Procedural Authoring of Solid Models

by

Barbara M. Cutler

Bachelor of Science, Computer Science and Engineering
Massachusetts Institute of Technology, 1997

Master of Engineering, Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1999

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2003

Author _____
Department of Electrical Engineering and Computer Science
August 29, 2003

Certified by _____
Julie Dorsey
Professor of Computer Science, Yale University
Thesis Supervisor

Certified by _____
Leonard McMillan
Professor of Computer Science, University of North Carolina, Chapel Hill
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Committee on Graduate Students
Department of Electrical Engineering and Computer Science

Procedural Authoring of Solid Models

by

Barbara M. Cutler

Submitted to the Department of Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology on August 29, 2003
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

This thesis investigates the creation, representation, and manipulation of volumetric geometry suitable for computer graphics applications. In order to capture and reproduce the appearance and behavior of many objects, it is necessary to model the internal structures and materials, and how they change over time. However, producing real-world effects with standard surface modeling techniques can be extremely challenging.

My key contribution is a concise procedural approach for authoring layered, solid models. Using a simple scripting language, a complete volumetric representation of an object, including its internal structure, can be created from one or more input surfaces, such as scanned polygonal meshes, CAD models or implicit surfaces. Furthermore, the resulting model can be easily modified using sculpting and simulation tools, such as the Finite Element Method or particle systems, which are embedded as operators in the language. Simulation is treated as a modeling tool rather than merely a device for animation, which provides a novel level of abstraction for interacting with simulation environments.

I present an implementation of the language using a flexible tetrahedral representation, which I chose because of its advantages for simulation tasks. The language and implementation are demonstrated on a variety of complex examples that were inspired by real-world objects.

Thesis Supervisor: Julie Dorsey
Title: Professor of Computer Science, Yale University

Thesis Supervisor: Leonard McMillan
Title: Professor of Computer Science, University of North Carolina, Chapel Hill

Acknowledgments

I would like to thank my advisors for their support and advice. To Julie for the opportunity to work on such an interesting and challenging project and keeping me focused on the big picture. And to Leonard for questioning my design choices and always pushing me to do my best, even though I can be stubborn. I would also like to thank my committee members, Professor John Ochsendorf and Professor Bill Freeman, for their time and helpful feedback.

Many thanks to Rob Jagnow and Matthias Müller for their part in developing the interactive system and the simulation modules. As one anonymous SIGGRAPH reviewer wrote: "My sense is it would be a big job and the grad student might not complete it before his or her funding ran out." Literally, I couldn't have done it without them. Special thanks to Justin Legakis for being the best officemate, and letting me use his various software libraries and ray tracer. Justin added new features to his ray tracer so that I could produce some neat visualizations. Thanks also to Stephen Duck for creating the input and scene geometry for several of the models in this thesis.

Bryt Bradley was very critical in providing the materials necessary for this research — in particular, the examples shown in Figure 4.4. Thanks to Adel Hanna and Tom Buehler for their timely technical expertise, Manish Jethwa for some last-minute production assistance, and Ray Jones for providing sincere encouragement. Thanks to everyone in the MIT Graphics Group for making the lab a fun and productive place to work (much credit goes to Professor Fredo Durand for encouraging card playing and tea breaks).

I thank my parents for supporting my decision to avoid the "real world" as long as possible (perhaps indefinitely). I also thank the MIT Figure Skating Club, Linda Blount (my skating coach), and my buddies in the pottery studio for helping me keep my priorities in order. Most importantly, I thank Derek Bruening for his ability and willingness to help during any crisis, large or small, real or imagined, 24 hours a day.

Contents

List of Figures and Tables

Chapter 1

Introduction

Digital models are used in a wide range of applications including architectural visualization, prototyping in Computer-Aided Manufacturing (CAM), scientific simulations, and digital characters and environments for the entertainment industry. Thus, there is tremendous demand for a wide variety of digital models and the tools to construct and edit them. While there has been significant progress in the area of rendering over the past three decades, creating and acquiring high-fidelity geometric models remains a challenging and tedious process. The widespread use of the same small set of models, such as the Stanford bunny and the Utah teapot, attests to these difficulties. Recent research has focused on virtual sculpting tools using haptics technology [SensAble Technologies]; however, these systems are still primitive, lacking the fidelity and responsiveness of real-world manipulation. In fact, the main characters in digitally-created movies are still sculpted by artists with clay or other traditional materials and techniques and then digitized.

Most computer graphics imagery is created with surface representations that are unable to capture object properties beyond shape and texture. Popular surface modeling programs such as AutoCAD [Autodesk, Inc.] and Maya [Alias Systems] can be used to compose beautiful imagery, such as the architectural visualization of a museum, shown in Figure 1.1. But what if the architect needs to visualize how this gallery will appear after hundreds of school field trips have scuffed the floor, the plaster ceiling is cracked, and the paint is peeling? Or what if these columns were instead on the exterior of a building, exposed to moisture, pollution and temperature extremes? Many objects gain visual interest over time; for example, the layered construction materials of the Venetian facades shown in Figure 1.2 have been affected by erosion in different ways to expose

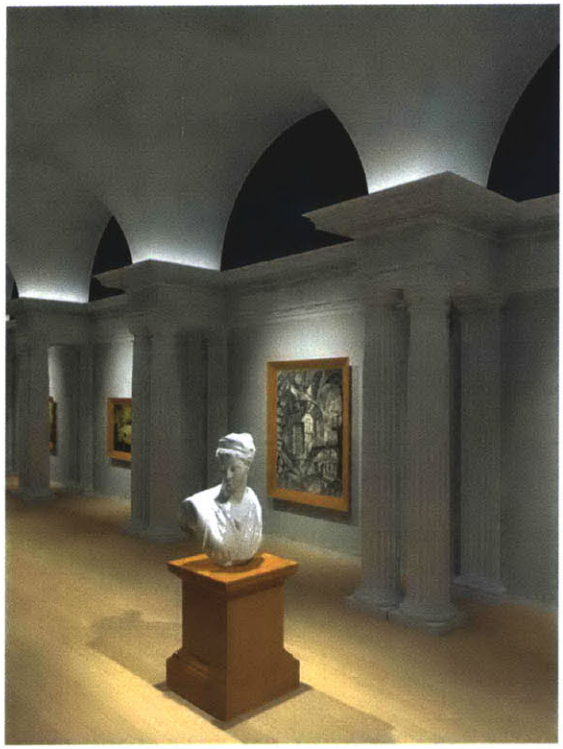


Figure 1.1: Gallery rendering. Image courtesy of Stephen Duck.

the underlying structure. Surface models are not adequate for these and other physically-based operations.

In order for a model to behave as a physically plausible object, it must be augmented with material properties and internal structure. For example, if we want to change the pose of a human figure statue, it is important to use a representation that respects its internal skeleton so that the model hinges correctly at the joints instead of bending bones unrealistically. Furthermore, deformations of a surface model can result in overall volume change or introduce self-intersections, and it can be difficult to perform simple changes in topology such as drilling a hole through the model. These operations are best performed on a volumetric representation.

1.1 Recent Modeling Innovations

There are three recent modeling innovations that contributed to the development of the main ideas in my dissertation: the wealth of new physically-based simulations, advances in three-dimensional digitizing, and the growth of procedural modeling as a tool for design.

Figure 1.2: Examples of complex real-world imagery that would be very difficult to reproduce using surface models. The photograph on the left is from Boccazzi-Varotto [1996].

With the increased speed and availability of computing cycles, physically-based simulations are becoming more practical and popular in computer graphics. However, they have not yet found widespread use in the entertainment industry because they lack the necessary artistic controls. Additionally, many physical simulations are still too expensive or fragile to be used in interactive environments, so user control is often not feasible. In fact, in the gaming industry, where interactivity is the prime consideration, designers make due with coarse approximations of reality. The general problem in all of these situations is the difficulty of preparing interesting volumetric models for simulation, and the lack of control and feedback when applying these simulations.

Three-dimensional digitizing has emerged as a popular technique for acquiring complex models, such as sculptures or mechanical parts, which would be difficult or impossible to create with interactive techniques [Cyberware]. While such digitizers are useful for acquiring surface shape and appearance properties, they do not capture the internal structure of the geometry, which may

be necessary for accurate animation or simulation. Often even the surface texture is lost since the object must be painted matte white for scanning. Tomography and/or dissection techniques can acquire some internal features, but these methods can be expensive, destructive, or impractical to use for certain materials or objects.

The third innovation is procedural modeling, which allows complicated shapes and processes to be described algorithmically. There are many reasons to consider a procedural approach to surface creation and modification. A concise specification framework provides a high-level abstraction, permitting a variety of different representations — for example, voxels, meshes and implicit functions — to coexist in the same environment. Additionally, a procedural definition can be used as an intermediate format for capturing, editing, and replaying interactive editing sessions.

In spite of the above developments, today's model generation tools are primitive in that they generally lack a formal specification framework. This stands in stark contrast to commonly available rendering systems, such as RenderMan [Hanrahan and Lawson 1990, Upstill 1990], in which lighting, materials, objects, and shading are specified procedurally. Just as these rendering systems provide a framework for light transport simulation, my work creates an analogous framework for physical processes and other operators that shape and modify solid geometry. Powerful simulation tools, such as the Finite Element Method (FEM) or particle systems, can be embedded as modeling operators within this procedural framework. And high-resolution scanned models are a natural starting point for the procedural creation and manipulation of complex realistic objects within this framework.

1.2 Thesis Statement

This thesis introduces and explores the power of a procedural approach for authoring solid models. A procedural modeling language provides a concise and expressive framework for specifying the geometric and material properties of solid objects and applying a palette of physically-inspired simulation operators that can be used to vary these attributes as a function of time. A tetrahedral infrastructure, which supports the procedural definition of the modeling components of the language, facilitates the construction of volumetric models from high-resolution scanned surface meshes.

1.3 Contributions

The main contributions of this thesis are:

- A language for authoring solid models constructed from layers of material applied to surface models, such as scanned meshes. Multiple intersecting surface models can be included, with several options for specifying the materials in the overlapping regions. The user can procedurally control variations in the thickness of the layers and the decomposition of a layer into different materials (Chapter 4).

- Procedural interface for the definition and application of complex physically-based sculpting and simulation operators (Chapter 5).

To implement this procedural solid modeling language, I developed a volumetric infrastructure based on tetrahedral meshes. The contributions of my modeling system include:

- Tetrahedral infrastructure to support interactive visualization and manipulation of solid models (Chapter 6).

- Robust implementation for constructing tetrahedral representations of models described in the language (Chapter 7).

- Simplification of tetrahedral meshes, while preserving the topology and details of material/air and material/material boundaries and improving the shape and proportion of elements to meet the requirements of physical simulations. My implementation of this algorithm handles large tetrahedral models with over a million tetrahedra (Chapter 8).

- Interactive visualization and modification of tetrahedral models. The sculpting actions performed within the interactive system are automatically logged to a script in the language for offline replay on higher resolution models (Chapter 9).

Chapter 2

Related Work

My dissertation synthesizes and extends work in the three-dimensional modeling, editing, sculpting and simulation subfields of computer graphics. In this chapter I describe these areas of research and present representative papers in each. First, in Section 2.1, I list a wide variety of the procedural techniques used in this field, which were an inspiration for my research. In my language, procedural modeling forms the high-level framework for authoring solid models and allows control over both the shape of the model and the operations that may be applied to the objects.

Beneath the high-level procedural framework, three-dimensional models can be represented by a number of different data structures. In Section 2.2, I discuss methods for modifying the surfaces of objects. Sometimes a surface representation is not sufficient to capture the desired manipulations, in which case one turns to volumetric modeling. In Section 2.3, I describe the variety of data structures that have been used to represent volumetric models, and I outline the various methods for constructing the particular volumetric data structure I have used in my work, the tetrahedral mesh.

Finally, physical simulations and interactive modeling applications that make use of these solid data structures are presented in Sections 2.4 and 2.5. I have created a procedural solid modeling system that incorporates, as modules, many of the interesting ideas developed in these areas of research.

2.1 Procedural Techniques

Procedural modeling techniques have proved valuable in several specific domains of computer graphics [Ebert et al. 1998]. For example, biological patterns have been a great inspiration and have led to procedural models for the development and structure of flowers, trees, pine cones, seashells, and many other natural objects [Smith 1984, Prusinkiewicz 1986, Prusinkiewicz et al. 1988, Deussen et al. 1998, Prusinkiewicz and Lindenmayer 1990, Buchanan 1998].

Procedural modeling is a convenient method for generating large amounts of data that can be used to render complex photo-realistic scenes. An artist can specify this detail with a small amount of code by leveraging standard functions, such as noise and turbulence [Cook 1984, Peachey 1985, Perlin 1985, Perlin and Hoffert 1989, Lewis 1989, Worley 1996]. Like fractals, procedural modeling can specify an unlimited amount of detail, allowing objects to be viewed close-up without loss of resolution [Demko et al. 1985, Oppenheimer 1986, Musgrave et al. 1989].

Since the definition of each object is procedural, geometry can be generated in a view-dependent way. For example, an entire city can be defined procedurally [Parish and Müller 2001], but only be instantiated where necessary, or the bricks holding up a building only created on the visible facades [Legakis et al. 2001]. Procedural modeling encourages reuse of modeling primitives and the development of a library of styles. Furthermore, the compact procedural representation of an object can be mutated or evolved to create a wide range of new objects [Sims 1991, Sims 1994].

Unfortunately, it is difficult to create a virtual replica of a particular object using procedural techniques alone. With a language for procedural specification, such as the RenderMan shading language [Hanrahan and Lawson 1990, Upstill 1990], experienced programmers can build just about anything; however, the process may be tedious. Also, depending on the structure of an object's definition, it may be very difficult to edit small details of the object without modifying the entire model.

With the procedural solid modeling language described in Chapters 4 and Chapter 5, I cater not only to the advanced programmer, but also to the novice user by accepting high-resolution scanned surface models as input, and providing an interactive interface in which sculpting actions can be sketched. Throughout this chapter I present different techniques for modeling, some of which I have incorporated into the system, and discuss their advantages and disadvantages.

2.2 Surface Manipulation

In order to create realistic-looking objects, many different techniques have been developed for constructing and modifying three-dimensional models. Painting programs were some of the first applications to make use of a mouse or other two-dimensional input devices. A number of important papers have explored the possibilities of three-dimensional surface manipulation, including three-dimensional painting, displacement editing, manipulation of control points, constructive solid geometry and implicit surfaces.

2.2.1 Three-Dimensional Painting

Hanrahan and Haeberli [1990] introduced the notion of three-dimensional digital painting, which encompasses surface reflectance properties such as diffuse and specular colors, texturing and bump mapping. They implemented surface texturing as a direct process to avoid the distortions caused by painting a two-dimensional texture and then mapping it to the actual object. An important contribution of their work was outlining the different ways to interpret two-dimensional gestures in the three-dimensional world of the object. They defined a variety of coordinate systems in which the brushes could operate, including parameter space, screen space and tangent space. Agrawala et al. [1995] developed a system that allows the user to move a tool over a real three-dimensional object to apply paint to its virtual double. The project made use of a six-degree-of-freedom tracking device. These systems introduced techniques that allow the user to operate a tool as if it existed in the real world. Likewise, as I built the user interface for the interactive portion of our modeling system, I maintained a focus on the needs of the artist as he interacts with a solid model.

Some researchers strive to bring real-world surface properties to the virtual world. Strassmann [1986] varied the texture of paint applied to a surface by approximating the geometry of different brushes, the angle and pressure between the bristles and the paper, and whether the bristles were wet or dry. Curtis et al. [1997] developed a physically-based simulation of watercolor painting by modeling the texture of the different papers, the fluidity of colored water and the layering of pigments with very convincing results. These simulations are interesting because they identify the aspects of real world materials that create different visual appearances. Additional simulation techniques are discussed in Section 2.4.

2.2.2 Displacement Editing

Pushing the notion of "surface painting", one can edit the *displacement map* of the surface to modify both the large scale and micro-geometry of the object surfaces. Williams [1990] introduced a method for painting height fields to allow the user to touch up scanned geometries or create new models. This early interactive sculpting program explicitly maps two-dimensional operations to a three-dimensional environment; however, the level of indirection makes it difficult to create complex shapes. Also the topology of the object is restricted to match the original surface and "undercutting" is not allowed.

2.2.3 Editing Surface Control Points

Another option for creating interesting surface geometry is freeform deformation, which is performed by editing the control points of Bezier, B-spline or NURBS surfaces [Sederberg and Parry 1986, Coquillart 1990, Hsu et al. 1992, Terzopoulos and Qin 1994, Dachille IX et al. 1999, Raviv and Elber 2000]. Unfortunately, control point manipulation is not always intuitive. In the hands of an experienced artist, freeform deformations can produce intricate geometric models, but the process is labor intensive. The range of tools available for specifying and editing shapes is also limited, and the tools are rarely physically-based because the underlying geometry lacks information about the physical properties of the model, which are necessary for creating complex modifications. Furthermore, these deformations are not volume conserving and can create self-intersections that are difficult to detect or prevent. Performing topological changes to a model, such as drilling a hole through it, can be challenging using a surface description alone, as it requires changing the connectivities within the control point grid.

2.2.4 Constructive Solid Geometry / Implicit Surfaces

A more intuitive method of generating three-dimensional models is Constructive Solid Geometry (CSG), which models solid objects as a sequence of addition and subtraction operations on basic geometric shapes [Laidlaw et al. 1986]. Because this representation has an exact mathematical description, the finished shape has precise edges, even for extreme close-ups, and changes in topology are automatically handled. Mizuno et al. [1998] built a sculpting system from CSG operations;

unfortunately, the size of the model and cost of rendering grows with each sculpting action, so this system was limited to models with approximately 1,000 simple ellipsoidal tool operations.

Sometimes the sharp edges produced by pure CSG operations are undesirable. By instead evaluating the operations within a signed distance field, the surface at the intersection joint can be blended to make a *fillet* [Ricci 1973, Wyvill et al. 1986]. Models can be filtered with operations such as interpolation, averaging, blurring or combination with solid textures [Payne and Toga 1992]. The Level Set framework for manipulating distance fields [Sethian 1999] was used by Museth et al. [2002] to build an editing system capable of modifying high resolution scanned geometry. Language-based projects building on these ideas have been developed [Adzhiev et al. 1999, Wyvill et al. 1999].

Unfortunately, CSG operations are purely geometric and do not consider the physical or material properties of the object; thus, all objects behave similarly in response to manipulations. Because the shape has a mathematical construction, there are few possibilities for physical simulation within this modeling framework.

2.3 Volumetric Data Structures

While surface manipulation techniques can be used to create complex, high-resolution surface textures, the data structures for these techniques do not capture the volumetric properties of the material or allow modifications that dramatically change the geometry of the object. A more complete representation of the object is required to support the visualizations and physical simulations we envisioned for our project. Unlike surfaces, which are merely hollow shells, volumetric representations can capture the internal material structure of a model. A number of different volumetric data structures have been developed to capture this information, each with different advantages and disadvantages. Below I describe three general categories of volumetric models: axis-aligned voxels or octree models; "slabs" — a hybrid, thick surface representation; and tetrahedral meshes.

2.3.1 Voxels and Octree-based Volumes

The simplest implementation for volumetric modeling chops space into tiny cubes, called *voxels*, which were used in an early sculpting system by Galyean and Hughes [1991]. The voxels, which

store material information such as color and opacity, are arranged on a rectilinear grid and can be rendered using ray casting or by conversion to a polygonal representation [Lorensen and Cline 1987, Bloomenthal 1994]. Medical imaging data acquired as regularly-spaced two-dimensional slices can quickly be converted to a voxel representation. Such models can be rendered with varying opacities to visualize the internal structures. The topology of voxel models is not limited and may be modified interactively; thus, this representation is popular for virtual sculpting systems.

Volumetric models often lack visual fidelity because high-resolution volumes are necessary to represent a complex object, but memory constraints limit voxel models to low resolutions. For example, one of the larger models constructed by Wang and Kaufman [1995] was only 75x125x75. With special-purpose hardware, voxel models up to 512x512x512 samples can be manipulated interactively [Pfister et al. 1995, TeraRecon, Inc.]. However, even at the higher resolutions, the rendered models generally have blurred edges and rounded corners. While these rendering limitations may be an asset when modeling objects that are fuzzy or soft, the representation is not general purpose.

The storage requirements of a voxel representation do not scale linearly with surface quality. If the sampling along each axis is doubled, the surface will be represented with four times as many points but will require eight times as much memory. However, this disadvantage is ameliorated if an octree is used instead of a regular grid of voxels. Benson and Davis [2002] and DeBry et al. [2002] demonstrate that octrees can efficiently store surface textures with memory usage proportional to standard texture maps. The octree volume approach was demonstrated with Adaptive Distance Fields (ADFs) [Frisken et al. 2000, Perry and Frisken 2001]. Unlike a pure voxel approach with a fixed, uniform resolution, the ADF data structure can be refined to capture portions of the model that contain more detail, e.g., a level-10 ADF can represent details at $\frac{1}{2^{10}}$ the scale of the object.

Unfortunately, both voxels and ADFs are defined within a rectilinear coordinate system. If the model is deformed, the data must be shifted over cell boundaries to update the volumetric representation, which is both expensive and lossy. This representation can be used for efficient rigid body collision detection by representing each object with a separate set of voxels, and comparing the distance fields surrounding the objects [Lin and Gottschalk 1998]. However, the representation is not amenable to simulations requiring contact forces within the same object or the propagation of

cracks in a fracture, since the surface of the object is not explicitly represented. For these reasons, I chose not to use voxels as the primary representation for the models in my system.

2.3.2 Slabs

The slab data structure, introduced by Dorsey et al. [1999] and Agarwala [1999], combines the volumetric information of a voxel grid with a traditional surface-only representation. Each slab is a large six-sided element locally aligned with the original surface, containing a grid of samples. By concentrating the samples near the surface and leaving the interior of the object hollow, this representation lessens the scalability problem of a voxel model while still capturing important volumetric information. By locally aligning a grid to the surface, the sampling density perpendicular to the surface can be varied relative to the density along the surface.

