areas.

Examples we have shown only use procedure generated volumetric textures. Procedure methods are very effective in generating regular texture patterns, such as chain mail, knits and fabrics, etc. A lot of irregular textures can be generated by procedure methods using Perlin noise [32, 33]. We also want to derived models and textures from real world objects. One should be able to build models and textures out of data from CT, MRI or ultrasound scanned objects. It involves fitting a mesh (or a shell directly) to the scan data and re-sampling it into volumetric textures. This approach has the advantage of achieving high resolution model with smaller data storage. Feature aligned textures are likely to have few visual artifacts over traditional volume representation.

Physical simulation on objects represented by thick surfaces is another interesting area worth exploring. Topologically complex objects present some tough problems in simulation, partially inherited from their conventional complex representation. Our effective representation based on volumetric textures might simplify the problem in certain ways.

8.3 Special thanks

I would like to thank Professor Denis Zorin, Professor Demetri Terzopoulos and Ms. Xin Ge for proof reading the document. They provided great help in correcting typos and improving the writing.

Appendix A

Analytical integration

The averaged distance function discussed in Chapter 4 is defined on the L_p norm,

$$d_p(\mathbf{x}, M) = A^{1/p} \left(\int_M |\mathbf{x} - \mathbf{y}|^{-p} d\mathbf{y} \right)^{-1/p} \tag{A.1}$$

When $p = 3$, the integral involved in the averaged distance function $d_p(\mathbf{x}, M)$ can be computed analytically on an arbitrary triangle. This enables us to compute the averaged distance function precisely and avoid the "escape" problem in the case that a surface region with high curvature is not sampled densely enough for numerical methods.

As indicated in Figure 4.9 on Page 35, we can decompose a triangle into three wedges and compute the integral in Equation (A.1) over a triangle by three parts:

$$\begin{aligned} \int_{\triangle P_1 P_2 P_3} |\mathbf{x} - \mathbf{y}|^{-p} d\mathbf{y} = & \int_{\angle(P_1, \overrightarrow{P_1P_2}, \overrightarrow{P_1P_3})} |\mathbf{x} - \mathbf{y}|^{-p} d\mathbf{y} \quad + \\ & \int_{\angle(P_2, \overrightarrow{P_3P_2}, \overrightarrow{P_1P_2})} |\mathbf{x} - \mathbf{y}|^{-p} d\mathbf{y} \quad - \\ & \int_{\angle(P_3, \overrightarrow{P_3P_2}, \overrightarrow{P_1P_3})} |\mathbf{x} - \mathbf{y}|^{-p} d\mathbf{y} \end{aligned}$$

where $\angle(V, \vec{e_1}, \vec{e_2})$ represents a wedge whose end point is V and covers the area be-

tween $\vec{e_1}$ and $\vec{e_2}$ counter-clockwise. We only need to compute the integral over a wedge explicitly to obtain the analytical result over a triangle.

In the following we develop the formulas to compute $\int_M |\mathbf{x}-\mathbf{y}|^{-3}d\mathbf{y}$ over a wedge $\angle(V,\vec{e_1},\vec{e_2})$. Without losing generality, assume that the wedge lies in the xy plane of a coordinate system with its vertex V at the orig point O and the starting edge $\vec{e_1}$ on the positive x coordinate axis. The interior of the wedge covers the counter-clockwise angle from the positive x axis to $\vec{e_2}$. Denote β the size of the angle. Since our wedges are decomposed from triangles, β is in the range $(0,\pi)$. Transform the xy plane to polar coordinate system (r,θ), the integral becomes:

$$\int_{\angle(O,\vec{e_1},\vec{e_2})} |\mathbf{x}-\mathbf{y}|^{-3}d\mathbf{y} = \int_0^{\beta}\int_0^{+\infty} |\mathbf{x}-\mathbf{y}|^{-3} r dr d\theta$$