Volumetric manipulations such as undercutting and holes can be performed in this data structure, as long as the modifications do not reach beyond the depth of the slab. The user must be aware of these limitations when performing sculpting operations. A simplified version of this representation that stores only a height field in each slab was used by Jagnow [Jagnow 2001, Jagnow and Dorsey 2002] to build a haptics-enabled, interactive model editing program.

Unfortunately, aligning the slabs to the surface is non-trivial and the representation is not appropriate for thin objects or objects with long skinny protrusions. If the general shape of the object is complicated and cannot be well approximated by a reasonable number of slabs, the advantages of locally-aligned volumes will be lost. Implementation and initialization of the data structures are complicated, and transformation operations within the volume are non-linear, because the slab volumes are often trapezoidal in shape. Also, this data structure does not lend itself well to visual rendering or finite element modeling for the same reasons discussed above, and thus was not adequate for my project.

2.3.3 Tetrahedral Meshes

In my work, I have focused on the tetrahedral mesh data structure, a representation that has been widely used for physical simulations. Unfortunately, constructing interesting tetrahedral models that are suitable for simulation is a challenge and requires significant infrastructure. A thorough

survey of meshing literature has been compiled by Owen [1998], and the companion website includes a huge database of the different meshing implementations currently available. George [1999] also prepared a summary of tetrahedral meshing techniques. Below I list relevant work in mesh initialization, simplification, improvement and refinement.

Initial Tetrahedralization

The generation of tetrahedral meshes for various modeling and simulation tasks continues to be an area of active research. There are three basic techniques for meshing the interior of a three-dimensional surface: Advancing Front methods, Delaunay triangulation, and octree tetrahedralization.

Advancing Front The Advancing Front and Advancing Layers techniques directly tetrahedralize a volume by adding well-proportioned tetrahedra, one at a time, to the interior faces of a triangle mesh [Lohner 1988, Pirzadeh 1993, Pirzadeh 1996]. The position for the fourth vertex of each tetrahedron is chosen greedily to maximize the quality of its shape. This method works well when the initial triangle mesh is manifold, consists of well-proportioned triangles and has an appropriate distribution of faces. However, in many applications it is not necessary or even desirable to exactly match the input surface. For example, scanned meshes contain many more triangles than necessary, and the triangles are evenly distributed. It is generally preferable to generate a tetrahedral mesh with the smaller elements near the details of the surface or internal structures, and larger elements in the areas of low curvature. Also, it may not be possible to retain the fine details and exact triangulation of the input surface when creating a low element count model for interactive manipulation.

Delaunay triangulation Another approach is to compute the three-dimensional Delaunay triangulation of the vertices, and then override the Delaunay property by performing various operations, such as tetrahedral swaps, to match the original surface edges and faces [Baker 1989, Conraud 1995, Shewchuk 1997, Shewchuk 1998, Cavalcanti and Mello 1999, Fleischmann 1999, Shewchuk 2000, Shewchuk 2002]. Unfortunately, in the three-dimensional case, the Delaunay property alone is insufficient to guarantee well-shaped elements. Most models require post-processing to add additional vertices and make modifications to the ini-

tial meshing to improve tetrahedral shape. Also, as mentioned above, it may be unnecessary to match the input surface.

Structured mesh generation The most brute-force technique, structured grid or octree tetrahedralization, is disadvantageous because it produces an approximation of the original surface sampled at a finite resolution [Yerry and Shephard 1984, Nielson and Sung 1997]. Additionally, the method produces a large number of tetrahedra and the elements near the boundary can have very small volumes. However, the octree method is simple and straightforward to implement and will always produce a consistent, manifold mesh with non-negative-volume elements. Some of the disadvantages of the structured grid approach can be mitigated by using simplification and mesh improvement techniques.

To construct the models described in this thesis, I chose to use a structured method of tetrahedralization because it is robust and produces a model independent from the original surface triangulation, which, in this case, may be a high-resolution scanned mesh or an implicit surface. See Chapter 7 for the details of the implementation.

Simplification

Motivated by the need for interactive rendering and transmission of complex meshes, much work has been done to simplify triangular surface meshes while maintaining surface fidelity [Schroeder et al. 1992, Hoppe et al. 1993, Hoppe 1996]. Some of these ideas and techniques have been translated to volumetric meshes [Trotts et al. 1998, Cignoni et al. 2000, Chopra and Meyer 2002]; in particular, Progressive Meshes have been extended to tetrahedral models [Staadt and Gross 1998]. Unfortunately, these techniques have been used primarily for reducing the number of elements, and do not necessarily improve the quality of the mesh elements.

Mesh Improvement

Turning to specific mesh improvement techniques, Frey and Field [1991] describe a vertex relaxation operation that adjusts the vertex positions to improve element shape, Joe [1995] uses local transformations to improve Delaunay triangulations, and Freitag and Ollivier-Gooch [1997] show

that a combination of remeshing techniques is more effective at improving shape than any single type of operation.

In Chapter 8, I introduce an algorithm for simplification and mesh improvement that prepares models specified in my language for simulation and/or interactive manipulation.

Mesh Refinement

To increase the resolution of a mesh for improved simulation accuracy, and to implement geometry modification operations such as fracture and the removal of material, meshes can be locally refined. One approach is to use *regular subdivision* — bisecting each edge of a triangle to create four self-similar triangles, or bisecting each edge of a tetrahedron to create four self-similar tetrahedra and an octahedron which is split into four additional congruent tetrahedra. To make this scheme adaptive across the mesh, elements between different levels need irregular subdivision. Elements are labeled based on their subdivision so that irregular "green" elements may be re-meshed to regular "red" subdivisions on future operations [Bank et al. 1983, Bey 1995, Liu and Joe 1996].

Another approach is to perform one edge split at a time, and require that the longest edge of each element be split first [Adler 1983, Rivara and Levin 1992, Rivara 1996, Rivara and Hitschfeld 1999]. Though effective in two dimensions, unfortunately in three dimensions this approach can result in infinitely many similarity classes, and in general it does not maintain element quality. Liu and Joe [1994] present a bisection scheme that projects each tetrahedron to a special tetrahedron, upon which longest edge bisection results in finitely many congruency classes of tetrahedra, and therefore provides a lower bound on element quality. A local refinement algorithm based on this bisection is presented in Liu and Joe [1995]. Lower bounds on tetrahedral element quality from edge bisection can similarly be guaranteed by using a vertex ordering scheme [Maubach 1995, Maubach 1996], or by systematically marking the edges of tetrahedra [Arnold et al. 2000].

Unfortunately, these approaches alone are not sufficient to ensure well-proportioned elements if the system allows the arbitrarily-positioned edge splits used in sculpting and fracture. Further study is necessary to determine the best strategy for refinement in these cases. One possibility is to lazily re-mesh the object, as necessary, after refinement [Ganovelli and C.O'Sullivan 2001].

2.4 Simulations

One of the main motivations for having volumetric models of objects is to perform simulations that reveal the internal structure or otherwise demonstrate the physical properties of different materials. There are a number of physical simulations that have been used in computer graphics, many borrowed from other disciplines such as material science.

Particle systems were one of the first graphics simulation techniques used to tackle the animation of fire and water [Reeves 1983]. Each particle independently follows a set of equations dictating its motion, color, and extinction. Simulations for these materials have progressed amazingly, including higher resolution, physically-correct behavior and photo-realistic rendering. As for many areas of graphics, the user must now consider the trade-offs between physical accuracy and the desire for animator control and real-time feedback [Nguyen et al. 2002, Lamorlette and Foster 2002]. Also, new data structures are emerging, such as the hybrid particles and level sets approach used by Enright et al. [2002] to model both the large scale turbulence of water and the intricate local surface structure.

Physically-based animation now includes more than simple rigid body dynamics. O'Brien and Hodgins [1999] introduced new simulation techniques for brittle fracture and Carlson et al. [2002] demonstrated a technique to simulate highly viscous liquids and melting solid objects. Cloth has always been a very challenging object to simulate [Baraff and Witkin 1998], because it requires a high resolution mesh to capture small folds, and very small time steps are needed to model the material's stiffness and avoid collisions [Bridson et al. 2002].

There has been significant interest in rendering imperfections and weathering effects on virtual objects for photorealistic scenes, such as staining and patinas [Dorsey et al. 1996, Dorsey and Hanrahan 1996], erosion [Dorsey et al. 1999], corrosion [Merillou et al. 2001], and cracking and peeling [Hirota et al. 1998, Lefebvre and Neyret 2002, Paquette et al. 2002]. These techniques make use of a wide range of different data structures and simulation methods. Some are strongly physics-based, while others are physically-inspired and aim to model only the most important characteristics of the resulting object. Most of these techniques run offline after the initial conditions for the simulation are set. Some allow limited user direction by specifying with masks the initial material properties.

All of these effects would make compelling interactive applications if the algorithms could be optimized and/or simplified for online use and integrated with real-time artistic controls. Additionally, these effects have previously only been demonstrated in isolation. My system allows these simulations to be brought together in one application and facilitates the use of these simulations in an interactive manner.

2.5 Interactive Volumetric Modeling

Many interesting interactive modeling systems have been built using both surface and volumetric data structures. A common problem for these three-dimensional editing environments is a non-intuitive user interface, since most computers are limited to two-dimensional input from the mouse and two-dimensional output to the screen. These limitations can be addressed by rendering shadow-like projections of the object and cursor on the sides and floor of a box surrounding the object [Raviv and Elber 2000] or with a custom three-dimensional display. To solve the input problem, Agrawala et al. [1995] uses a six-degree-of-freedom input device that tracks both the position and orientation of the cursor. Galyean and Hughes [1991] experiment with a number of different three-dimensional input devices, some of which include force feedback to let the user know when the cursor is near the surface by resisting attempts to penetrate the surface with the cursor. Commercially-available force feedback haptics devices [SensAble Technologies] have been used in a number of other interactive sculpting systems [Agarwala 1999, Dachille IX et al. 1999, Jagnow and Dorsey 2002]. We have incorporated some of these techniques in our interactive system, which is described in Chapter 9.

A number of techniques have been used to improve the speed of tetrahedral simulations for interactive applications. For example, in a multi-resolution model [Debunne et al. 2001], the coarsest resolution is used when possible for maximum efficiency, and higher resolutions are swapped in locally, as needed, to improve the accuracy of object deformations. One added difficulty with the multi-resolution approach is correctly handling cuts at all levels of the hierarchy, and remeshing as necessary [Ganovelli et al. 2000, Ganovelli et al. 2001]. Often it is possible to use approximations or simplifications of the most physically-correct simulations to obtain reasonable interactive results. For example, Smith et al. [2000] used a spring-mass model to perform real-time frac-

tures. By using a hybrid technique between rigid body simulation and static analysis, Müller et al. [2001] achieved both speed and stability for the fracture of stiff materials. In later work, Müller et al. [2002] augmented the linear deformation equations to correct for artifacts in rotation. The new algorithm has the speed and stability of the linear system while mimicking the behavior of the non-linear equations. Several examples computed with the last two algorithms, which were developed in our system and use models designed with the language, are presented at the end of Chapter 10.

2.6 Chapter Summary

This thesis grew out of a larger group project to create a physically realistic sculpting system. As I began develop the infrastructure for this project and reviewed the related work, I realized that there was an obvious hole in the creation of interesting models for simulation. Specific complex geometry is difficult to design procedurally or create from scratch with surface modeling programs. Mesh scanning technology [Cyberware] facilitates the creation of digital copies of real-world objects, but there is a lack of tools available to modify them. It was a natural decision to merge these technologies.

I have developed a language which, similar to RenderMan [Upstill 1990], allows the user to procedurally define features of the object. In the next chapter I present an overview of my procedural approach for authoring solid models.

Chapter 3

Modeling System Overview

To explore the procedural specification of solid models, I developed a modeling system that consists of two major components: a modeling language and a tetrahedral infrastructure. The basic pipeline of my approach is shown in Figure 3.1.

The first part is the procedural solid modeling language, which allows the specification of layered volumes from standard surface models such as scanned meshes, Computer-Aided Design (CAD) models and implicit surfaces. The materials that fill these layers are defined within the

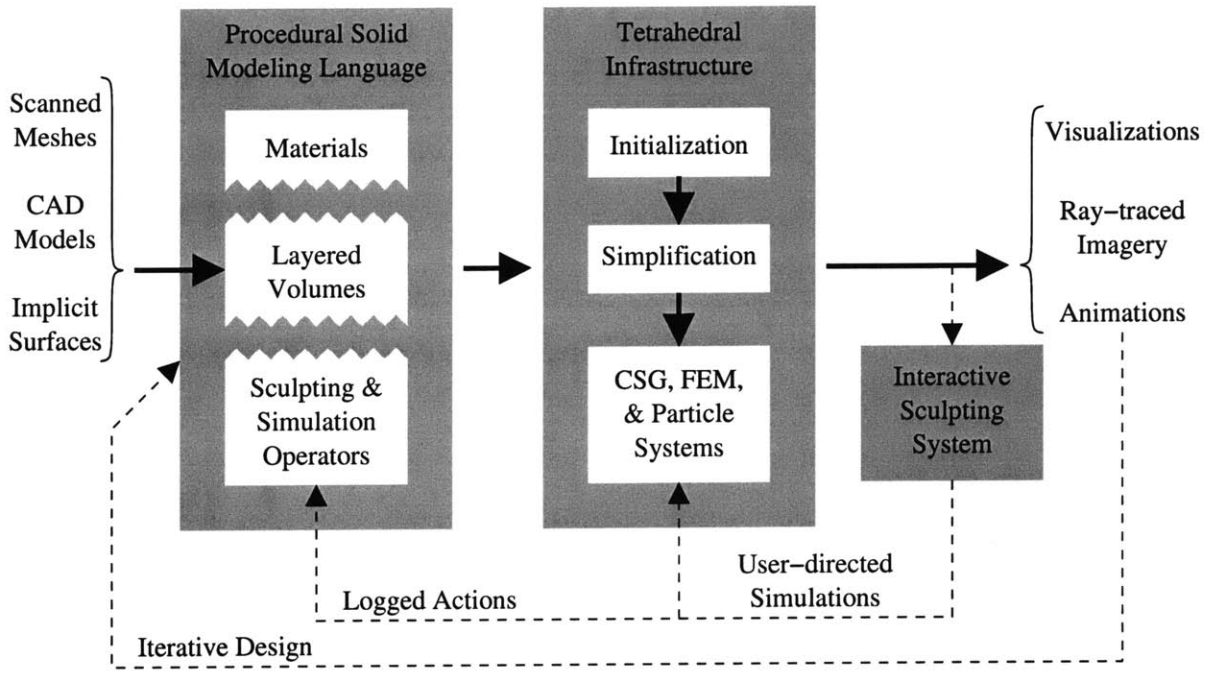


Figure 3.1: System Overview

language. The language also includes an interface for the specification and application of sculpting and simulation operators that can be used to vary the physical properties of an object over time. In Chapter 4, I describe the layered volume technique and the definition of interesting materials for these models, and in Chapter 5, I present the operator interface for sculpting and simulation tools.

The second part of my system is the tetrahedral infrastructure I have created to demonstrate the language. Realization of models designed in the language is accomplished in three steps: initialization, simplification, and modification using packages such as Constructive Solid Geometry (CSG), the Finite Element Method (FEM) and Particle Systems. Details of the tetrahedral data structure are presented in Chapter 6. A discussion of the structured mesh generation technique for initializing tetrahedral meshes is given in Chapter 7. These initial meshes must be simplified in preparation for simulation and interactive manipulation to reduce the overall number of elements and improve their shape, which I describe in Chapter 8.

Throughout this document I present examples of the output of my system, which includes volumetric visualizations, high quality ray-traced imagery, and animations. One of the key advantages of a language-based modeling representation is that the artist is encouraged to iteratively refine the model based on intermediate visualizations. I show this iterative design feedback loop with a dashed line in the system diagram. A second feedback loop shows how actions performed in our interactive sculpting system may be integrated within the design process. In Chapter 9, I present some of the visualization and interaction issues that were addressed in building the sculpting system. Several complex examples are shown in Chapter 10, and finally, in Chapter 11, I conclude by discussing these results and presenting areas of future work.

Chapter 4

Language for Volumetric Modeling

I have developed a procedural scripting language for authoring the geometric and material properties of a solid model and varying these attributes as a function of time. One of the main benefits of designing models in a language is the repeatable, iterative nature of the design process. At the end of the process, the user is left with a complete record of the steps needed to create a particular model. The final model need not be permanently stored (if space is limited), but can instead be recreated as needed. My language is also used as an intermediate representation for capturing, editing, and replaying interactive sculpting operations.

The process of writing a specification for the language focused my efforts and resulted in a concise and modular system. Different types and implementations of components (representations, rendering, sculpting, and simulations) can be used through a common interface for both interactive and offline applications. The system is designed to be extendable, as the full range of functionality is beyond the scope of this research.

In this chapter, I will present more details about the modeling language described in Cutler et al. [2002]. In Section 4.1, I describe how *layers of material*, the main building block of the language, can be constructed from a surface mesh. In Section 4.2, I demonstrate how multiple surface meshes can be combined within a single model to allow greater control over the shape of the internal structures. The boundaries between layers of material are defined within a signed distance field, which is discussed in Sections 4.3 and 4.4. In Section 4.5, I further emphasize the power of a procedural approach for modeling with sample material definitions. Finally, in Section 4.6, I describe some of the language implementation details. A specification of the language can be

Figure 4.1: Everyday objects whose internal structure is comprised of layers: citrus fruit, chocolate covered caramel and nut candies, and a stone architectural detail that has been recently damaged revealing visual differences between layers due to long term exposure.

found in Appendix A. In the next chapter, I will describe the operators portion of the language, which is used to modify the models.

4.1 Layers of Material

Many real-world objects are composed of *layers*: architectural framing, insulation and siding; the skeleton, muscles, and skin of an animal; or the peel of a fruit (see Figure 4.1). Traditionally, constructing complete volumetric representations of these objects has been a volumetric challenging task — providing motivation for my research. Building a physically-realistic model of any of these objects requires a description of the boundaries between materials and the variations within each material. Such a model could be created by an artist, but the process would be time-consuming. The data could be obtained using technology such as tomography or dissection approaches, but this requires an exact physical copy of the object and can be noisy or destructive.

I have developed a system to create volumetric models from readily-available surface meshes. My modeling language is based on the observation that the geometry of an object's internal structures can often be inferred from its primary interface (see Figure 4.2). The basic building block in the procedural modeling language is the *layered volume*. Within this framework, volumes can be combined, and layer composition can be controlled procedurally. Through a series of examples, based on a model of a simple chocolate candy, I show that the language provides a natural and expressive way to construct volumetric models. I have included relevant portions of the scripts used to generate the example images.

Figure 4.2: Often the boundaries between layers can be inferred from the primary interface. For example, a forensic anthropologist can reconstruct a face using only a skull and average muscle and skin thickness data. Images from Evison [1996].

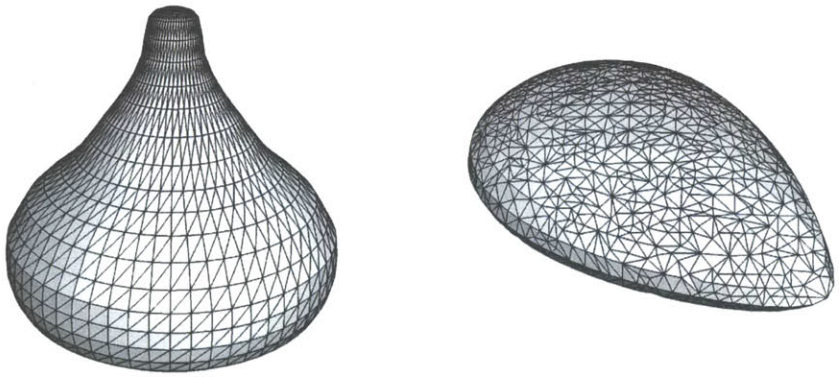


Figure 4.3: The candy and almond triangle meshes shown above (not to scale) are used to create the variety of solid models shown in Figures 4.4, 4.5 and 4.7.

The modeling language consists of variable assignments and nested procedure calls. The parameters to a procedure are specified as a list of name/value pairs within curly braces. These procedures may take a variable number of arguments which may appear out of order or be left unspecified if reasonable defaults have been assigned. As a convention in the examples, I use all capital letters to indicate user-defined functions and materials. A grammar for the language appears in Appendix A.1.

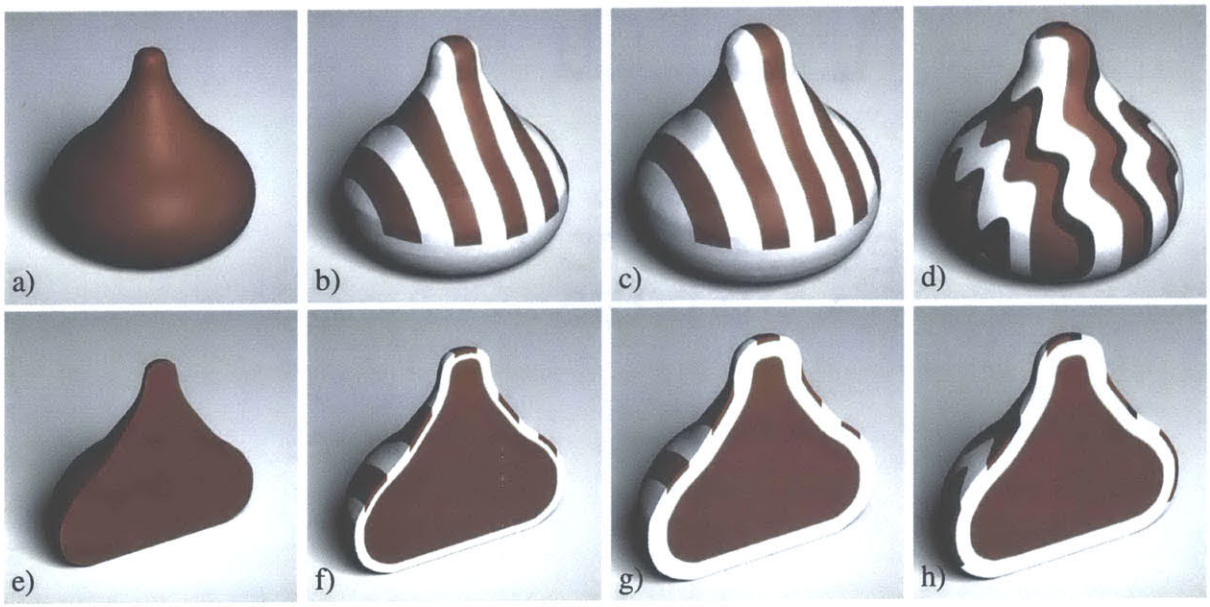


Figure 4.4: Using my modeling language, the user can quickly create a wide variety of models. Here is a sampling created from the candy surface mesh shown in Figure 4.3. To visualize the internal structures, a clip plane CSG tool has been used.

To construct a volume with interesting internal structure, layers of material are built from a surface model, such as those shown in Figure 4.3. In the first example, shown in Figure 4.4 a and e, a solid chocolate candy is created by filling the interior of a simple triangle mesh.

```
CANDY = volume {
  distance_field = surface_mesh {
    file = candy.obj }
  interior_layers = {
    layer {
      material = CHOCOLATE
      thickness = fill } } }
```

In the next step, I add two layers to the exterior of the mesh, shown in Figure 4.4 b and f. Each layer has a material type and thickness. The type and thickness can be uniform or vary procedurally. The thickness keyword `fill` can be used with a well-defined closed mesh (manifold) to describe an interior layer that is thick enough to fill the remaining interior space. The material keyword `nothing` can be used to describe a layer of air with no volumetric properties. In this example, the outermost layer has a procedural definition to create stripes of chocolate (see Section 4.5 for additional details).

```
STRIPED_CANDY = volume {
  distance_field = surface_mesh {
    file = candy.obj }
  interior_layers = {
    layer {
      material = CHOCOLATE
      thickness = fill } }
  exterior_layers = {
    layer {
      material = WHITE_CHOCOLATE
      thickness = 0.1 }
    layer {
      material = STRIPED_CHOCOLATE
      thickness = 0.1 } } }
```

In Figure 4.4 c and g, the thickness of the `WHITE_CHOCOLATE` layer has been increased to `0.3`, and in Figure 4.4 d and h, the material of the outermost layer is switched to `WAVY_CHOCOLATE`. Definitions for these materials are provided and discussed in Section 4.5.

4.2 Mesh Interactions

Many objects are more complicated than layers of material constructed from a primary interface. Often these objects can be easily described as a collection of overlapping shapes. To have more control over the shape of internal structures, the user can provide additional surface meshes and specify how they are combined to create the final volume. In the example below, the `precedence` construct is used to first create the volume for the almond, and then define the candy shape around the almond (Figure 4.5 a). Using nested precedence calls, subsequent shapes could be defined to fill the remaining unoccupied space.

```
ALMOND = volume {
  distance_field = surface_mesh {
    file = almond.obj }
  layers = {
    interior_layer {
      material = NUT
      thickness = fill } } }

STRIPED_ALMOND_CANDY = precedence {
  volume_1 = ALMOND
  volume_2 = STRIPED_CANDY }
```

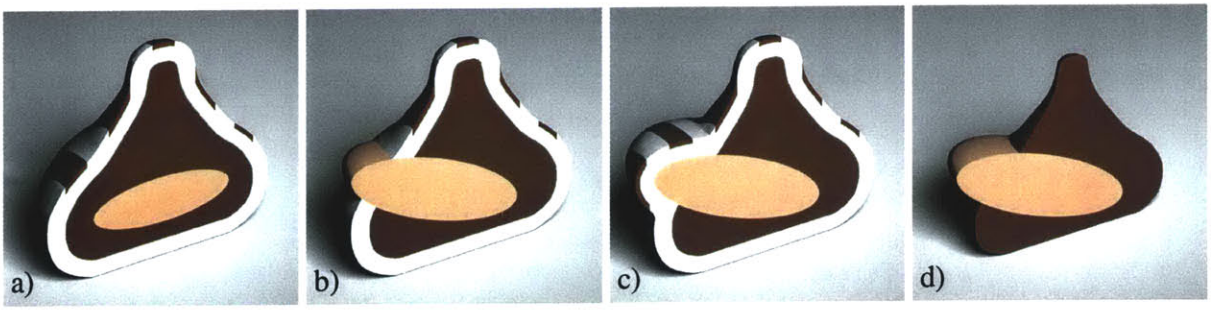


Figure 4.5: Using the `precedence` construct, the user can specify the interaction of multiple layered volumes created from different input surfaces. Here is a sampling of models created using both surface meshes shown in Figure 4.3.

The use of the precedence operator is particularly interesting when the original surface meshes intersect. In Figure 4.5 b the almond shape is larger and rotated so that it protrudes from the original candy surface and beyond the additional layers of material. However, the user may instead wish to describe a candy in which the outer layers are wrapped around the protruding almond as shown in Figure 4.5 c. To do this, the surface of an existing volume is used as the primary shape for a new volume. First, precedence is used to combine the almond shape with the interior layer of the chocolate (Figure 4.5 d). Then, the outer surface of the intermediate volume is used as the initializing surface for the new volume that adds the exterior layers of chocolate.

```
ALMOND_CANDY = precedence {
  volume_1 = ALMOND
  volume_2 = CANDY }

EXTERIOR_LAYERS = {
  layer {
    material = WHITE_CHOCOLATE
    thickness = 0.3 }
  layer {
    material = STRIPED_CHOCOLATE
    thickness = 0.1 } }

STRIPED_ALMOND_CANDY_2 = precedence {
  volume_1 = ALMOND_CANDY
  volume_2 = volume {
    distance_field = from_volume_surface {
      volume = ALMOND_CANDY }
    exterior_layers = EXTERIOR_LAYERS } }
```

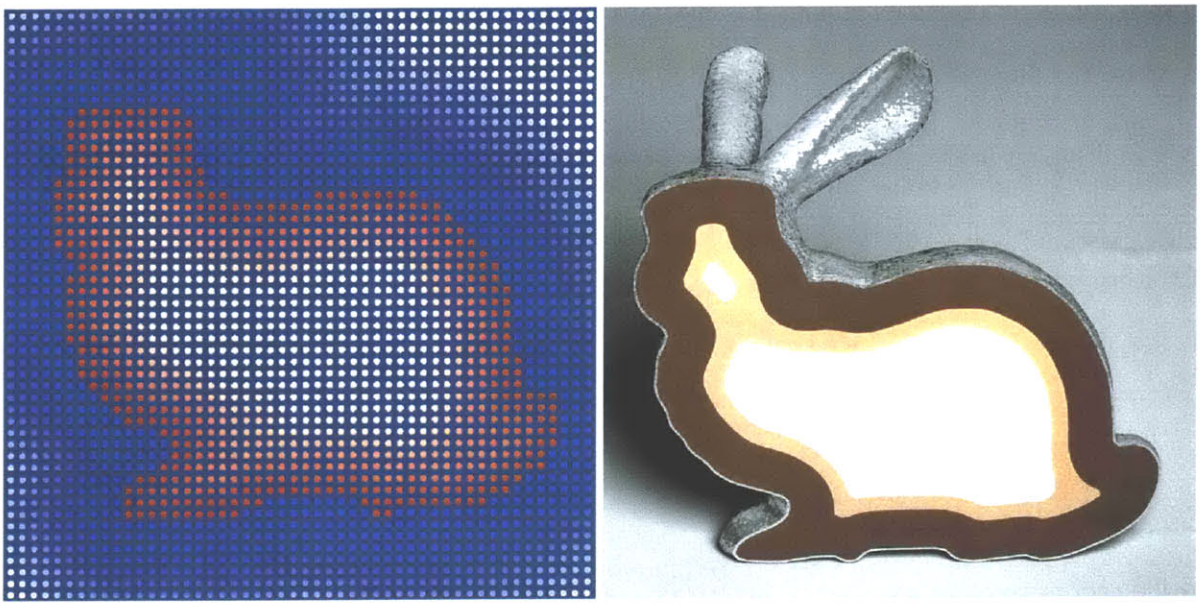


Figure 4.6: A visualization of a cut plane through the signed distance field of the Stanford bunny model and a simple layered model created from this field. Signed distance fields place no restrictions on layer thickness and seamlessly handle changes in topology without self-intersection problems.

The model may be equivalently defined using the convenience construct `precedence_layers`, defined in Appendix A.4:

```
STRIPED_ALMOND_CANDY_2 = precedence_layers {
  volume_1 = ALMOND
  volume_2 = CANDY
  exterior_layers = EXTERIOR_LAYERS }
```

4.3 Definition of Isosurfaces with the Signed Distance Field

Signed distance fields are a natural choice for describing and implementing layers in volumetric models. A signed distance field is a continuous scalar function defined throughout a volume, which can be used to compute offset isosurfaces. Often the value of this function is simply the distance to the closest point of some polygon mesh. Alternatively, the field can be initialized from an implicit surface. A slice through the distance field of the Stanford bunny is shown in the left image of Figure 4.6.

The set of all points with distance value v form the *v-isosurface*. The original mesh is the zero isosurface within the distance field, and the layers of a volumetric model are defined as ranges of distance values. Boundaries between the layers in the volume are isosurfaces that may be extracted using a technique similar to Marching Cubes [Wyvill et al. 1986, Lorensen and Cline 1987, Bloomenthal 1994, Nielson and Sung 1997].

Using a signed distance field to describe isosurfaces prevents self-intersection problems, seamlessly handles changes in topology, and places no restrictions on layer thickness, as illustrated in the example shown in the right image of Figure 4.6. The signed distance field can be efficiently evaluated on a uniform grid using the Level Sets Fast Marching Method [Sethian 1999], which is described in Chapter 7 of this dissertation.

Often a desired distance field for a volumetric model is most easily described by combining other distance fields using simple operators such as scaling, union (minimum), intersection (maximum), and subtraction. For example, in Figure 4.7 e, the union operator is used to combine the almond and candy meshes and then three layers are applied to the resulting distance field. The zero isosurface of the resulting distance field lies between the `CHOCOLATE` and `WHITE_CHOCOLATE` layers.

```
UNION_CANDY = volume {
  distance_field = union {
    distance_field_1 = surface_mesh {
      file = almond.obj }
    distance_field_2 = surface_mesh {
      file = candy.obj } }
  interior_layers = {
    layer {
      material = CHOCOLATE
      thickness = 0.4 }
    layer {
      material = PINK_FROSTING
      thickness = fill } }
  exterior_layers = {
    layer {
      material = WHITE_CHOCOLATE
      thickness = 0.3 } } }
```

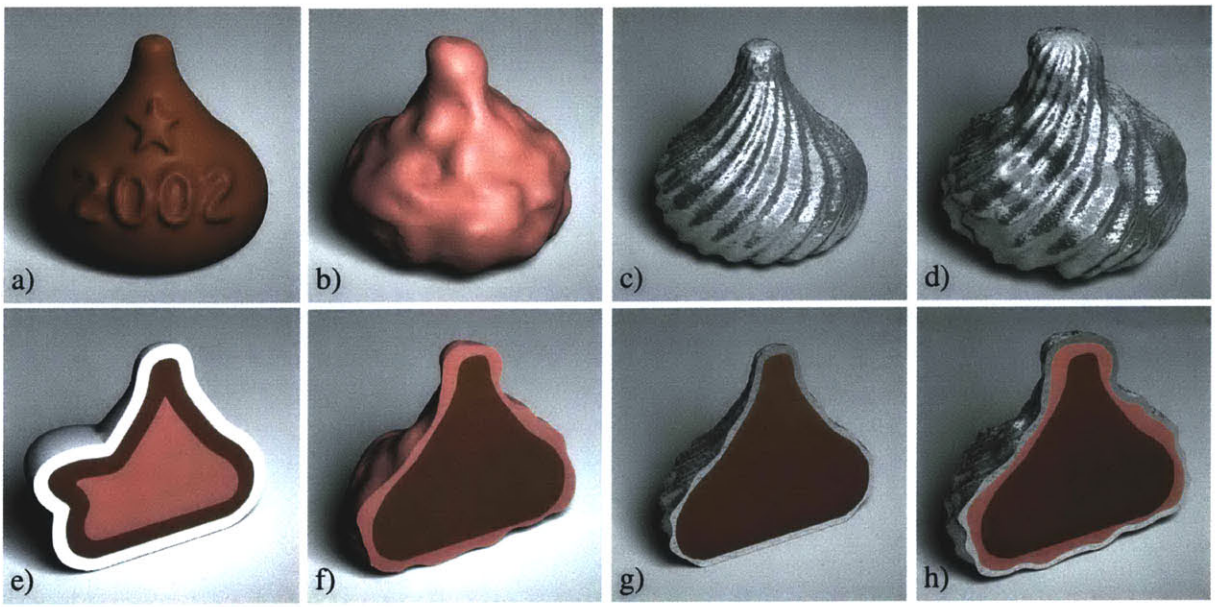


Figure 4.7: Layers are defined as ranges of distance values within a signed distance field. Distance fields may be combined using operators such as union, intersection, or subtraction, and non-Euclidean distance metrics can be used to define interesting thickness patterns.

4.4 Modified Isosurface Velocity

Usually, a distance field is simply a Euclidean measurement from each point to the original surface. Layers defined within this type of distance field will have a uniform thickness within each layer. However, it is often natural to describe layers that vary in thickness according to some pattern. To create interesting layers that have variable thicknesses, non-Euclidean distance metrics are defined by modifying the *isosurface velocity*. The spacing between isosurfaces in a distance field is greater where the velocity is higher. The user may define a pattern of increased velocity by painting on the surface. For example, the embossed effect shown in Figure 4.7 a is created from a texture map of a star and the text "2002". Darker areas of the texture correspond to higher isosurface velocity values and thicker areas of the layer. Museth et al. [2002] created a similar effect using a different signed distance field technique. Alternatively, the isosurface velocity can be defined procedurally. For example, the lumpy appearance shown in Figure 4.7 b and f is produced by a random turbulence function [Ebert et al. 1998], and the diagonal swirl pattern of Figure 4.7 c and g is created with the definitions below.

```
float DIAGONAL_VELOCITY(Vec3f &p) {
  float stripes = 20;
  float radius = 3;
  float a = atan2f(p.x(),p.z());
  float v = (sinf(a*stripes-3*radius*p.y())+1)/2.0;
  float r = sqrt(p.x()*p.x()+p.z()*p.z())/radius;
  if (r > 1) r = 1;
  return v*r + (1-r); }

FOIL_WRAPPED_CANDY = volume {
  distance_field = from_surface_mesh {
    file = candy.obj }
  interior_layers = {
    layer {
      material = CHOCOLATE
      thickness = fill } }
  exterior_layers = {
    layer {
      material = FOIL
      thickness = 0.2
      velocity = DIAGONAL_VELOCITY } } }
```

In these examples, the isosurface velocity is independent of the original mesh. The velocity may instead be defined from an estimate of the local surface properties (such as curvature, visibility or accessibility) at each point in the distance field. For example, the user could specify that a layer of moss should growth thicker on the shady side of a tree — that is, the portion of the model not visible from the main light source. To do this, the velocity function would also accept the distance field as an argument.

The isosurface velocity of a distance field initialized from a surface mesh is Euclidean, but this can be changed using a new non-uniform velocity (see Section 7.3.2). Also, a non-Euclidean distance field (e.g., initialized by an implicit surface) can be remapped to a Euclidean field with a velocity of 1. The new velocity is applied starting at a particular isosurface (usually the zero isosurface) of the original distance field. All layers defined within a particular distance field will have a thickness pattern based on that same isosurface velocity. In the example above, the velocity has been specified for a single layer, with a compact easy-to-read convenience construct (*syntactic sugar*). For more details, see Appendix A.4. Below is the desugared version. Since the velocity of a `fill`-ed layer is irrelevant, `DIAGONAL_VELOCITY` can be used for the entire field.

```
FOIL_WRAPPED_CANDY = volume {
  distance_field = from_surface_mesh {
    file = candy.obj
    velocity = DIAGONAL_VELOCITY }
  interior_layers = {
    layer {
      material = CHOCOLATE
      thickness = fill } }
  exterior_layer  = {
    layer {
      material = FOIL
      thickness = 0.2 } } }
```

In fact, this example still contains syntactic sugar, since modified isosurface velocities are implemented as a wrapper around another distance field (see Appendix A.4). This example fully desugars to:

```
FOIL_WRAPPED_CANDY = volume {
  distance_field = from_isosurface {
    distance_field = from_surface_mesh {
      file = candy.obj }
    isosurface = 0
    velocity = DIAGONAL_VELOCITY }
  interior_layers = {
    layer {
      material = CHOCOLATE
      thickness = fill } }
  exterior_layers  = {
    layer {
      material = FOIL
      thickness = 0.2 } } }
```

The examples given do not make use of the `isosurface` parameter in the `from_isosurface` construct. Details on the implementation of this parameter are given in Chapter 7.

If the layers require different isosurface velocities, volume specifications must instead be nested so that the layers will have different thickness patterns. For example, in Figure 4.7 d and h I build a layer of lumpy frosting from a turbulent velocity field followed by a layer of foil with a diagonal pattern. The effects of both isosurface velocities are apparent in the resulting surface. This type of specification is common enough to warrant a syntactic sugar construct, which desugars velocities specified per layer into nested volume specifications.

```
FOIL_WRAPPED_LUMPY_CANDY = volume {
  distance_field = from_surface_mesh {
    file = candy.obj }
  interior_layers = {
    layer {
      material = CHOCOLATE
      thickness = fill } }
  exterior_layers = {
    layer {
      material = PINK_FROSTING
      thickness = 0.1
      velocity = LUMPY_VELOCITY }
    layer {
      material = FOIL
      thickness = 0.05
      velocity = DIAGONAL_VELOCITY } } }
```

is equivalent to:

```
LUMPY_CANDY = volume {
  distance_field = from_surface_mesh {
    file = candy.obj
    velocity = LUMPY_VELOCITY }
  interior_layers = {
    layer {
      material = CHOCOLATE
      thickness = fill } }
  exterior_layers = {
    layer {
      material = PINK_FROSTING
      thickness = 0.1 } } }

FOIL_WRAPPED_LUMPY_CANDY = precedence {
  volume_1 = LUMPY_CANDY
  volume_2 = volume {
    distance_field = from_volume_surface {
      volume = LUMPY_CANDY
      velocity = DIAGONAL_VELOCITY }
    exterior_layers = {
      layer {
        material = FOIL
        thickness = 0.05 } } } }
```

The combination of procedurally-defined isosurface velocities and interactions between multiple meshes (Section 4.2) provides the user with many options for describing the internal structures of a solid model.

Figure 4.8: Material variations within a layer, such as brick and mortar, concrete particles and wood grain, can be procedurally defined within the script by the user. Images from Juracek [1996].

4.5 Procedural Material and Layer Definition

Simple materials are defined by a list of the necessary rendering and simulation parameters. I provide a small library of built-in materials, and additional materials can be defined within the script file as shown below. Default values are assigned to any unspecified parameters.

```
CHOCOLATE = material {
  name = "CHOCOLATE"
  diffuse_color = { 0.31 0.17 0.15 }
  shininess = 10
  density = 1100       /* kg/m^3 */
  elasticity = 1.0e5   /* N/m^2 */ }
```

However, a layer need not be composed of a uniform material. Figure 4.8 shows a few examples of layers containing spatially-varying materials. The user can procedurally define a continuous variation of material properties — that is, a *solid texture* for use during interactive and offline rendering [Peachey 1985, Perlin 1985]. If the details of these variations are important to the simulation, the volumetric representation for the layer is subdivided into discrete materials. Examples include a brick and mortar layer covering the walls of a building or the patterns of chocolate on the candy in Figure 4.4 c and d. Below are the definitions used to create the striped and wavy layers of chocolate. If different material properties are assigned to the different types of chocolate (such as melting temperature), these spatially-varying properties can be taken into account during simulation. However, it is impractical to represent highly-detailed or continuous material variation, such as concrete particles or wood grain, as distinct materials within a traditional volumetric data structure. This is a subject for future work.

```
Material* STRIPED_CHOCOLATE(Vec3f &p) {
  if (p.y() < 0.8) Lookup("WHITE_CHOCOLATE");
  if ((p.x() >= -2.25 && p.x() <= -1.75) ||
      (p.x() >= -1.25 && p.x() <= -0.75) ||
      (p.x() >= -0.25 && p.x() <=  0.25) ||
      (p.x() >=  0.75 && p.x() <=  1.25) ||
      (p.x() >=  1.75 && p.x() <=  2.25))
    return Lookup("CHOCOLATE");
  return Lookup("WHITE_CHOCOLATE"); }


Material* WAVY_CHOCOLATE(Vec3f &p) {
  if (y < 0) return Lookup("WHITE_CHOCOLATE");
  x = 13*(x+0.25*cosf(5*z));
  if (x > 0) { x = int(x)%16; }
  else { x = int(-x)%16; x = 16-x; }
  if (x <= 6) return Lookup("CHOCOLATE");
  if (x <= 9) return Lookup("DARK_CHOCOLATE");
  return Lookup("WHITE_CHOCOLATE"); }
```

The complex arrangement of materials constructed with these definitions could also be constructed by specifying boundary meshes for each stripe of chocolate. Deciding whether to use separate surface meshes or a procedural definition to represent these types of variations is up to the user, and may depend on the intended application.