Suppose that the Cartesian coordinates of $\mathbf{x}$ is (u,v,w), where u,v,w are arbitrary real numbers. Define q to be the distance between the origin and $\mathbf{x}$ ($q=\sqrt{u^2+v^2+w^2}$). The Cartesian coordinates of $\mathbf{y}$ can be written in terms of (r,θ) as $(r\cos\theta, r\sin\theta, 0)$. Denote the above integral $H(\mathbf{x})$. Then we have:

$$\begin{aligned}
H(\mathbf{x}) &= \int_0^{\beta}\int_0^{+\infty} |\mathbf{x}-\mathbf{y}|^{-3} r dr d\theta \\
&= \int_0^{\beta}\int_0^{+\infty} \frac{r}{((u-r\cos\theta)^2+(v-r\sin\theta)^2+w^2)^{3/2}} dr d\theta \\
&= \int_0^{\beta}\int_0^{+\infty} \frac{r}{(r^2-2r(u\cos\theta+v\sin\theta)+q^2)^{3/2}} dr d\theta
\end{aligned}$$

Define $U(\theta)=u\cos\theta+v\sin\theta$, which does not depend on r. So, we can integrate on r as $U(\theta)$ is constant:

$$\begin{aligned}
H(\mathbf{x}) &= \int_0^{\beta}\int_0^{+\infty} \frac{r}{(r^2-2rU(\theta)+q^2)^{3/2}} dr d\theta \\
&= \int_0^{\beta} \frac{rU(\theta)-q^2}{(q^2-U(\theta)^2)\sqrt{r^2-2rU(\theta)+q^2}}\Bigg|_0^{+\infty} d\theta \\
&= \int_0^{\beta} \frac{1}{q-U(\theta)} d\theta
\end{aligned}$$

Substitute $U(\theta)$ back in and compute the integral on θ:

$$\begin{aligned} H(\mathbf{x}) &= \int_0^{\beta} \frac{1}{q - (u\cos\theta + v\sin\theta)} d\theta \\ &= \frac{2}{w} \arctan\left(\frac{(q+u)\tan\frac{\theta}{2} - v}{w}\right)\Bigg|_0^{\beta} \\ &= \frac{2}{w}\left(\arctan\left(\frac{(q+u)\tan\frac{\beta}{2} - v}{w}\right) + \arctan\left(\frac{v}{w}\right)\right) \end{aligned}$$

Define

$$\theta_1 = \arctan\left(\frac{(q+u)\tan\frac{\beta}{2} - v}{w}\right), \quad \text{and} \quad \theta_2 = \arctan\left(\frac{v}{w}\right),$$

then

$$\begin{aligned} \tan\theta_1 &= \frac{(q+u)\tan\frac{\beta}{2} - v}{w}, \quad \tan\theta_2 = \frac{v}{w}, \\ \tan(\theta_1+\theta_2) &= \frac{\tan\theta_1 + \tan\theta_2}{1 - \tan\theta_1\tan\theta_2} = \frac{w}{(q-u)\cot\frac{\beta}{2} - v} \end{aligned}$$

It leads to the formula presented in Chapter 4:

$$H(\mathbf{x}) = \frac{2}{w}\arctan\left(\frac{w}{(q-u)\cot\frac{\beta}{2} - v}\right) \tag{A.2}$$

Now, the gradient of Equation (A.1):

$$\begin{aligned} g_3(\mathbf{x}, M) &= \nabla_{\mathbf{x}}(d_3(\mathbf{x}, M)) \\ &= -\frac{1}{3A}(d_3(\mathbf{x}, M))^4 \nabla_{\mathbf{x}} \left(\int_M |\mathbf{x} - \mathbf{y}|^{-3} d\mathbf{y}\right) \end{aligned}$$

can be computed analytically by taking the partial derivatives of $H(\mathbf{x})$ on u, v, w explicitly:

$$\begin{aligned} H_u(\mathbf{x}) &= \frac{\sin\beta}{q(q - (u\cos\beta + v\sin\beta))} \\ H_v(\mathbf{x}) &= \frac{1}{q(q-u)} - \frac{\cos\beta}{q(q - (u\cos\beta + v\sin\beta))} \\ H_w(\mathbf{x}) &= -\frac{H(\mathbf{x})}{w} + \frac{(u^2+v^2-qu)\sin\beta - qv(1-\cos\beta)}{qw(q-u)(q - (u\cos\beta + v\sin\beta))} \end{aligned}$$