4.6 Language Implementation Details

The modeling language described in this chapter is more than just a simple list of parameter assignments. To implement the full features of the language, I chose to leverage an existing programming language. I designed a simple syntax for the main components of the language and allow the user to include C functions to interface with the core system as needed. I implemented the language using lex and yacc [Levine et al. 1992], which are popular tools for scanning and parsing special-purpose languages.. The specification of the language is given in Appendix A.

To create a model, the user places the necessary definitions in a script file and calls my program with one argument, the file name. Additional files of definitions shared with other scripts may be included using the standard `#include` interface. Arbitrary C functions to define material variations and other procedurally-controlled components of the language can be included within the script file between `#BEGIN_C_CODE` and `#END_C_CODE` markers. These functions are com-

piled and linked at runtime to interface with the core system. The name space of variables in the scripting language and auxiliary C functions is shared.

To call a C function from the scripting language, the user specifies the name of the function and a list of `name = value` argument pairs within curly braces. The arguments may appear out of order, or be left unspecified if a default value is given. To be used as a procedural definition for a distance field, isosurface velocity or material, the function must match the prototype given in the grammar. The `Lookup` function gives access to script file variable assignments from within the C functions. Script variables are not typed, but the implementation does perform dynamic type checking at function calls.

4.7 Chapter Summary

I have described a simple procedural framework for initializing interesting volumetric models. The main ideas of the language are adding layers to a primary interface, describing the interactions between different meshes, changing the isosurface velocity of the signed distance field to create layers with different thickness patterns, and procedural definitions for material variation. I have demonstrated these language features on a simple example of a chocolate candy. More complex and interesting examples of models built with these tools are presented throughout the remainder of this thesis. In the next chapter, I will describe the modular operator language that is used to modify these objects with a variety of physically-inspired sculpting tools.

Chapter 5

Sculpting and Simulation Operators

In the previous chapter I demonstrated how the authoring language can be used to design the layers of material that form the internal structures of solid models. The advantages of a volumetric representation for virtual objects become apparent when these objects are visualized and modified in complex ways. A sampling of the different simulation techniques that have been developed for weathering, deformation and fracture were presented in Chapter 2. These simulation techniques have previously been demonstrated only in isolation, with dedicated systems for the desired effect. However, by incorporating implementations of these techniques into the same framework, their effects can be combined. By expressing simulation operators within the context of a modeling language, the user can also directly control the application of each simulation technique as an editing tool. Finally, advanced users can modify the behavior of a tool for a novel application, or can create new tools that exercise different features of the simulation libraries.

In Section 5.1, I introduce a gargoyle model, which is used throughout the chapter to illustrate a variety of different user-controlled simulation operators. In Section 5.2, I explain how abstraction allows a novice user to apply complicated tools without a deep understanding of the underlying data representation or simulation algorithms. Examples of how to specify new simulation behaviors are given in Section 5.3. Finally, in Section 5.4, I explain how the interactive sculpting system can be used in conjunction with the operator language for more effective modeling.

5.1 A Complex, Multi-Stage Simulation Example

To demonstrate the wide variety of modification operators that can be applied within the language, I present as a running example a sequence of weathering effects applied to a gargoyle statue mounted on the exterior of a building. Gargoyles are subjected to interesting flow patterns because they were originally used as decorative downspouts to direct rainwater away from the exterior building facades. Long-term exposure causes a variety of effects on these exterior architectural details, including discoloration, weakening, erosion, biological growth and fracture due to the freeze/thaw cycle. Figures 5.2 and 5.3 outline the multi-stage simulation performed on this object.

The initial model is relatively simple, consisting of a single layer of stone applied to the interior of a high-resolution scanned triangle mesh. Since the sculpting and simulation operations I had planned would not penetrate deep into the model, a hollow model was created to save memory. The initial mesh is shown in Figure 5.2 a and b.

5.2 Usability through Abstraction

One of the main obstacles that a user must overcome in order to use a particular simulation package is determining proper values for the numerous parameters needed to control the system. Different implementations of the same simulation technique may require different sets of parameters. The primary goal of the tool interface is to provide abstraction and standardization so that the user may apply complex simulation operators to the model without studying the details of each implementation.

From the user's point of view, all tools have the same interface and accept the same intuitive parameters, such as position, orientation and size. Each tool makes one or more calls to simulation modules, which map these standard parameters to the specific parameters required by their particular implementations (see Figure 5.1). A new sculpting or simulation module is added to the language by creating a custom link between the simulation library and the core system. Also, a set of sample tools that exercise the various features of the simulation are created. Below I give definitions for two simple tools:

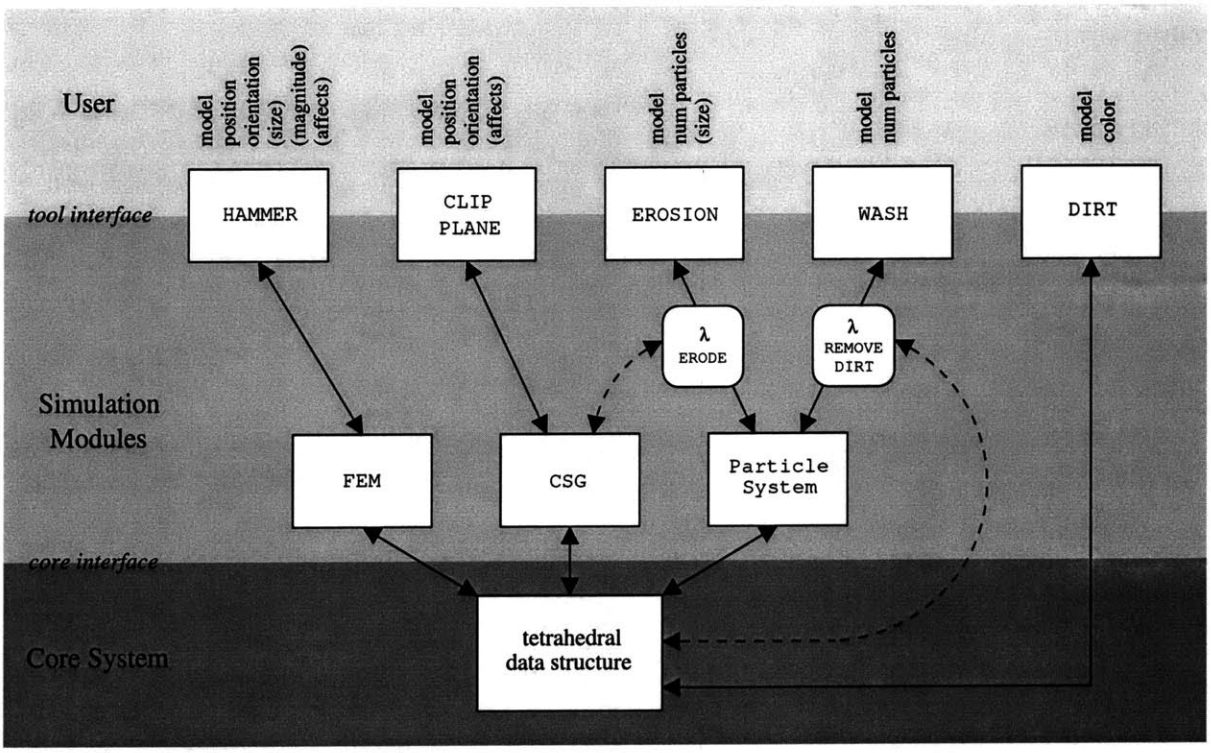
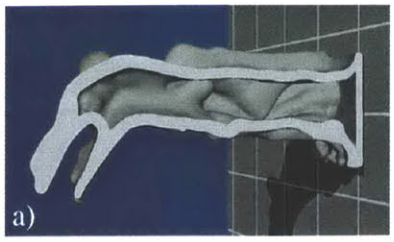


Figure 5.1: The user interacts with tools that export standard interfaces. These tools call one or more simulation modules, which interact with the core representation. Parameters to the particle system include functions (λ), which can call other simulation packages or directly modify the data structures.

```
void CLIP_PLANE(Volume *model,
                Vec3f position,
                Vec3f orientation,
                List<Material*> *affects = NULL) {
  AppliedArea *a = HalfSpace(position,orientation);
  CSG(model,a,NULL,affects); }
```

The `CLIP_PLANE` tool was used in the previous chapter to visualize the internal structure of the volumetric candy meshes. The third argument (whose type is `Material`) to the Constructive Solid Geometry (CSG) module indicates whether this operation is a subtraction or an addition operation. If the value is `NULL`, a subtraction is performed; otherwise, volume of the specified material is added.

```
GARGOYLE = volume {
  distance_field = surface {
    file = gargoyle.obj }
  interior_layers = {
    layer {
    material = STONE
    thickness = 0.5 } } }
```

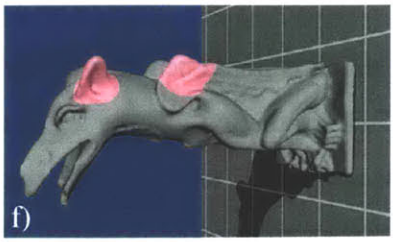



```
DIRT {
  model = GARGOYLE
  color = { 0.5 0.5 0.5 } }
```


```
WASH {
  model = GARGOYLE
  num_particles = 200000 }
```

Figure 5.2: A sequence of images from the gargoyle simulation and the corresponding part of the script used to generate each step: a) a cut plane reveals that the initial model shown in b) is hollow, c) applying a layer of dirt, d) particles that flow over the surface in the direction of gravity, and e) after washing away the dirt.

```
HAMMER {
  model = GARGOYLE
  position = { -0.7 1.2 0.7 }
  orientation = { -0.2 -0.4 0.8 } }
HAMMER {
  model = GARGOYLE
  position = { -2.5 1.0 1.0 }
  orientation = { 0.5 -0.1 -0.8 } }
```




```
ERODE {
  model = GARGOYLE
  num_particles = 2000 }
```


```
BIOLOGICAL_GROWTH {
  model = GARGOYLE
  num_particles = 40000 }
```

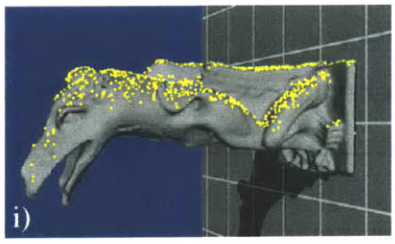


Figure 5.3: Continuing the sequence from Figure 5.2: f) two hammer hits are interactively positioned to g) remove the gargoyle's ear and a corner of its wing, h) a particle simulation of erosion affects top surfaces of the model and i) a different particle motion function is used to j) add color suggestive of biological growth.

```
void HAMMER(Volume *model,
            Vec3f position,
            Vec3f orientation,
            float size = 0.02,          /* meters */
            float magnitude = 1000,     /* Newtons */
            List<Material*> *affects = NULL) {
  Vec3f force = orientation; force *= magnitude;
  AppliedArea *a = GaussSphere(position,size);
  FEM(model,a,force,affects); }
```

The HAMMER tool applies a single impact force to the object, which is implemented with the Finite Element Method (FEM) [Zienkiewicz and Taylor 1989]. This simulation module applies a Gaussian distribution of forces to the model, computes the induced stresses, and performs any necessary fractures [Müller et al. 2001]. An example of the HAMMER tool usage is shown in Figure 5.3 f and g to crack off the gargoyle's ear and a corner of its wing. The HAMMER tool has three optional parameters, which are not used in the gargoyle example. To create damage similar to that from a larger and heavier *sledgehammer*, the user can override the default values, as in the following call:

```
HAMMER {
  model = GARGOYLE
  position = { 0 0 0 }
  orientation = { 1 0 0 }
  size = 0.05
  magnitude = 10000 }
```

Alternatively, the user could create a new tool called a CHISEL, which is similar to the HAMMER, except for the shape of the force that is applied.

An optional argument for all calls to the CSG and FEM modules is the *affected materials list*. Only the materials so listed will be modified by the operator. For example, by specifying that a CSG tool only affects skin and muscle, the skeleton of an animal can be exposed for visualization. Or by excluding a particular material from the list for a HAMMER operation, the material becomes unbreakable. If the `affects` list is NULL, all materials are subject to modification. More examples of this feature are presented in Chapter 10.

Of course there is a limit to abstraction, and some simple tools are best implemented by directly manipulating the underlying data structures. For example, the gargoyle model is modified in Figure 5.2 c with the DIRT tool, which darkens the color of all exposed material. To implement

the DIRT tool for the tetrahedral infrastructure presented in Chapter 6, which is shaded by interpolating color values stored at the vertices, all vertices on the exterior of the mesh are set to the new color. The tool definition is as follows:

```
void DIRT(Volume *model, Vec3f color) {
  List<Vertex*> vlist;
  model->CollectVertices(vlist);
  for (int i = 0; i < vlist.numElements(); i++) {
    Vertex *v = vlist.getElement(i);
    if (!v->isSurfaceVertex()) continue;
    v->SetColor(color); } }
```

5.3 Defining Simulation Behavior

The power of a language for tool definition extends beyond copying and modifying existing tools. A language facilitates the specification of new behaviors for simulation. One type of simulation that offers many possibilities for flexibility is a particle system. Particle systems have been used in many different applications, including modeling water, fire, smoke, clouds and sand, to create a variety of effects that span a wide range of physical accuracy [Reeves 1983]. The behavior and complexity of a particle system simulation are controlled by the definition of particle initialization, motion, action and interaction functions. The basic structure of a particle simulation is presented in the pseudo-code below:

```
for i ← 0 to num_particles - 1
    particles[i] ← initialize(model);

while ∃ i such that particles[i] ≠ NULL
    for i ← 0 to num_particles - 1
        p ← particles[i]
        if (p ≠ NULL)
            action(model,p)
            motion(model,p,particles)
            if (life(p) = 0)
                delete p
                particles[i] = NULL
```

The first step in a particle simulation is to initialize the specified number of particles. In the gargoyle example, all particles are initialized at the outset, although some applications may require them to be created mid-simulation. The system iterates over all of the particles, adjusting their positions and performing any specified actions. At the end of each iteration, each particle that is no longer "alive" is removed from the system. When all particles have been removed, the simulation is complete. Prototypes for the various particle functions are given in Appendix A.3.

Below are definitions for the three particle simulations used in the gargoyle example. The `WASH` tool removes dirt from the statue based on rain flow patterns (Figure 5.2 d and e), the `EROSION` tool removes material from the top faces of the statue (Figure 5.3 h), and color, suggestive of lichen or moss, is added with the `BIOLOGICAL_GROWTH` tool (Figure 5.3 i and j).

```
void WASH(Volume *model,
          int num_particles = 10000) {
  Function *initialize = Lookup("VERTICAL_FALL");
  Function *action = Lookup("REMOVE_DIRT");
  Function *motion = Lookup("CLINGING");
  Function *life = Lookup("HUNDRED_ITERS");
  ParticleSystem(model,num_particles,initialize,
                 action,motion,life); }


void EROSION(Volume *model,
             int num_particles = 1000) {
  Function *initialize = Lookup("VERTICAL_FALL");
  Function *action = Lookup("ERODE");
  Function *motion = Lookup("RANDOM");
  Function *life = Lookup("TWENTY_ITERS");
  ParticleSystem(model,num_particles,initialize,
                 action,motion,life); }


void BIOLOGICAL_GROWTH(Volume *model,
                       int num_particles = 2000) {
  Function *initialize = Lookup("VERTICAL_FALL");
  Function *action = Lookup("ADD_LICHEN");
  Function *motion = Lookup("RANDOM");
  Function *life = Lookup("TWENTY_ITERS");
  ParticleSystem(model,num_particles,initialize,
                 action,motion,life); }
```

All three tools use the same particle initialization function, `VERTICAL_FALL`, which drops

each particle from a random point on the top face of the object's bounding box. The first surface it hits is the initial position of that particle. If desired, an alternate initialization function could be constructed for the BIOLOGICAL_GROWTH simulation to direct its effects on the shady surfaces of the statue.

The particle action function for the WASH tool changes the color stored at vertices near the particle's current position:

```
void REMOVE_DIRT(Volume *model, Particle *p) {
  Vec3f clean_color = Vec3f(1,1,1);
  List<Vertex*> vlist;
  float radius = 0.1;
  model->CollectVertices(vlist,p->pos(),radius);
  for (int i = 0; i < vlist.numElements(); i++) {
    Vertex *v = vlist.getElement(i);
    v->BlendColor(clean_color,p->pos(),radius); } }
```

Similarly, the ADD_LICHEN action function used by the BIOLOGICAL_GROWTH tool blends the vertex color toward a greenish hue. The particle action function can also perform more dramatic changes to the model; for example, the ERODE action function used by the EROSION tool uses the CSG module to remove a small chunk of material.

```
void ERODE(Volume *model, Particle *p) {
  AppliedArea *a = Sphere(p->pos(),0.01);
  CSG(model,a,NULL); }
```

The RANDOM particle motion function used for EROSION and BIOLOGICAL_GROWTH tools moves the particle in a random direction along the surface. In the CLINGING motion used by the WASH tool, defined below, smaller values for the falling angle result in flow that behaves with greater surface tension. Motion functions that compute interactions between particles need the list of all particles as an additional argument.

```
void CLINGING(Volume *model, Particle *p) {
  Vec3f n;
  model->NormalAt(p->pos(),n);
  if (n.dot(Gravity) > cos(p->FallingAngle()))
    p->Drip(model,Gravity);
  else
    p->MoveAlongMesh(model,Gravity); }
```

Finally, in these examples, the particle life function simply specifies that each particle is alive for a constant number of iterations.

```
int HUNDRED_ITERS(Particle *p) {
  p->setAge(p->getAge()++);
  if (p->getAge() > 100) return 0;
  return 1; }
```

The particle system tools developed for the gargoyle example were inspired by physical processes. The current implementations are simplistic, but with more extensive knowledge about the processes that create these effects, more physically-correct computations could easily be employed. I have proposed a general framework for sculpting and simulation operators and new system components can be substituted as they become available. The user does not need to be aware of these changes, as long as the new implementation exports the same interface.

5.4 Interactive Application of Operators

Sculpting and simulation operations can remain difficult to use and control, even with the abstraction and flexibility in specifying tool behavior described in the previous sections. For example, just choosing the appropriate position, orientation and size of these tools can be tedious for complex models.

To solve this problem, the user can sketch the desired simulations in an interactive modeling system, using a simplified version of the model. In the gargoyle example, the position and orientation for the HAMMER tool are chosen in this way. The language can be used as an intermediate format for the output of the interactive system. Actions are logged to a script using the operator interface described in this chapter, and thus are human-readable and can easily be modified for reprocessing offline. The information in these logfiles can be a starting point for iteratively fine-tuning the finished model.

Using an evolutionary design strategy [Sims 1994], scripts with slight variations can be generated and sent to idle machines for batch processing. Examples of simple modifications to the script include: adjusting the position, orientation or size of the tools, replacing calls to the HAMMER tool with the CHISEL tool, and changing the color of the dirt and lichen applied to the model.

Since the simulation will now be run offline without as much concern for processing time or memory usage, the low-resolution model of the object used for interactive manipulation can be replaced with a higher resolution version, accurate collision detection can be enabled for the FEM simulations, the number of particles used in the simulation increased, and smaller step sizes taken for better accuracy.

5.5 Chapter Summary

In this chapter I described the operator portion of the language for authoring solid models. The user can apply and modify sculpting and simulation tools without knowing the underlying representation or implementation details, specify the required tool parameters interactively, and describe new simulation behavior. The operator language was demonstrated with a complex example, in which the user effortlessly applied a number of different simulation operators. The language framework for solid model design and manipulation encourages the development of a library of interesting physically-inspired tools.

Now that I have finished presenting the components of the solid modeling language, I will continue in the next three chapters with a description of the implementation: the data structures (Chapter 6), model construction (Chapter 7) and simplification (Chapter 8).

Chapter 6

Tetrahedral Representation

When choosing the representation and support infrastructure for a solid modeling system, several system goals must be considered. To obtain a convincing interactive sculpting system one must capture not just the two-dimensional outer surface of the objects, but also the three-dimensional properties of the materials; for example, a solid three-dimensional texture for wood or marble is much more convincing than a two-dimensional texture "wallpapered" onto a polygonal mesh. Likewise, a surface mesh augmented with bump and displacement maps, while more visually complex, is not sufficient to capture real-world effects such as under-cutting, cracking, and peeling. A volumetric data structure is needed to represent internal structures and the complex layering of different materials.

One of the more important goals for the system is efficiency to allow full user interactivity. This implies that the memory usage of the implementation must scale so the user may manipulate reasonably complex models. The chosen data representation must also work well with a haptic user interface and simulation techniques such as the Finite Element Method (FEM) [Zienkiewicz and Taylor 1989].

In light of these requirements, a tetrahedral mesh was chosen as the most appropriate model representation. In Section 6.1, I discuss the advantages and disadvantages of this design choice. In Section 6.2, I outline the details of my three-dimensional mesh data structure. In Section 6.3, I describe the different operations that are performed on the mesh to implement the tools presented in Chapter 5. Finally, in Section 6.4, I describe some additional system implementation details.

6.1 System Requirements

An important goal of this project is to provide the capability of processing large, interesting models. Motivated by the recent technological advances in volumetric model acquisition, the system is expected to handle high-resolution scanned meshes, which include sharp edges and interesting topology. It became clear early on that a representation with adaptive levels of detail would be necessary to allow higher resolution in some areas and less in others. In particular, I want the capabilities of a volumetric representation without the cost of a dense uniform sampling on the interior of the model, which can far exceed a comparable surface-only representation.

In addition, real-world objects are composed of combinations of materials which have different behaviors as well as different visual appearances. I want to model several materials interacting in a single model and accurately represent the boundaries between different materials without any restrictions on the shape or thickness of the materials. Therefore, the volumetric representation needs to store both the uniform and spatially-varying properties of the material, including the color, shininess, strength, and density.

I want to select a volumetric representation that is amenable to both sculpting operations, specified as Constructive Solid Geometry (CSG) operations, and simulations such as particle systems or FEM. The model representation should support the conservation of volume, if desired by the user. Additionally, collisions and contact forces, which are notoriously expensive to handle in simulations, should be supported by the representation.

Finally, to complete our vision for user-directed simulations, the volumetric representation should facilitate interactive visualization and manipulation of models, as well as interface with advanced user interface technology such as force feedback haptics [SensAble Technologies].

6.1.1 Advantages of Tetrahedral Representation

After considering the goals of the project, I decided that the advantages of a tetrahedral data structure, discussed below, best met the requirements of the target applications.

Triangles are the dominant surface representation in computer graphics applications for a number of reasons, and many of the same advantages also apply for tetrahedra. Tetrahedral meshes are a simple extension of triangle meshes, and their geometric properties, such as simplification

and subdivision, are well understood. In a tetrahedral mesh, there is a simple correlation between volume and surface. The subset of triangular faces that form the exterior surface can be extracted from a tetrahedral representation and rendered interactively on standard graphics hardware. The material boundaries of the model can also be extracted for offline raytracing to produce photorealistic effects such as transparency and refraction. Using normal interpolation, the representation can capture both sharp edges and smoothly curving surfaces. Figure 6.1 shows an object with multiple materials arranged in a complex pattern. Tetrahedral elements can conform to local geometry to represent the shape of exterior and interior boundary surfaces with arbitrary precision. There are no inherent restrictions on the topology of the object or its internal structures, or on the size or thickness of object components.

Like triangular meshes, tetrahedral meshes can be used to represent adaptive detail. Triangular and tetrahedral simplification strategies concentrate elements where the surface curvature is highest and the mesh can be locally refined as desired. Furthermore, by maintaining the corresponding triangle mesh, the complexity of the internal structures is hidden from both graphic and haptic display devices. For example, the GHOST® SDK interface for the PHANTOM™ force feedback, three-dimensional cursor system [SensAble Technologies] requires just a set of triangles and surface properties such as friction and softness. Other volumetric representations, such as voxels or Adaptive Distance Fields [Frisken et al. 2000] would need to first be converted into a triangle mesh using a method like Marching Cubes [Lorensen and Cline 1987]. Alternatively, a custom interface to the haptic device could be developed for these grid-based representations[1].

Another important advantage of a true volumetric representation is the ability to easily perform arbitrary topology changes. With tetrahedral models, voxels and octree representations, an object can be cut into pieces or a hole may be punched through the model. In contrast, the hybrid surface/volume "slab" structure [Dorsey et al. 1999] does not readily support extreme changes in geometry. The slab data structure has obvious advantages for manipulating the geometry within a pre-specified distance from the original surface, but is not useful when a full volume is required.

Many popular simulation techniques, such as FEM, are designed to work on tetrahedral, hexahedral, or other polyhedral element meshes by computing new positions for the element vertices.

[1]In fact, for very soft objects or objects covered with fur or objects with other non-distinct boundaries, a custom haptic interface may provide superior realism to a polygonal mesh.

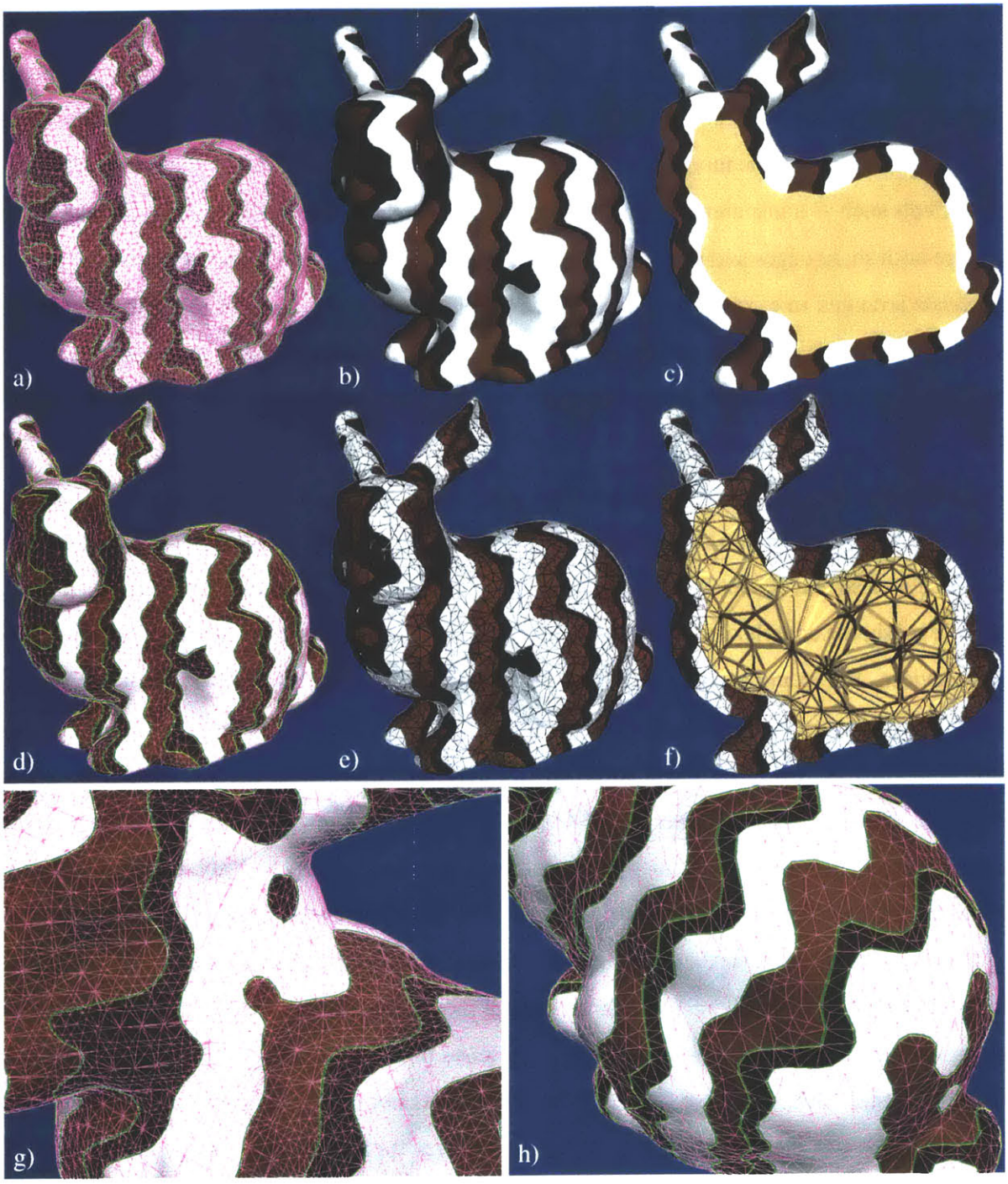
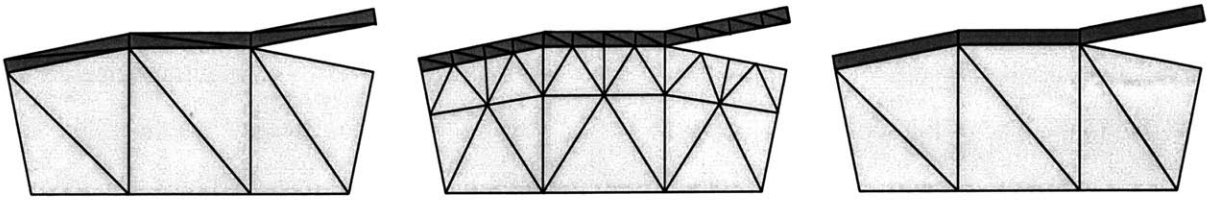


Figure 6.1: Complex arrangements of materials within a volumetric model may be precisely modeled with a tetrahedral mesh. The initial tetrahedralization in a) contains over a million tetrahedra. The thick green lines show boundaries between material types while all other edges are drawn with thin purple lines. The simplified mesh, which contains ~100,000 tetrahedra and ~11,000 triangular faces, is shown in d). Closeups of initial and simplified meshes are shown g) and h). Additional visualizations of the simplified mesh are shown in b), c), e) and f).



a) minimal representation b) subdivision of thin layer c) thick triangles

Figure 6.2: Tetrahedral elements can represent arbitrarily thin materials, but either a) the shape of the elements will be poor, or b) many elements will be necessary. A better solution is to integrate different elements into the mesh such as c) *thick triangles* — triangular prisms that neighbor tetrahedral elements on boundary surfaces.

In contrast, if these computations were performed on regular three-dimensional grids (voxels or ADFs/octrees), the resulting computation would be more expensive and tend to lose accuracy as volume is shifted back and forth across axis-aligned cells. Also, since grid-based data structures do not explicitly store a surface, they are poorly suited to handle operations that fracture the model or require collisions and surface contact forces.

6.1.2 Disadvantages of Tetrahedral Data Structure

There are a number of disadvantages to using a tetrahedral data structure, which needed to be addressed in the system implementation.

It can be difficult to find or create interesting tetrahedral meshes that are appropriate for sculpting and simulation, even though there exists an enormous amount of meshing research and a large number of software packages that solve various types of meshing problems. My contributions in this area are described in Chapters 7 and 8. Tetrahedral meshes are difficult to maintain, because efficient data structures require many pointers that must be consistent across the various operations. Local tetrahedral refinements to add detail can result in poorly-proportioned tetrahedra or an infinite loop if done naively. In contrast, an axis-aligned data structure, with fixed connectivities, such as regular grids or ADFs, is significantly easier to initialize and maintain. Hardware support for storing and processing high resolution grid-based geometries might well make these representations the best choice for many modeling tasks [Pfister et al. 1995].

Although tetrahedral elements can represent objects that are arbitrarily thin, there are disadvantages to constructing such a mesh, illustrated in Figure 6.2. Either the tetrahedral elements will be

poorly proportioned, which can cause problems during simulation, or the model will need to be excessively refined. The refinement approach does not scale well as the material thickness decreases relative to the size of the scene. One of the motivating examples for this project was the faded and curled paint seen in Figure 1.2. It is impractical to model this sort of geometry with tetrahedra *and* require the elements to be well-proportioned. Instead, to handle this situation, I introduce a second volumetric element type to my models, called a *thick triangle*, described in Section 6.2.4.

Tetrahedral elements are not the most appropriate representation for modeling fuzzy or furry objects, soft "blobby" objects, or objects whose density tapers off near the surface. These types of objects have been more successfully represented using point samples and image-based rendering techniques [Wyvill et al. 1986]. Also, tetrahedra are not the best representation for modeling gases, fluids or melting materials. Tetrahedra could be used to represent slowly moving thick fluids such as cooling lava, if an efficient and stable remeshing algorithm is employed online [Ganovelli and C.O'Sullivan 2001]. However, frequent remeshing during a simulation may be impractical if the simulation relies on precomputed factors that are based on the original connectivities.

No single volumetric representation is ideal for all applications. This project is a prototype of a more complete modeling system, which supports several data structures and converts models between representations or constructs a hybrid representation as appropriate for the simulations directed by the user. Ideally, changes in the representation should be transparent to the user.

6.2 Tetrahedral Mesh Representation

In this section I discuss the details of the tetrahedral mesh representation and the challenges that were addressed during implementation. The core mesh data structure maintains lists of vertices, tetrahedra, triangles, thick triangles, edges and materials. These lists are stored in hash tables, using a hash function based on the tuple of vertex indices of each element. The hash table allows (on average) constant time additions, deletions and lookups. A stylized two-dimensional illustration of the pointers between the data structures of the tetrahedral representation is shown in Figure 6.3.

6.2.1 Vertices

The current three-dimensional position of each vertex is stored in the vertex data structure. Each vertex element also stores the original vertex position for consistent solid texturing (see Section 9.3). The vertex stores the original distance field values (computed during model initialization, Chapter 7), which can be used to represent materials whose properties vary throughout the layer based on distance from a given surface or interface. For example, an architectural detail which has been exposed to weathering may be weaker or stained near the surface where rain has penetrated into the stone. Other properties, such as element stress computed during FEM simulation, are also stored at the vertices (see visualizations in Section 9.1).

A new vertex is created when the mesh is subdivided by splitting an edge. The values for the original position, signed distance field and stress for the new vertex are determined by interpolation of the values at the parent vertices. Likewise, if a vertex is moved as part of a remeshing operation, described in Chapter 8, these values are adjusted as appropriate using interpolation.

6.2.2 Tetrahedra

The data structure for each tetrahedron stores pointers to its four vertices and its neighbors across each of its four faces. Neighbors may be tetrahedra, triangles, or thick triangles. Each tetrahedron also stores its material type, volume, and the minimum solid angle of its four vertices. The volume and minimum solid angle of a tetrahedron are computed from the vertex positions, and thus are only stored for computation efficiency. These values are lazily evaluated upon the first request and the result is cached in the tetrahedral data structure for subsequent queries. If any of the four vertices is moved, the cached values are cleared. The original volume, proportion and orientation of each tetrahedron can be computed using the original vertex positions so that simulations can be constrained to minimize element stretch or conserve volume. For simulation, the relevant portions of the FEM matrices are stored in the vertex and tetrahedron data structures [Müller et al. 2001, Müller et al. 2002].

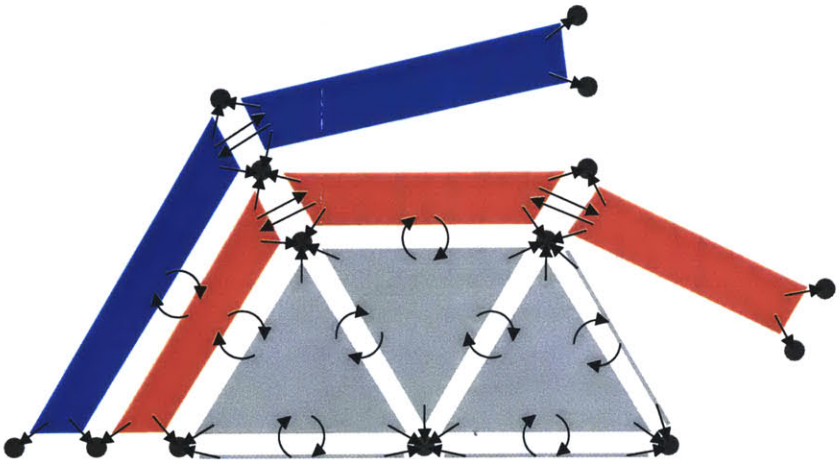


Figure 6.3: A simple example of the pointers between tetrahedra, triangles, thick triangles and vertices is shown in this two-dimensional illustration. Three tetrahedra and three unmatched triangular faces, which appear as triangles and lines respectively, are shown in light gray. Two layers of thick triangles, which are shown as red and blue quadrilaterals, are peeling from the tetrahedral volume.

6.2.3 Triangles

A list of the visible triangle boundary faces (unpaired tetrahedral faces) is stored for efficient rendering and collision detection. Each triangle stores its material type, a pointer to its three vertices, and a pointer to its corresponding tetrahedron. Triangles are simply the visible faces of tetrahedra — they cannot exist without a matching tetrahedral neighbor, and the material of a triangle must match the material of its tetrahedral neighbor. By following face neighbor pointers through tetrahedral elements about a given edge, the triangle-triangle neighbors can be located. As mentioned earlier, the material/material triangle interfaces (all tetrahedral faces whose neighbor has a different material) can also be cached for rendering transparency and refraction.

Setting OpenGL material properties, such as color and shininess, before rendering each triangle is slow [Woo et al. 1999]. For display efficiency, the triangles of each material type are stored in separate lists. Then the material properties can be set once to render all the triangles of a particular material before changing the values for the next material. Chapter 9 contains further options for interactive display and visualization.

6.2.4 Thick Triangles

The adaptive representation strategy of tetrahedral meshes breaks down when representing very thin layers of material relative to the size of the model. Thin layers can be represented, but require either many tetrahedra or poorly shaped tetrahedra, as shown in Figure 6.2. I extended the tetrahedral data structure to include thin triangular prisms, which I term *thick triangles*. These elements have six vertices and may attach to a tetrahedral element on their bottom face. Thick triangles attach to other thick triangles on their side faces, and multiple layers of thick triangles can be stacked by connecting the bottom face of one thick triangle to the top face of the layer below. On a curved surface, the rectangular sides of the prisms are not perfectly parallel or rectangular.

Thick triangles can be used alone to represent volume-less surface shells with finite thickness, such as paper or cloth, and deformed using spring-mass simulations [Baraff and Witkin 1998, Hirota et al. 1998]. Therefore, in addition to vertex indices and material type, each thick triangle also stores spring-mass simulation values such as contraction and curvature. The real advantage of the thick triangle element type is realized when combined with tetrahedral elements to represent models with internal structure *and* very thin surface layers, such as the crumbling plaster and peeling, faded paint shown in Figure 1.2. Some preliminary examples of this data structure are shown in Figure 6.4.

6.2.5 Edges

An optional additional data structure allows information to be stored at the edges of the mesh. This option is used during simplification to cache the cost or *weight* of collapsing a particular edge (see Chapter 8). If a Progressive Mesh technique is used, these edges are stored in a priority queue so that edges may be collapsed in order of increasing cost [Hoppe 1996]. Since the weight of an edge collapse will change as the model is locally modified, the cached values are cleared and recomputed as necessary. Each edge stores a pointer to one of the tetrahedral elements sharing it. Since the edge data structure is optional, edges are only created as needed for particular operations. When the last element sharing an edge is deleted, the edge is also deleted.

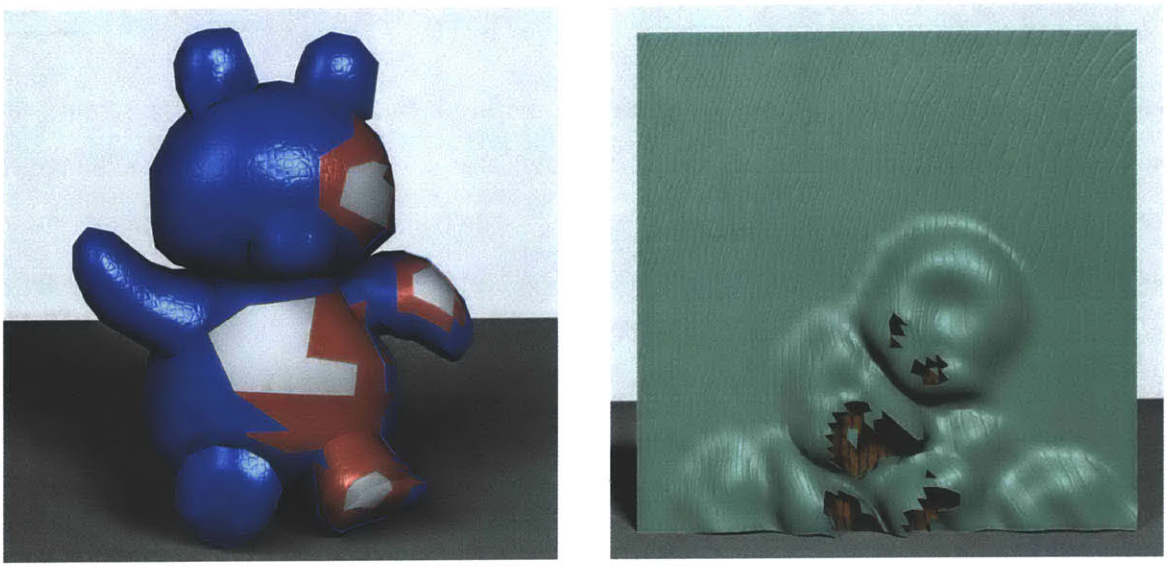


Figure 6.4: These images illustrate a prototype peeling operation that is performed on thick triangles attached to a tetrahedral model. The simulation was developed by Matthias Müller. The left image shows the teddy bear model (from Igarashi et al. [1999]) covered with two layers of peeling paint and the right image shows a chunk of wood covered with paint that has blistered. Both images are rendered with procedural bump maps. These models are low resolution test objects.

6.2.6 Materials

Each tetrahedron, triangle and thick triangle is assigned a material type, which refers to a material stored in the materials list. Each material stores its base color, specular color, shininess, and optional texture rendering information. Material simulation parameters include the density, Young's Modulus, Poisson Ratio and fracture threshold [Anderson 1989], while haptic material parameters include the spring and damping constants and static and dynamic friction [SensAble Technologies].

The material properties of real-world objects change over time. To model these variations, additional information can be stored about the material within each volumetric element.

6.3 Operations on Tetrahedral Meshes

In the next few sections I describe various atomic operations that can be performed on a tetrahedral mesh. These operations are used to implement the simulation module interfaces presented in Chapter 5. The discussion of simplification and mesh improvement (Chapter 8) includes additional operations used for remeshing.

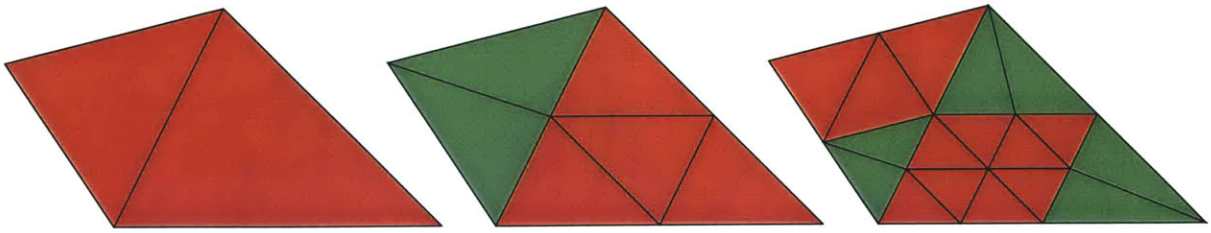


Figure 6.5: The red/green refinement strategy tracks elements that have be regularly subdivided and are congruent to their parent element (red). Elements along the border of areas with different levels of refinement (green) require further subdivision and/or element swaps to match the proportions of their parent element.