Equation (A.2) assumes $w \neq 0$. While $w = 0$, point $\mathbf{x}$ is in the xy plane. For this special case, we can compute the limit of Equation (A.2) when $w \to 0$, or recompute $H(x)$ with $w = 0$ (for now, $q = \sqrt{u^2 + v^2}$):

$$\begin{aligned} H(\mathbf{x}) &= \int_0^\beta \frac{1}{q - (u\cos\theta + v\sin\theta)} d\theta \\ &= \left. \frac{2(q-u)}{v(q - u - v\tan\frac{\theta}{2})} \right|_0^\beta \\ &= \frac{2}{(q-u)\cot\frac{\beta}{2} - v} \end{aligned}$$

For the gradient of $H(\mathbf{x})$ in case of $w = 0$, the first two components $H_u(\mathbf{x})$ and $H_v(\mathbf{x})$ are still computed by the formulas shown above, and $H_w(\mathbf{x}) = 0$.

$H(\mathbf{x})$ only exists while the point $\mathbf{x}$ is out of the wedge. Furthermore, when $\mathbf{x}$ is very close to the wedge, the computation of $H(\mathbf{x})$ becomes unstable. Rounding errors may cause the integral on a triangle to be negative, and result in wrong gradient directions. During implementation, we need to take these into account and decompose a triangle in a way that $\mathbf{x}$ does not lie in any of the three wedges. To ensure stability, $\mathbf{x}$ is preferred to be as far from the wedges as possible.

Appendix B

Quadratic form representation of ellipsoids

Denote vectors v_1, v_2, v_3 to be three normalized axes of an ellipsoid and scalars a_1, a_2, a_3 the length of the three axes respectively. Then the followings hold:

$$|v_i| = 1, \quad v_i \cdot v_j = 0, \quad a_i > 0, \quad (1 \leq i, j \leq 3,\ i \neq j)$$

where '$\cdot$' is the dot product of two vectors.

Assume the ellipsoid is centered at the origin. Then the equation of the ellipsoid is:

$$\frac{(M \cdot v_1)^2}{a_1^2} + \frac{(M \cdot v_2)^2}{a_2^2} + \frac{(M \cdot v_3)^2}{a_3^2} = 1 \tag{B.1}$$

where M is a point of space.

The left side of Equation (B.1) can be written as:

$$\sum_{i=1}^{3} \frac{(M \cdot v_i)^2}{a_i^2} = \sum_{i=1}^{3} \frac{M^T v_i v_i^T M}{a_i^2} = M^T \left(\sum_{i=1}^{3} \frac{v_i}{a_i} \left(\frac{v_i}{a_i} \right)^T \right) M$$

where X^T is the transpose of the matrix X.

We can use the 3×3 matrix in the middle of the last term to represent the origin-centered ellipsoid and denote it Q. A point M is on the ellipsoid if and only if $M^T Q M = 1$. Q has the following format in terms of the ellipsoid axes and their length:

$$Q = \sum_{i=1}^{3} \frac{v_i}{a_i} \left(\frac{v_i}{a_i}\right)^T = \begin{pmatrix} \frac{v_1}{a_1} & \frac{v_2}{a_2} & \frac{v_2}{a_2} \end{pmatrix} \begin{pmatrix} \frac{v_1}{a_1} & \frac{v_2}{a_2} & \frac{v_2}{a_2} \end{pmatrix}^T$$

Notice that v_1, v_2, v_3 form an orthogonal basis of the 3D space. So we have Q^{-1} as:

$$\begin{aligned} Q^{-1} &= \left(\begin{pmatrix} \frac{v_1}{a_1} & \frac{v_2}{a_2} & \frac{v_2}{a_2} \end{pmatrix}^T \right)^{-1} \begin{pmatrix} \frac{v_1}{a_1} & \frac{v_2}{a_2} & \frac{v_2}{a_2} \end{pmatrix}^{-1} \\ &= (a_1 v_1 \quad a_2 v_2 \quad a_3 v_3)(a_1 v_1 \quad a_2 v_2 \quad a_3 v_3)^T \end{aligned}$$

Write the orthogonal basis $v_i = (b_{1,i}, b_{2,i}, b_{3,i})^T,\ (1 \le i \le 3)$. Define $w_i = (b_{i,1}, b_{i,2}, b_{i,3})^T,\ (1 \le i \le 3)$. It is easy to prove that $|w_i| = 1$ and they form another orthogonal basis of the 3D space. If the ellipsoid is scaled so that its longest axis has unit length, (i.e. $a_i \le 1,\ 1 \le i \le 3$), every component $q_{i,j}\ (1 \le i, j \le 3)$ of the matrix Q^{-1} is in the range of $[-1, 1]$.