6.3.1 Local Mesh Refinement

The initial tetrahedralization of the mesh may need adaptive refinement to model the results of detailed sculpting operations. Also, refinement in areas of high stress can improve the results of an FEM simulation. The basic refinement operation is an edge split — one edge of a tetrahedron is selected and a point is added somewhere along that edge. Each element (tetrahedra, triangles and thick triangles) sharing that edge is split into two new elements. Local refinement must be managed to maintain the overall shape and proportion of the elements from the initial mesh.

There are two refinement strategies for ensuring that elements remain well-proportioned. The first is the *red/green method* (shown in Figure 6.5), which bisects each edge of a triangle to create four congruent triangles that have the same proportions as the original [Bank et al. 1983]. In three dimensions, a tetrahedron is split into four congruent tetrahedra and an octahedron which is split into four more tetrahedra [Bey 1995, Liu and Joe 1996]. To use this decomposition adaptively, triangles on the border of the "red" refined area are labeled "green" since they are only partially refined. Future refinements perform the necessary edge bisections and element "flips" to convert green triangles to red before additional refinement. The second approach is to use only edge bisection, and maintain the proportion of triangles by always splitting the longest edge first, even if that means performing many splits recursively [Adler 1983, Rivara and Hitschfeld 1999], as shown in Figure 6.6. In two dimensions, it is easy to show that longest edge bisection will result in triangles with no angle less than 30 degrees or the smallest input angle. However, in order to make such claims for general tetrahedra, the bisection algorithm must be modified to use projections [Liu and Joe 1995] or label edges [Maubach 1995, Arnold et al. 2000].

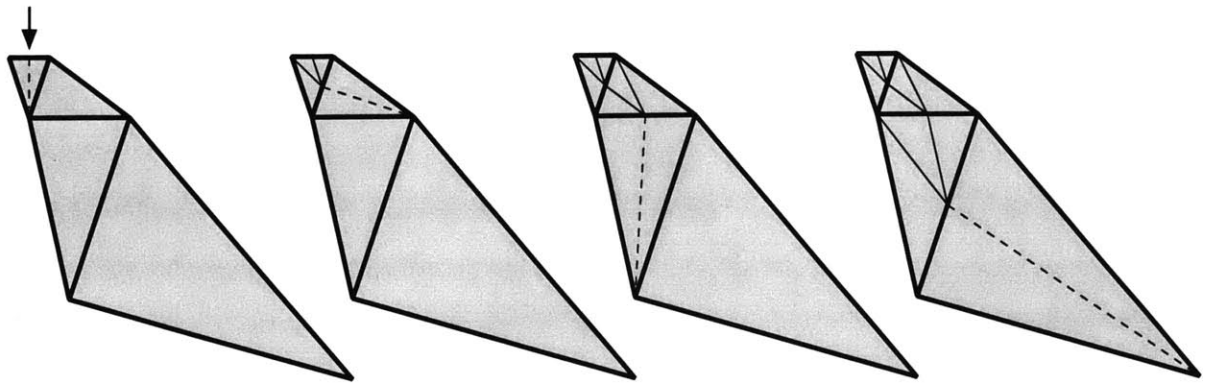


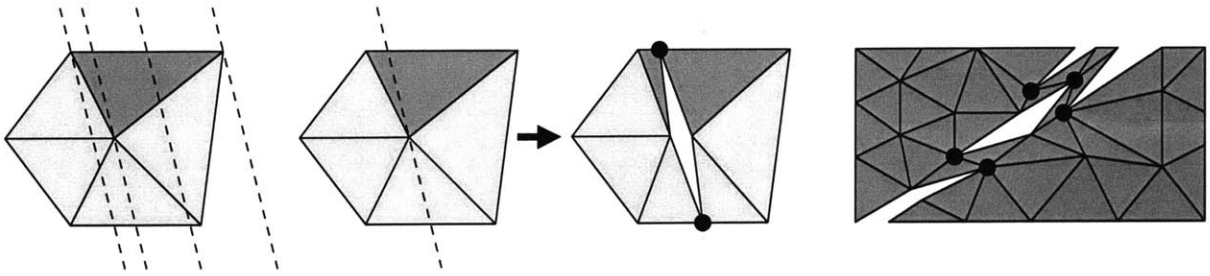
Figure 6.6: A two-dimensional refinement scheme can ensure the continuation of good triangular proportions by always splitting the longest side of the triangle first. In order to split a particular edge (labeled with the arrow), the neighboring triangle sharing that edge is checked. If it has a longer edge, that edge will be split first, after recursing as necessary along the *Longest Edge Propagation Path* [Rivara and Hitschfeld 1999].

Neither regular refinement nor selective bisection alone were acceptable choices for my procedural modeling system, since edges must often be split at a particular interface crossing or tool intersection rather than merely bisected. I implemented a modified version of longest edge bisection that allowed arbitrary edge splits[2], but unfortunately this scheme was not well-behaved (tetrahedral shape continued to worsen with each operation) and often did not even terminate. Currently, I do not attempt to maintain the proportionality of tetrahedral elements during refinement, but perform global simplification and mesh improvement (Chapter 8) before applying any simulations that require well-shaped elements. Developing a refinement algorithm to manage arbitrary edge splits in a well-behaved manner is an area requiring further study.

6.3.2 Fracture

Fracture is an important operation that must be supported in order to perform simulations that break apart objects. An FEM simulation computes the stresses within the model due to externally applied forces (for example, a hammer or gravity), constraints within the model (such as volume preservation or material properties), and interactions with its environment (the ground or other objects). If the stresses in the model exceed the allowable level for the material, the object fractures. The

[2]If a split at point p on edge e is requested, first check all the other edges of all neighboring elements. If there exists a neighboring edge e_2, such that $||e_2|| > 1.5 \cdot ||e||$, then bisect e_2 first (performing this check recursively) before splitting e. Additionally, if p is not in the middle third of e, bisect e before splitting e at p.



a) plane choice b) subdivision to match plane c) difficult to control

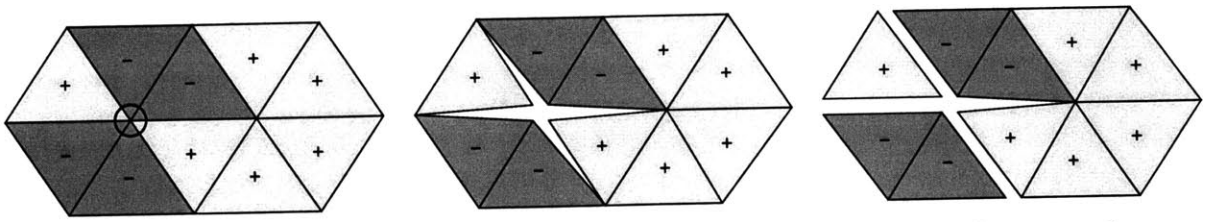
Figure 6.7: The initial fracture interface between the FEM simulation and Geometry Manager identified a) a tetrahedral element and the normal of the ideal fracture plane to release its stress. One of the vertices of that element is selected and neighboring tetrahedra are bisected and separated to b) form a crack along the fracture plane. Poorly shaped tetrahedra may result. This operation is difficult to control because subsequent fractures may form c) separate or parallel cracks. We labeled *crack tip vertices* (black circles) as an attempt to guide fractures through the model, but this approach was not entirely successful.

FEM module (implemented by Matthias Müller) must communicate with the *Geometry Manager* (Section 6.4) to separate the model as necessary.

Initially, we implemented the one-element-per-iteration fracture propagation strategy used by O'Brien and Hodgins [1999]. The disadvantages of this strategy in our real-time simulation environment led me to design a simpler *atomic fracture*. Both methods are described below.

Fracture that Propagates One-Element-Per-Iteration

In this first fracture strategy, the FEM module identifies a tetrahedron exceeding the stress threshold, and the normal for the ideal fracture plane which would release that stress. This limited information is problematic for a number of reasons, the foremost being that the Geometry Manager must choose where within the tetrahedra the model should be separated (Figure 6.7a): At a vertex? Which vertex? Through the middle? We decided that the simulation engine would specify a fracture plane that passes through the vertex with the highest average element stress. A further problem is how to handle tetrahedra split by the fracture plane. Clearly the model should not be limited to break along the original mesh faces, so some refinement of the tetrahedra should be allowed. But tetrahedra with faces almost parallel to the fracture plane probably do not need to be split, as this tends to introduce very poorly proportioned elements (Figure 6.7b). This initial fracture strategy is complicated to implement and places too many decisions and too much work on the



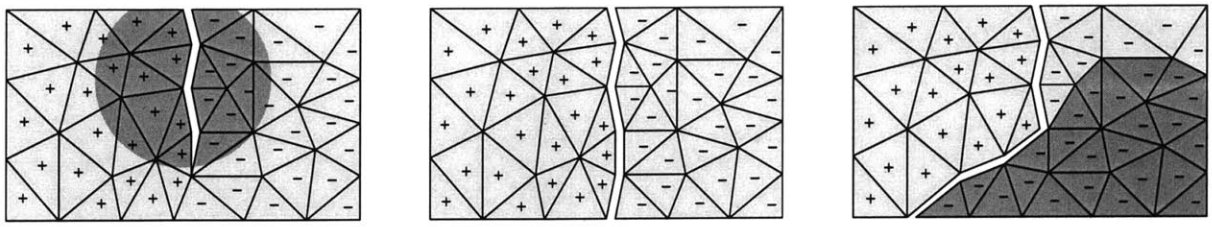
a) labeled b) separate vertex c) consistency update

Figure 6.8: Illustration of the atomic vertex fracture operation. All tetrahedra that may potentially be affected are a) labeled $+/-$ to indicate on which side of the fracture they lie, and then b) `SeparateVertex` is called on the circled vertex. The vertex is duplicated as necessary to separate the faces of neighboring tetrahedra with different signs. As shown in this example, the labeling may split the tetrahedra into more than two groups. The duplicated vertices are placed at the same position in space, but are shown separated in this diagram for clarity. After separating elements at the circled vertex, c) a connectivity check identifies any additional separations that must be performed to ensure than no tetrahedra remain connected by just an edge or vertex (see Section 6.3.4). The model is consistent after this atomic vertex operation, through an additional operation may be performed by calling `SeparateVertex` on the right interior vertex.

Geometry Manager's side. Also, the operation is difficult to control because multiple fracture calls often result in parallel cuts rather than a clean break through the model (see Figure 6.7c). In an attempt to reduce the number of parallel cracks, *crack tip* vertices are identified and labeled. Then future fracture operations are encouraged to occur at these vertices. Unfortunately, this addition is neither physically-based nor particularly successful.

Atomic Fracture for Control of Fracture Propagation Speed

Instead, I chose to implement the simplest atomic operation for fracture, which just separates tetrahedra at a single vertex based on the sign of a splitting function (Figure 6.8). The calling procedure performs any desired tetrahedral refinement first, assigns each neighboring tetrahedron a *side* $(+/-)$ and then calls `SeparateVertex` on each vertex, as desired. More complex fracture operations that use edge splits to refine the mesh can be built from this simple operation. In particular, with this operation one can directly control the speed of fracture propagation allowing implementation of real-time rigid body fracturing [Müller et al. 2001]. All tetrahedra within the *tool radius* are assigned a side, and then all vertices within that radius are fractured (see Figure 6.9). This allows a fracture to propagate across an unlimited number of elements, in contrast to the (slower, but more physically correct) one-element-per-iteration scheme by O'Brien and Hodgins



a) local fracture b) global effect c) unbreakable material

Figure 6.9: More complex operations can be built from the atomic fracture operation shown in Figure 6.8. We can call `SeparateVertex` on all vertices, within a specified radius, to create a) local fractures which propagate partially into the model. Executing this operation on all the vertices causes the fracture to b) slice through the entire object. We can model c) *unbreakable* materials (dark gray) by giving all tetrahedra of that material the same sign. Then fractures will propagate along the material boundaries instead.

[1999]. Additionally, unbreakable materials can be modeled by simply assigning all elements of that material type to the same side of the fracture.

6.3.3 Constructive Solid Geometry (CSG)

Another operation supported in my system is the removal of material by standard set operations such as CSG subtraction, shown in Figure 6.10. Tetrahedra that are completely inside the tool volume are removed and tetrahedra that are completely outside the tool volume are ignored. Tetrahedra that lie on the border are split, then checked again.

The simplest refinement scheme for CSG operations simply bisects the longest edge of each boundary tetrahedron until the volume is less than epsilon. This approach results in a very jagged appearance for the surface after the sculpting operation because the normals of the faces do not conform to the local shape of the tool. Also, the resulting mesh is likely oversampled due to the excessive subdivision. A much better technique subdivides each boundary tetrahedron along its longest edge that has one vertex inside the tool boundary and one vertex outside the tool boundary, at the crossing point. Although more expensive to compute, this subdivision method produces a smooth post-sculpt surface appearance and fewer total subdivisions than the first method.

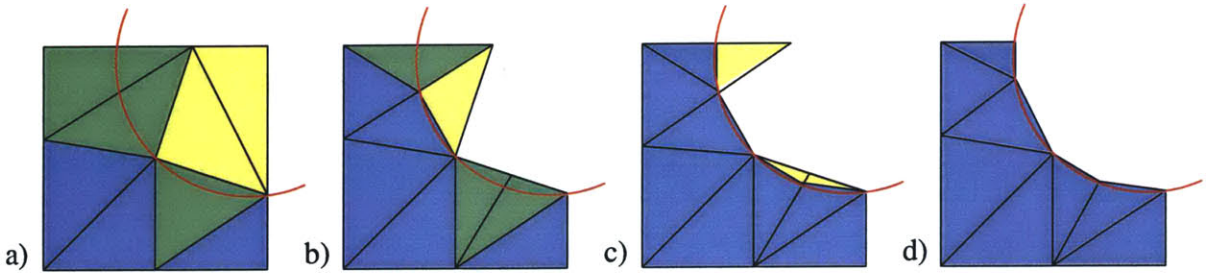


Figure 6.10: A sequence from a CSG sphere subtraction action is illustrated in two dimensions. All tetrahedra completely inside the tool (yellow) are removed. All tetrahedra completely outside the tool (blue) are ignored. The remaining tetrahedra (green) are split along the edges that cross the tool boundary. If both endpoints are on the boundary or outside the tool, the edge is bisected. The process terminates when the closest point on each edge or face is outside or within epsilon of the tool boundary.

6.3.4 Mesh Connectivity and Consistency

To allow efficient computation of the exterior interface for rendering and linear or constant running time for various mesh operations, the inter-element connectivities, shown in Figure 6.3, must be maintained. This connectivity information does not need to be stored when the mesh is saved to a file. The system can compute the face/face connectivity information from "polygon soup" or in this case "tetrahedral soup". In fact, the volumetric model file format only stores materials, vertices, tetrahedra and thick triangles. Triangles and edges are recreated when the model is loaded. As each tetrahedron or thick triangle is loaded, its faces are added to a hash table, hashed by the tuple of vertex indices, with smallest index first. If a matching face is found (flipped traversal of vertices), the two faces are removed from the hash table and linked with pointers. When all of the tetrahedra and thick triangles are loaded, any faces remaining in the hash table are unpaired, triggering the creation of triangles to form the air/material boundary.

To debug the implementation of the various mesh operations described in the previous section, an exhaustive `CheckMesh` procedure can be performed between atomic actions to verify the following mesh properties:

- All elements claimed by one element as neighbors should point back, and tetrahedral neighbors that point to each other must share exactly three vertices. For example, a naive edge collapse algorithm may create an illegal situation where a pair of tetrahedra also share the fourth vertex.

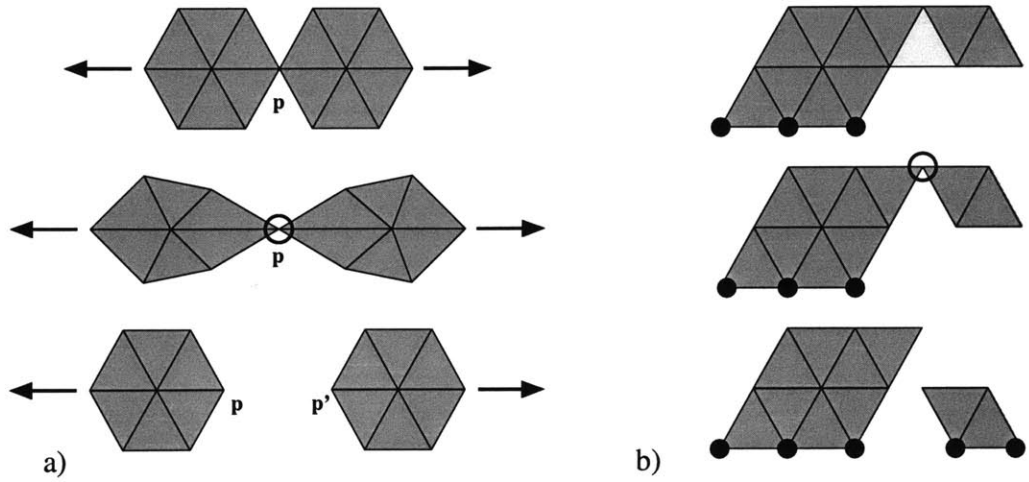


Figure 6.11: The model shown in a) has two pieces which are connected by only one vertex, p. This model is non-manifold and disallowed in the system since no element connections exist between the two pieces. If a tensile force is applied to pull the pieces apart, the model will be unrealistically stretched. In the correct representation with a duplicate vertex, p', the two pieces separate appropriately. The model shown in b) illustrates how a simple check before a tetrahedron is removed from the model (shown in light gray) will reveal that the model will be connected by only one vertex, so that it may be properly duplicated. Once disconnected, the separate piece can be controlled by the rigid body simulation to fall to the ground and establish new contact points (black circles).

- There should be no unmatched faces in the face hash table.

- Tetrahedral volumes are also checked, but the system is tolerant to negative volumes as they may be present in input models, or at intermediate stages of a simulation which will eventually re-stabilize, or just be the result of rounding errors in very small volume elements. Negative volumes are discouraged as they violate assumptions for rendering and collision detection, but I was careful not to rely on the volume sign for any of the infrastructure procedures.

Additional mesh consistency checks are discussed in Section 8.3.5.

Each time a tetrahedron is removed from the system, its vertices are analyzed. If removal of this tetrahedron will cause two or more tetrahedra to be connected by only a vertex or an edge, these tetrahedra must be disconnected by creating one or more duplicate vertices (Figure 6.11). Since the tetrahedral data structure maintains links between tetrahedra and from tetrahedra to vertices, but not from vertices to tetrahedra, detecting such a disconnection after-the-fact is very costly (quadratic in the number of tetrahedra). However, by inspecting the neighborhood of each tetrahedron just

before it is removed, these disconnections can be efficiently performed. Also, the system must recognize if this disconnection separates the model into two or more rigid bodies. For a simple environment with gravity and a ground plane, every resting tetrahedron should be connected to a vertex with a contact force.

6.4 Additional System Implementation Details

Previous work with three-dimensional painting and sculpting programs convinced me of the importance of a responsive system. It is not acceptable to freeze the display window while the system performs a computationally intensive operation. The user should not become so frustrated that he repeatedly clicks mouse buttons or the keyboard, unknowingly queuing up a long list of future actions, which will further delay the much needed screen refresh. To ensure adequate responsiveness, the interactive program consists of two threads, the *Geometry Manager* and the *Device Manager*, which communicate through shared memory (Figure 6.12).

The Geometry Manager receives commands from the *command queue* to perform certain operations. The commands are passed to the appropriate module, such as CSG sculpting, FEM, particle system, etc. These modules modify the Geometry Manager's copy of the object. Periodically, when the mesh is in a consistent state, a copy of the model is exported to a shared memory location that can be accessed by the other thread. If a module is performing a complex operation, the geometry may not be updated for an extended period of time, but the Device Manager still has access to an older copy of the geometry.

The Device Manager thread runs a tight event loop which checks for mouse, keyboard, haptics and window manager events. The geometry is redrawn when the camera is moved, the cursor is moved and tool display is enabled, or the window is uncovered. Each type of geometry (vertices, tetrahedra, triangles, and thick triangles) is rendered to a separate OpenGL display list [Woo et al. 1999], so that it can be efficiently displayed when the appropriate GUI toggle button is selected. When new geometry becomes available or the display options for a type of geometry are changed, the display lists are regenerated as necessary. Display lists for element types that are not currently selected are not updated.

In the implementation, the haptics update loop is actually parceled off to a third thread to ensure

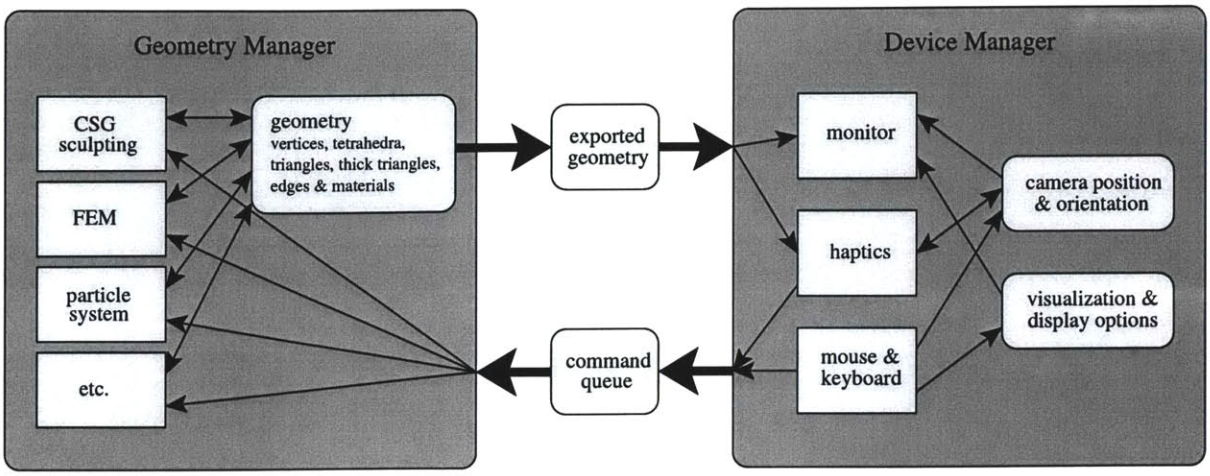


Figure 6.12: The interactive system is logically divided into two threads which communicate through shared memory.

the required 1,000 Hz haptic refresh rate. Additionally, geometry modules such as FEM may be parallelized by adding additional computation threads.