$$\begin{aligned} |q_{i,j}| &= |a_1^2 b_{i,1} b_{j,1} + a_2^2 b_{i,2} b_{j,2} + a_3^2 b_{i,3} b_{j,3}| \\ &\le a_1^2 |b_{i,1} b_{j,1}| + a_2^2 |b_{i,2} b_{j,2}| + a_3^2 |b_{i,3} b_{j,3}| \\ &\le |b_{i,1} b_{j,1}| + |b_{i,2} b_{j,2}| + |b_{i,3} b_{j,3}| \\ &\le \sqrt{(b_{i,1}^2 + b_{i,2}^2 + b_{i,3}^2)(b_{j,1}^2 + b_{j,2}^2 + b_{j,3}^2)} \\ &= 1 \end{aligned}$$

So, normalizing the longest axis of the ellipsoid puts all $q_{i,j}$ in $[-1, 1]$. This enables us to transform all components of Q^{-1} to $[0, 1]$ by half scaling then half bias. Then we can store the transformed matrix in textures without truncation. In fragment programs, the matrix Q^{-1} can be easily restored by the inverse transformation.

Bibliography

[1] A. Agarwala. Volumetric surface sculpting. Master's thesis, MIT, 1999.

[2] J. Barnes and P. Hut. A hierarchical $o(N \log N)$ force calculation algorithm. *Nature*, 324:446, 1986.

[3] M. Bern and P. Plassmann. Mesh generation. In *Handbook of computational geometry*, pages 291–332. North-Holland, Amsterdam, 2000.

[4] H. Biermann, I. Martin, F. Bernardini, and D. Zorin. Cut-and-paste editing of multiresolution surfaces. *ACM Transactions on Graphics*, 21(3):312–321, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[5] J. Bloomenthal, editor. *Introduction to implicit surfaces*. Morgan Kaufmann, 1997.

[6] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of SIGGRAPH '94*, pages 91–98. ACM SIGGRAPH, October 1994. ISBN 0-89791-741-3.

[7] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and representation of 3D objects with

radial basis functions. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 67–76. ACM Press / ACM SIGGRAPH, August 2001. ISBN 1-58113-292-1.

[8] E. Chernyaev. Marching cubes 33 : Construction of topologically correct isosurfaces. Technical Report CN/95-17, CERN, 1995.

[9] H. Choi, S. Choi, and H. Moon. Mathematical theory of medial axis transform. *Pacific Journal of Mathematics*, 181(1):57–88, 1997.

[10] H. Choi, S. Choi, H. Moon, and N. Wee. New algorithm for medial axis transform of plane domain. *Graphical Models and Image Processing*, 59(6):463–483, 1997.

[11] H. Choi, C. Han, S. Choi, H. Moon, K. Roh, and N. Wee. Two-dimensional offsets via meidal axis transform I: Mathematical theory. *preprint*, 2001.

[12] H. Choi, C. Han, S. Choi, H. Moon, K. Roh, and N. Wee. Two-dimensional offsets via meidal axis transform II: Algorithm. *preprint*, 2001.

[13] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. P. B. Jr., and W. Wright. Simplification envelopes. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 119–128, New Orleans, Louisiana, August 1996. ACM SIGGRAPH / Addison Wesley.

[14] T. J. Cullip and U. Neumann. Accelerating volume reconstruction with 3D texturing hardware. Technical Report TR93-027, University of North Carolina, 1993.

[15] B. Cutler, J. Dorsey, L. McMillan, M. Müller, and R. Jagnow. A procedural approach to authoring solid models. *ACM Transactions on Graphics*, 21(3):302–311, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[16] S. Dietrich. Elevation maps. Technical report, NVIDIA Corporation, 2000.

[17] J. Dorsey, A. Edelman, J. Legakis, H. W. Jensen, and H. K. Pedersen. Modeling and rendering of weathered stone. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 225–234, Los Angeles, California, August 1999. ACM SIGGRAPH / Addison Wesley Longman. ISBN 0-20148-560-5.

[18] D. Eberly, R. Gardner, B. Morse, S. Pizer, and C. Scharlach. Ridges for image analysis. *Journal of Mathematical Imaging and Vision*, 4:351–371, 1994.