6.5 Chapter Summary

In this chapter I explained the rationale behind the decision to use tetrahedral meshes in the prototype implementation. In the description of the details of the representation and the operations applied to the model, I enumerated some of the added difficulties of working with a volumetric model rather than a surface mesh. Overall, once these challenges were tackled, the implementation was successfully used to produce convincing results both interactively and offline on complex high-resolution models. In the next chapter I will describe methods for initializing interesting tetrahedral meshes.

Chapter 7

Creating Tetrahedral Models

To represent interesting solid objects with tetrahedral meshes, I developed a pipeline for generating these models from the scripting language presented in Chapter 4. I used a structured meshing technique, which is the volumetric extension of the Marching Cubes algorithm [Lorensen and Cline 1987], and Implicit Surface Polygonization [Bloomenthal 1994]. I chose this method of tetrahedralization because it was simple to implement, allows much flexibility, and its drawbacks, compared with other meshing strategies, are not as significant for this application.

Other meshing strategies require that all surface boundaries be explicitly defined and provided as input. In order to implement layers, one of the main features of the solid modeling language, the appropriate offset surfaces (isosurfaces) of the input model must be computed. The most robust method for computing the isosurfaces of general surface meshes is to use a *signed distance field* (Section 7.1). By using a structured meshing technique, I bypass the explicit construction of these offset surfaces and simply generate volumetric elements from the signed distance field. In this chapter, I will describe my extensions to the basic octree tetrahedral mesh generation technique. In Section 7.2, I explain how to initialize distance fields from a surface mesh, even if it contains minor inconsistencies. Distance field manipulation operations are listed in Section 7.3. Finally, in Section 7.4, I describe how a distance field is tetrahedralized.

7.1 Signed Distance Field

The signed distance field is a robust method for computing isosurfaces from general surface meshes. It seamlessly handles changes in topology and prevents overlapping or intersection of

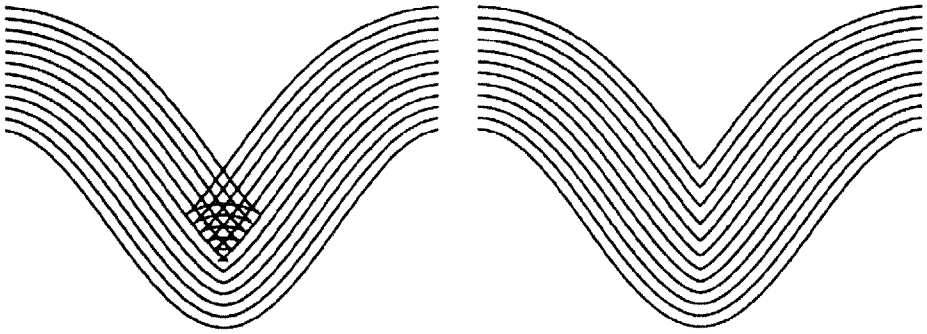
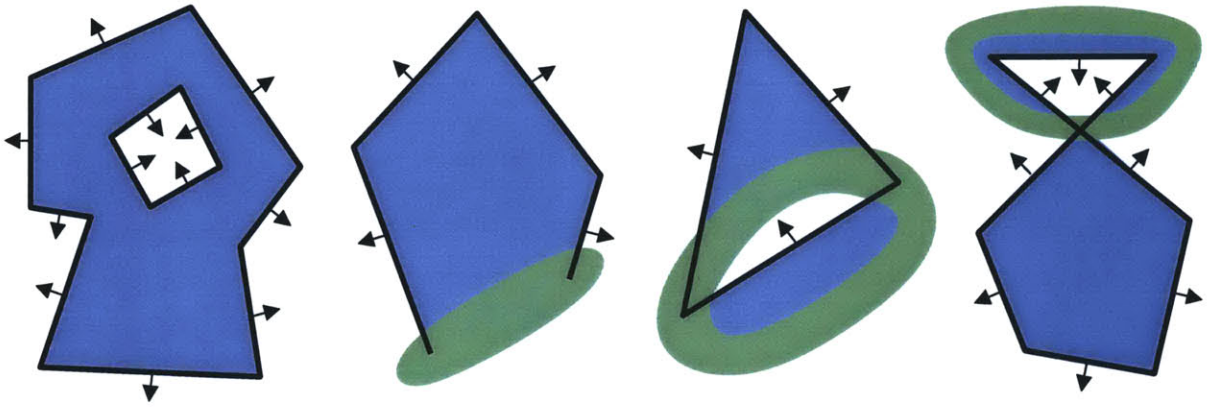


Figure 7.1: The *swallowtail* shape of the left image is the result of tracking offset surfaces from the bottom curve using a traditional *marker and string* approach. Each marker is moved a fixed distance in the direction of the normal. The level set solution on the right correctly tracks the offset surface without self intersection and also allows changes in topology. (Illustration from Sethian [1999].)

the offset surfaces. In contrast, a simple projection technique can only be used for certain classes of meshes, such as *star-shaped surfaces*. Figure 7.1 illustrates a common problem with these naive projection methods.

The tetrahedralization process begins by evaluating the signed distance field on a uniform three-dimensional grid. The signed distance field is a continuous[1] function defined over a volume. Each point in the grid is labeled with the distance to the closest point on the surface and whether that point is inside or outside the surface. In order to consistently label the correct side for each point, the surface must be *watertight* — that is, closed, orientable, free from self-intersections, and manifold. Beyond that, the topology of the surface is not limited: it may contain holes or be composed of disjoint pieces. For example, if the surface is a triangular mesh, each edge should be shared by exactly two triangles, the triangles should be consistently labeled with normals, and the surface should contain no self-intersections. If these properties are not met, the sidedness of the surface is not well-defined, as illustrated in Figure 7.2. In Section 7.2, I describe how to detect and repair minor self-intersections in the surface.

[1]The distance field need only be C[0] continuous; in other words, the derivative of the field does not need to be continuous.



a) topology b) non-closed c) conflicting normals d) self-intersection

Figure 7.2: A closed polygonal manifold uniquely designates each point in space as either inside or outside, as shown in a). If the mesh is not closed, that is, some elements are missing neighbors, as in b), then the space around these missing faces is undefined. Additionally, if the direction of the normal is not consistent across the faces, shown in c), or if the surface self-intersects, as in d), then the sidedness is not well-defined.

7.1.1 Grid

The first step in constructing a signed distance field is selecting an appropriate spacing for the uniform grid. If the grid is too coarse, the model may be under-sampled and lose important details. If the grid is too fine, extra time must be spent initializing and rendering the data, and the system may run out of memory. Changing the grid alignment and sample placement can also change the resulting tetrahedralization. Sometimes a carefully selected, lower resolution grid results in a more accurate output model, as shown in Figure 7.3. A grid size should be selected to sample the object satisfactorily, regardless of placement. Stander and Hart [1997] show how to determine an appropriate sampling of an implicit surface to guarantee correct topology.

Usually the grid spacing is supplied as a modeling parameter in the script file; however, if unspecified, the system will choose a grid spacing that will produce roughly 5,000 tetrahedra, using the bounding box of the input model as a guide.

$$\text{default grid} = \sqrt[3]{\frac{5 * \text{height} * \text{width} * \text{depth}}{5,000}}$$

Each cubic cell (or voxel) of the grid will be subdivided into five tetrahedra (Section 7.4.1); thus, the algorithm selects a grid spacing that results in roughly 1,000 cubic grid cells. Of course, depending on the general shape of the model, some of the grid cells in the bounding box may

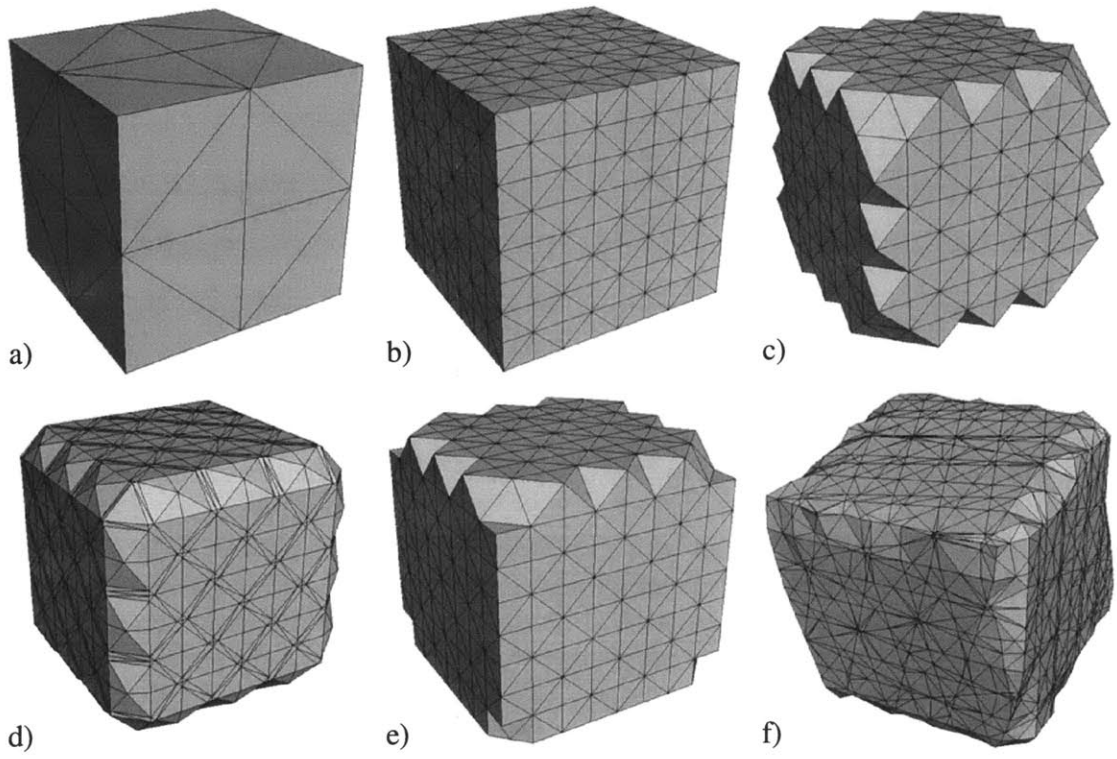


Figure 7.3: Grid spacing and alignment can have a large impact on resulting tetrahedralization. Here I show a simple axis-aligned cube spanning (-1,-1,-1) ⇒ (1,1,1) meshed with grids a) 1.0, b) 0.25, c) 0.25001, and d) 0.26. If the object is e) translated vertically by 0.001 or f) rotated so it is no longer axis aligned, the resulting tetrahedralization and artifacts will be different. In Section 7.4.5, I explain the cause of these artifacts and outline a partial solution.

never be initialized. Alternatively, the grid size could be chosen based on the shortest edge length in the mesh to provide some assurance that the details in the original mesh are captured. However, if the original mesh is over-sampled or has zero-length edges this method is unsatisfactory as it provides no bound to the number of elements produced.

The signed distance field grid points are stored in a hash table indexed by location to allow constant time (on average) additions, deletions and lookups. Location is specified as a triple of integer multiples of the grid size, which is a floating point value. Each point can store the values for several different distance fields, which are referenced by a distance field index value. Not all distance fields must be evaluated at every point. For each distance field, the point stores the status (*KNOWN*, *TRIAL* or *FAR AWAY*), the magnitude and sign of the distance (interior or exterior), and the isosurface velocity (spacing between isosurfaces of the distance field). The velocity equals 1.0

for Euclidean distance fields. The status and velocity are used when propagating the field and will be explained in Sections 7.2.3 and 7.3.2.

7.1.2 Implicit Surfaces

To initialize the distance field from an implicit surface, the user provides a shape function $f(x, y, z)$ and a bounding region. Then, the function is evaluated at each grid point within the bounding region. If the function is not continuous, interpolation will less accurately predict the field between samples, and the output shape might change dramatically with a different grid size or alignment. However, the grid tetrahedralization method described in Section 7.4 will robustly process any assignment of distance values; thus, the function does not need to be continuous, it just needs to be defined within the bounding region.

The bounding region can be any shape, though usually it is just a rectilinear box. If the user can accurately identify which areas of space the input and the output models will occupy, the field will only need to be evaluated in these areas, saving time and storage space. In fact, the bounding region need not specify a *closed* surface or offset surface. For example, a cylinder of height h can be created from the shape function for an infinite cylinder by specifying a bounding region of height h.

7.2 Signed Distance Field of a Triangular Mesh

Computing the signed distance field from a triangle mesh can be viewed as a special built-in implicit surface function. However, for efficiency and robustness, I have implemented the evaluation of this function as a forward-mapping algorithm (each triangle initializes neighboring points) rather than as a backward-mapping algorithm (each point queries neighboring triangles). In the sections below, I describe the requirements placed on input meshes, how to initialize a band of distance values near the surface, and how to propagate these values to fill out the distance field, as necessary.

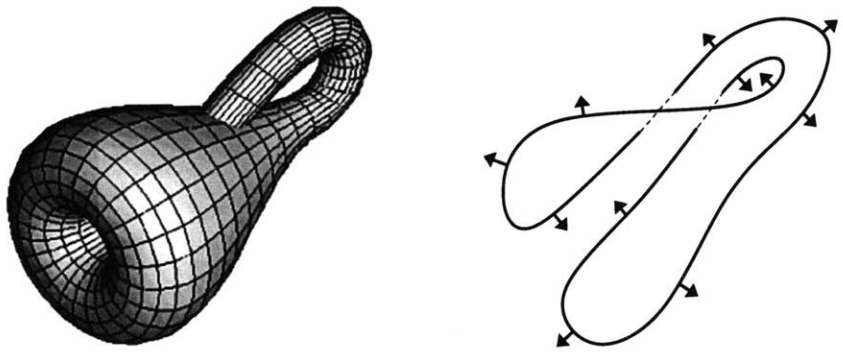


Figure 7.4: The Klein Bottle is a classic example of a non-orientable surface. In the cut-through illustration on the right, a partial labeling of orientation shows inconsistencies (dashed lines) at the intersection joint. The orientations could be selectively flipped to produce a consistent two-dimensional slice, but the three-dimensional model would remain inconsistent.

7.2.1 Requirements on Input Meshes

Figure 7.2 illustrates some of the problems that exist in polygonal surface representations of objects. For the implementation, I assumed that the mesh is closed and orientable — these two properties are simple to verify. Each triangle should have exactly three neighbors, one across each edge, and triangles that share an edge should have consistently oriented normals. Even if the mesh is presented as "triangle soup", neighbor connectivity can be efficiently computed by hashing the triangle edges. It is not unreasonable to expect input polygonal meshes to meet these criteria. Not only can these criteria be efficiently verified, problems are often simple to correct.

A *hole* in the mesh, a ring of n unpaired edges, can be filled by adding n-2 triangles. Cyberware's 3D Color Scanner [Cyberware] is shipped with software for automatic hole-filling that optionally allows the user to adjust the boundaries of the largest holes for a better overall mesh. Automatic hole-filling may introduce intersections, webbing or volumetric holes in the object, but these situations do not pose difficulties for my distance field initialization implementation. More sophisticated hole-filling techniques can be used to reduce the chance of undesirable shape and topology artifacts [Norruddin and Turk 1999, Nooruddin and Turk 2000, Davis et al. 2002].

Small inconsistencies in orientation can be resolved by flipping individual triangles in a systematic way or by disconnecting edges between triangles with inconsistently-labeled normals, duplicating vertices along these edges, and then filling the newly created hole(s). More significant orientation inconsistencies, such as the Klein Bottle, shown in Figure 7.4, are irreparable and

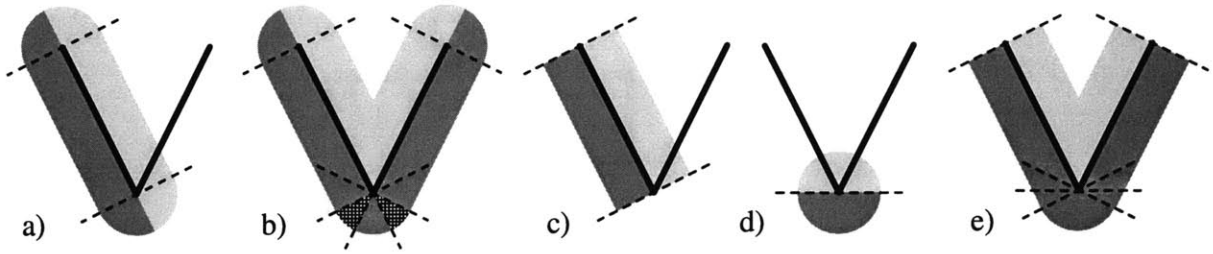


Figure 7.5: Care must be taken to correctly assign the sidedness of points near acute mesh edges and vertices. The naive approach a) assigns sidedness based only on the normal of each face. Unfortunately, for acute angles, the faces b) agree about the magnitude, but disagree about the sign for points in the two hatched wedges. Instead, the normal of the plane is used c) only for points that project within the boundary of the face. For points that project to vertices or edges, the d) weighted average of the normals of all neighboring faces is used instead, resulting in e) a consistent spatial labeling.

meshes that contain them cannot be used in this system.

I do not require input meshes to be free from self-intersections. This property is expensive to check and is subject to floating point rounding errors. In fact, most high-resolution scanned meshes do contain tiny intersections, even though a visual inspection may not reveal the problems. Removing intersections is not always straightforward, and it is one of the most difficult problems to solve when implementing a cloth simulation engine [Bridson et al. 2002, Baraff et al. 2003]. Even if the intersections in a high-resolution mesh are removed, surface simplification may introduce new intersections unless this property is explicitly maintained. In contrast, most off-the-shelf triangle simplification packages, such as QSlim [Garland], do maintain the connectivity and orientation of the object.

7.2.2 Initializing a Band of Distance Values

Given surface S, the signed distance function f_S is defined as follows: for any point p in $\mathbf{R}^3$, the magnitude of $f_S(p)$ is the distance from p to the closest point on S, and the sign of $f_S(p)$ is negative if p lies in the interior volume of S and positive if it lies outside. I initialize a band of points within $2\sqrt{3}$ grid units of the original surface by iterating over the faces in the mesh and *rasterizing* each face F into the volume grid. If I were instead to iterate over all points in the grid, the surface mesh would need to be stored in a spatial data structure for efficiency, and time would be wasted computing distance values for points that are more than $2\sqrt{3}$ grid units from the surface.

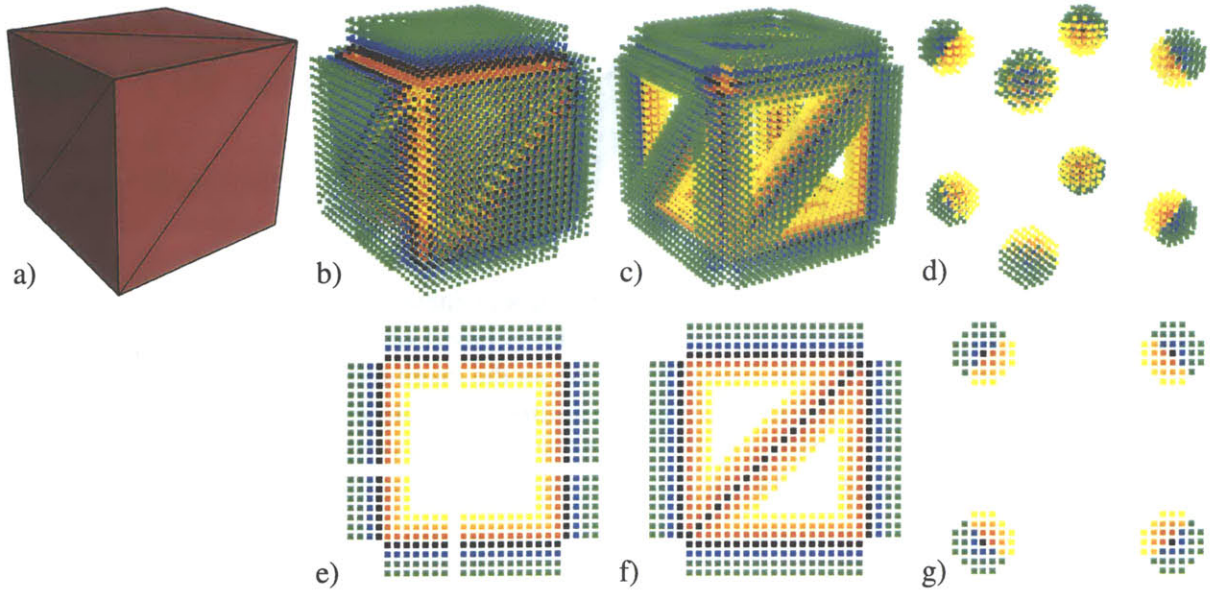


Figure 7.6: To rasterize the cube mesh shown in a), I combine the distance fields for the b) faces, c) edges and d) vertices of the mesh. To correctly label the sign of each point, normals for each edge and vertex are computed as a weighted average of their neighboring faces. In this visualization, interior points are colored red and fade to yellow as they move away from the surface, while exterior points are colored blue and fade to green. Points within epsilon of the surface are colored black. Also shown is a cut through the middle of the cube for the e) face distances and cuts in the plane of a face for the f) edge distances and g) vertex distances.