[19] C. Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, 2001.

[20] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 249–254. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, July 2000. ISBN 1-58113-208-5.

[21] W. D. Henshaw. Automatic grid generation. In *Acta numerica, 1996*, volume 5 of *Acta Numer.*, pages 121–148. Cambridge Univ. Press, Cambridge, 1996.

[22] J. T. Kajiya. Anisotropic reflection models. *SIGGRAPH Computer Graphics*, 19(3):15–21, 1985.

[23] J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 271–280. ACM Press, 1989.

[24] J. Kautz and H.-P. Seidel. Hardware accelerated displacement mapping for image based rendering. In *Graphics Interface 2001*, pages 61–70, June 2001. ISBN 0-96888-080-0.

[25] J. Lengyel. Real-time hair. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 243–256. Eurographics, June 2000. ISBN 3-211-83535-0.

[26] J. E. Lengyel, E. Praun, A. Finkelstein, and H. Hoppe. Real-time fur over arbitrary surfaces. In *2001 ACM Symposium on Interactive 3D Graphics*, pages 227–232, March 2001. ISBN 1-58113-292-1.

[27] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, volume 21, pages 163–169, Anaheim, California, July 1987.

[28] A. Meyer and F. Neyret. Interactive volumetric textures. In G. Drettakis and N. Max, editors, *Eurographics Rendering Workshop 1998*, pages 157–168, New York City, NY, July 1998. Eurographics, Springer Wein. ISBN 3-211-83213-0.

[29] F. Neyret. A general and multiscale model for volumetric textures. In W. A. Davis and P. Prusinkiewicz, editors, *Graphics Interface '95*, pages 83–91. Canadian Information Processing Society, Canadian Human-Computer Communications Society, May 1995. ISBN 0-9695338-4-5.

[30] F. Neyret. Modeling animating and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):55–70, Jan.–Mar. 1998. ISSN 1077-2626.

[31] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko. Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer*, 11(8):429–446, 1995.

[32] K. Perlin. An image synthesizer. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296. ACM Press, 1985.

[33] K. Perlin and E. M. Hoffert. Hypertexture. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, volume 23, pages 253–262, Boston, Massachusetts, July 1989.

[34] S. Pizer, C. Burbeck, J. Coggins, D. Fritsch, and B. Morse. Object shape before boundary shape: Scale-space medial axes. *Journal of Mathematical Imaging and Vision*, 4:303–313, 1994.

[35] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 109–118. ACM SIGGRAPH / Eurographics / ACM Press, August 2000.

[36] A. Ricci. A constructive geometry for computer graphics. *The Computer Journal*, 16(2):157–160, 1973.

[37] A. Rockwood. *Blending surfaces in solid geometric modeling*. PhD thesis, Cambridge University, 1987.

[38] G. Schaufler. Per-object image warping with layered impostors. In *9th Eurographics Rendering Workshop*, pages 145–156, June 1998.

[39] J. A. Sethian. Curvature flow and entropy conditions applied to grid generation. *J. Comput. Phys.*, 115(2):440–454, 1994.

[40] K. Siddiqi, S. Bouix, A. Tannenbaum, and S. W. Zucker. The Hamilton-Jacobi skeleton. In *ICCV (2)*, pages 828–834, 1999.

[41] S. Steinberg and P. M. Knupp. *Fundamentals of Grid Generation*. CRC Press, 1993. ISBN 0849389879.

[42] S. W. Wang and A. E. Kaufman. Volume sculpting. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 151–ff. ACM Press, 1995.

[43] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 169–178, Orlando, Florida, July 1998. ACM SIGGRAPH / Addison Wesley. ISBN 0-89791-999-8.

[44] L. Williams. Pyramidal parametrics. *SIGGRAPH Comput. Graph.*, 17(3):1–11, 1983.

[45] O. Wilson, A. V. Gelder, and J. Wilhelms. Direct volume rendering via 3D textures. Technical Report UCSC-CRL-94-19, UCSC, 1994.

[46] A. Yezzi and J. L. Prince. A PDE approach for thickness, correspondence, and gridding of annular tissues. In *ECCV (4)*, volume 2353 of *Lecture Notes in Computer Science*, pages 575–589. Springer, 2002.