For all grid points p near F, I compute $f_F(p)$, the signed distance from point p to F. If $|f_F(p)| \leq 2\sqrt{3}$, and $|f_F(p)| < |f_S(p)|$ or $f_S(p)$ is undefined, I *update* the value of the distance field at p, by setting $f_S(p) = f_F(p)$. To compute $f_F(p)$, I find point p' on F closest to p. Then, $|f_F(p)| = \|p - p'\|$ and the sign of $f_F(p)$ is obtained as the sign of $(p' - p) \cdot n$, where n is the surface normal at p'. To make this scheme robust, if p' lies on a vertex or edge of F, the normal n is obtained by a weighted average of the normals of adjacent faces (see Figures 7.5 and 7.6).

7.2.3 Propagating the Signed Distance Field

After all faces have been rasterized, the function f_S is defined in the $2\sqrt{3}$-proximity region of S. In the next step, I propagate the distance field on both the interior and exterior using the Fast Marching Method [Sethian 1999], so that the range of distance values for each layer is defined. To summarize the Fast Marching Method:

1. Mark all points with valid distance values *KNOWN* and all undefined or uninstantiated points

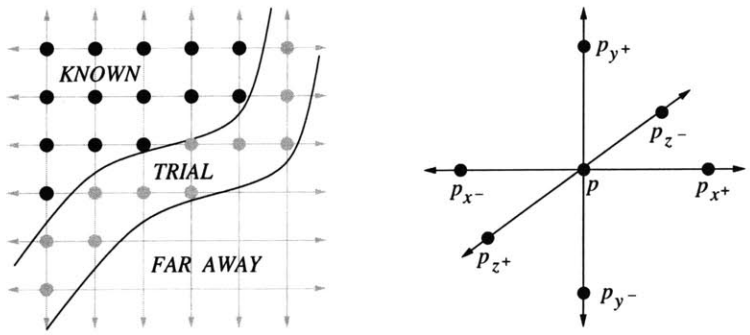


Figure 7.7: The image on the left, redrawn from Sethian [1999], illustrates the Fast Marching Method. The updated distance field value at point p is computed from three neighboring points, one per axis: p_x, p_y, and p_z, with the smallest absolute distance values.

FAR AWAY.

2. Place all *KNOWN* points in a priority queue or heap ordered by the magnitude of their distance value. The point with the smallest distance value is on top of the heap.

3. Remove the point with the smallest distance value from the top of the heap. Mark it *KNOWN*. *Update* the values of the six neighboring points ($\pm$ 1 grid unit along each axis) that are marked *TRIAL* or *FAR AWAY*.

4. Stop when there are no points left in the heap.

To update the value of point p:

1. Check the 26 neighboring points (including diagonals). If the distance values for this point and all neighbors are either undefined (marked *FAR AWAY*) or outside the range required by the layers of the volume, do nothing.

2. Otherwise, for each axis, collect the point $\pm$ 1 grid unit away with the smallest distance field value (see Figure 7.7). Call these points p_x, p_y, and p_z.

$$\text{if } |f(p_{x+})| < |f(p_{x-})| \text{ then } p_x = p_{x+} \text{ else } p_x = p_{x-}$$
$$\text{if } |f(p_{y+})| < |f(p_{y-})| \text{ then } p_y = p_{y+} \text{ else } p_y = p_{y-}$$
$$\text{if } |f(p_{z+})| < |f(p_{z-})| \text{ then } p_z = p_{z+} \text{ else } p_z = p_{z-}$$

If neither point is defined (marked *TRIAL* or *KNOWN*) for a particular axis, do not collect a point for that axis.

3. Solve the following equation for $f(p)$, the updated distance value for point p:

$$\sqrt{\begin{array}{l}\text{maximum}(0, |f(p)| - |f(p_x)|)^2 + \\ \text{maximum}(0, |f(p)| - |f(p_y)|)^2 + \\ \text{maximum}(0, |f(p)| - |f(p_z)|)^2\end{array}} = \frac{\text{grid}}{\text{velocity}}$$

For Euclidean distance fields, velocity equals 1.0. If p_x is *FAR AWAY*, $f(p_x)$ equals infinity, and the x-term drops from the equation; likewise for y and z. "maximum" is removed from this equation by dropping different combinations of terms from the master equation and solving the resulting quadratic equations. The final answer is the value for $f(p)$ with the smallest magnitude.

4. The sign of $f(p)$ is the same as the signs of $f(p_x)$, $f(p_y)$, and $f(p_z)$[2].

Propagating the signed distance field using the Level Sets Fast Marching Method is much faster. The speed is linear in n_p, the number of points initialized, whereas direct instantiation can be as slow as $O(n_p \cdot n_t)$, where n_t is the number of triangles. Spatial data structures for the triangle mesh lower this bound substantially, but the direct approach will initialize more points than the Fast Marching approach, which only initializes the points that are needed to tetrahedralize the volume. The main advantage of the Level Sets technique is that it allows variations in the isosurface velocity, which I describe in Section 7.3.

7.2.4 Resolving Inconsistencies in the Initialized Distance Field

Unfortunately, even after the careful rasterization of each triangle in the mesh, described in Section 7.2.2, the initialized distance field may contain inconsistencies that will lead to unacceptable errors in the tetrahedralization. For example, if we try to fill the interior of a mesh whose distance field contains a point labeled *interior* that should have been labeled *exterior*, the resulting error can propagate into a larger bubble of incorrectly signed points. The incorrect points become erroneous extra volume floating around the object or lack of volume on the interior of the object when the

[2]Because all points within $2\sqrt{3}$ units of the surface were initialized and set to *KNOWN*, and the update routine is only performed on *TRIAL* and *FAR AWAY* points, the signs of $f(p_x)$, $f(p_y)$, and $f(p_z)$ must match. This property is guaranteed by the cleanup algorithm described in the next section.

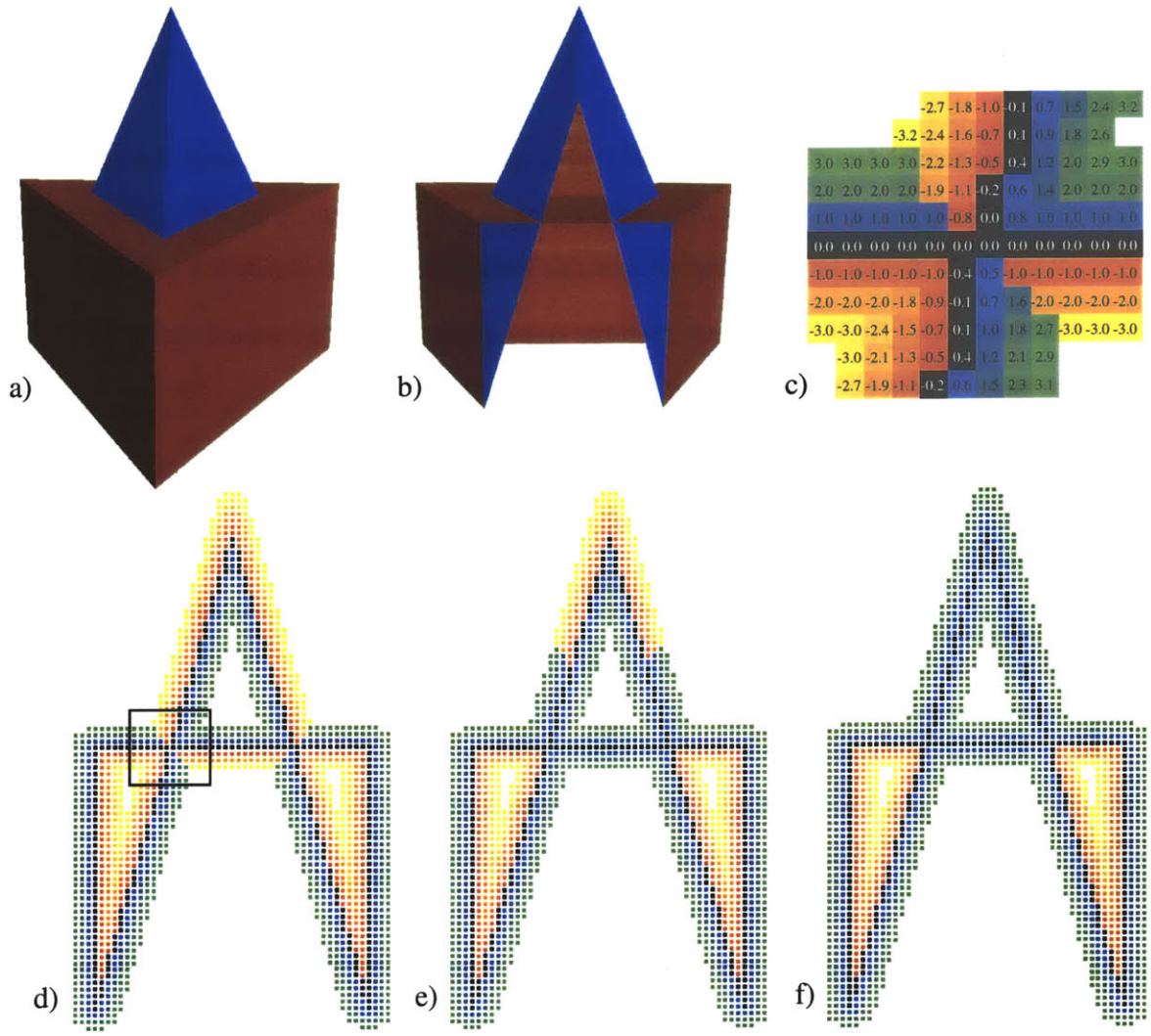


Figure 7.8: A self-intersection has been introduced to the triangular prism shown in a) and with a clipping plane in b). An additional vertex in the middle of the bottom face has been translated upward and crosses the top face. The initial distance field visualized in d) contains inconsistencies, shown in close-up in c), which are iteratively resolved as illustrated in e) an intermediate stage and f) the final result.

model is tetrahedralized. Errors in sign can arise from floating point rounding errors and intersections or coincident triangles in the original mesh. To remove these inconsistencies, I developed a simple cleanup procedure to be used after distance field initialization but before value propagation (Section 7.2.3). The algorithm, illustrated in Figure 7.8, repeatedly loops over all points in the distance field and selectively flips the sign of each point as follows:

1. Examine the 26 neighbors (including diagonals) of point p. For each neighboring point n,

compare the difference in distance value $|f(p) - f(n)|$ with the Euclidean distance, $\|p - n\|$, between the points. Collect two statistics: `flip` and `stay`.

- An inconsistency is detected if $|f(p) - f(n)| > \|p - n\|$, because this incorrectly suggests that the velocity between p and n is > 1.0. Either the sign of $f(p)$ or the sign of $f(n)$ should be flipped to remove this inconsistency[3], so increase `flip` by 1.
- If $|f(p) + f(n)| > \|p - n\|$, then the signs of $f(p)$ and $f(n)$ should not be flipped (or should both be flipped), so increase `stay` by 1.

2. If p is an interior point and `flip` $\geq 0.5 *$ `stay` or if p is an exterior point and `flip` $\geq 2 *$ `stay` then flip the sign of $f(p)$.

3. Stop after a loop over all points in which none were flipped. If the algorithm finishes successfully, `flip` equals zero for each point.

The band of known grid vertices was initialized within $2\sqrt{3}$ units of the surface (Section 7.2.2) to ensure that each point near the surface has adequate samples for this cleanup procedure. Once the field is consistently labeled, no point will have both a neighbor on the opposite side of the surface and an undefined (*FAR AWAY*) neighbor.

The cleanup algorithm could easily get stuck in an infinite loop if the likelihood of interior $\rightarrow$ exterior and exterior $\rightarrow$ interior flips were equal. As outlined above, the algorithm favors interior $\rightarrow$ exterior flips. In the example shown in Figure 7.8, this favoring means that the inverted portion of the intersection is effectively erased. If layers are grown only on the interior of the corrected distance field, the model will not contain any volume for the top triangular wedge of material. I believe this is the best compromise solution.

Worst case initial distance fields could probably be constructed for which this cleanup algorithm enters an infinite loop. This situation can be detected by specifying a maximum number of iterations based on the total number of distance field points, or by tracking the number of times the sign of each point is flipped. Additionally, the algorithm does not address the problem of an "inside-out" initial mesh. Using the Fast Marching Method to propagate distance values and *fill*

[3]Because the field was initialized with a Euclidean distance metric, once the sign of p or n is flipped, then it must be true that $|f(p) - f(n)| \leq \|p - n\|$.

the "interior" of the distance field of such an object will not terminate. This problem can be detected by visual inspection or by casting rays from the external bounding box. If a ray through a consistently-signed distance field first intersects a point with a negative distance value, the field must be inside-out.

7.3 Operations on Distance Fields

Once a distance field has been initialized from either an implicit function or polygonal mesh, mathematical operations can be performed on it. Two fields can be combined and stored in a new field, or a particular isosurface of a field can be used to initialize another field with a different isosurface velocity.

7.3.1 Combining Distance Fields

To implement a union, intersection, or subtraction of distance fields, the two distance fields are evaluated separately throughout the bounding region and then performs the minimum, maximum or subtraction, per point, as appropriate, and stores the result in a combined distance field. The bounding region for the third field is necessarily the intersection of the bounding regions for the two sub-fields.

Layers rendered in the resulting distance field will contain patterns that reflect the composite nature of the field. Alternatively, the user may instead specify that layers are to be grown from the new shape. The difference in these approaches is illustrated in Figure 7.9.

7.3.2 Changing the Isosurface Velocity

To achieve the result shown in the second image of Figure 7.9, a new Euclidean distance field is created from the union-ed field. Similarly, a non-Euclidean distance field can be initialized from a Euclidean field, which is how I implemented the isosurface velocity variations described in Chapter 4 to create objects with layers having varying isosurface velocities (Figure 4.7). One isosurface will match between the two fields; usually this is the zero isosurface. To make the zero isosurface of the new distance field f_2 match the i isosurface of the base field f_1, a band of values surrounding the isosurface is initialized in the following manner.

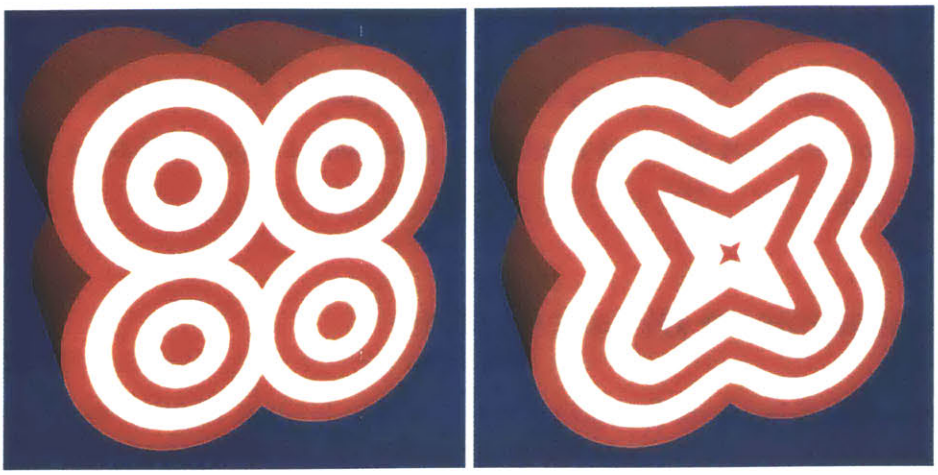


Figure 7.9: In the left image, the distance fields of four cylinders have been combined with the union operation, and the interior is filled with alternating layers of red and white material. On the right, the combined distance field is revised to be a Euclidean measurement from the zero isosurface, resulting in a different pattern of layers.

1. First the algorithm verifies that f_1 has been consistently initialized for the i isosurface. For all points p:

   - for all 26 neighbors n, $f_1(n)$ is *KNOWN* OR
   - $f_1(p) > i$ and for all 26 neighbors n, $f_1(n)$ is *FAR AWAY* or $f_1(n) > i$ OR
   - $f_1(p) < i$ and for all 26 neighbors n, $f_1(n)$ is *FAR AWAY* or $f_1(n) < i$.

2. Then for each point p where $f_1(p)$ is *KNOWN*, the value of $f_2(p)$ is updated, similarly to Section 7.2.3. p_x, p_y, and p_z, the points $\pm$ 1 grid unit along each axis with the smallest value of f_2, are collected; however, if a neighbor lies on the opposite side of the isosurface (Figure 7.10), the distance value for that neighbor is defined in terms of $f_2(p)$. This definition is required, since initially there are no distance values available for f_2. The sign of $f_2(p)$ is the same as the sign of $f_1(p)$.

3. Repeat until the distance values of f_2 stop changing. Iteration is necessary to propagate distance information within a band near the isosurface, but only a few iterations are needed.

Since the sign is set from f_1, which is consistently initialized, field f_2 will also be consistently initialized, and it can be propagated as described in Section 7.2.3 without running the cleanup

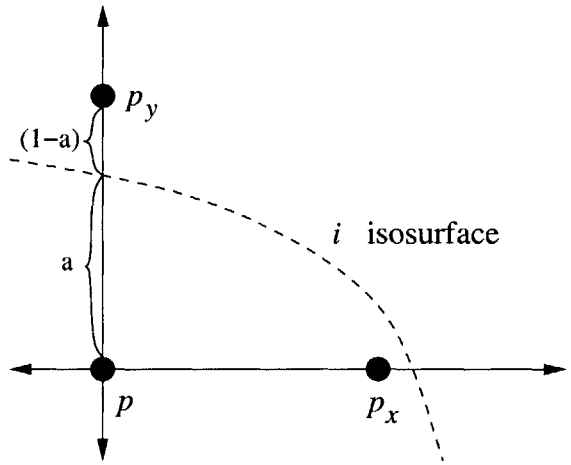
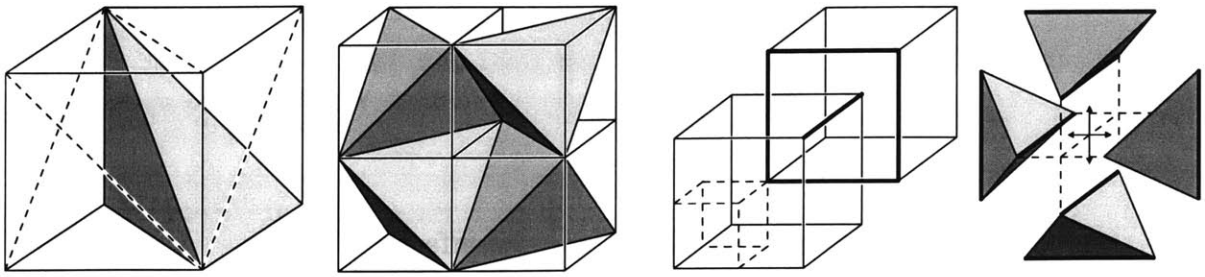


Figure 7.10: If the i isosurface of f_1 passes between p and one of its neighbors p_y, the value $f_2(p_y)$ is defined as $\frac{-(1-a)}{a} \cdot f_2(p)$, where $a = \frac{f_1(p) - i}{f_1(p) - f_1(p_y)}$.

procedure of Section 7.2.4. The velocity values for the new field can be defined in a number of different ways. The easiest to evaluate is an implicit velocity function that, like an implicit surface function, has a well-defined value at every point in space. Alternatively, the field can be defined as a function of the local isosurface properties that can be extracted from the distance field, such as normal direction, curvature, etc. Or the velocity values can be "painted" on the surface and carried along to subsequent isosurfaces. Isosurface velocity must be positive in order to make use of the Fast Marching technique. However, this is not a limitation in our application, since negative isosurface velocities would imply overlapping volumes, which is not physically plausible. Many other signed distance field applications are examined in Sethian [1999].

7.4 Tetrahedralization

Once the signed distance function (discussed in Section 7.1) has been initialized, I use a *structured method* for creating tetrahedral meshes based on an axis-aligned grid or octree [Yerry and Shephard 1984], which is the volumetric equivalent of the techniques for constructing surface meshes from distance fields [Lorensen and Cline 1987, Bloomenthal 1994, Bloomenthal and Ferguson 1995]. My approach is an extension of the Interval Volume Tetrahedralization (IVT) method [Nielson and Sung 1997].



a) congruent b) alternating c) tetrahedral lattice

Figure 7.11: Illustration of three different tetrahedral packing methods. If space is first divided into cubes, each cube can then be subdivided into a) six congruent tetrahedra with reflections or b) one large equilateral tetrahedron and four smaller tetrahedra in the corners. If the method shown in a) is used, each cube is divided along the same long diagonal. However, if the method shown in b) is used, the decompositions of neighboring cubes must be alternated to align neighboring faces. A third method of packing is the c) tetrahedral lattice, which is created from two offset grids. Four tetrahedra surround each edge, connecting that edge to the corresponding square of edges in the offset grid, shown with thick lines. All tetrahedra in the lattice are congruent and well-proportioned, though not equilateral.

The octree method is robust and simpler to implement than *unstructured methods* such as advancing front and Delaunay methods [Lohner 1988, Baker 1989]. Unstructured methods produce a mesh independent of object orientation and attempt to match the vertices and faces of the original surface mesh. Because I want to manipulate large scanned surface meshes with the system, matching the surface is usually unnecessary and even undesirable. For example, most scanned meshes contain much more detail than can be manipulated interactively. A pre-process stage that simplifies the input surface can reduce the number of surface elements to a manageable number, but the new surface sampling may not be optimal for creating well-proportioned volumetric elements. Both structured and unstructured mesh generation techniques are usually used in conjunction with mesh simplification and improvement, which will be discussed in Chapter 8.

7.4.1 Structured Mesh Generation

In the first step of my tetrahedralization algorithm, each *cubic grid cell* of the signed distance field is divided into five *tetrahedral cells*, alternating the orientation of the central tetrahedron so that diagonals match on adjacent cubic cells. This decomposition along with two other methods for filling space with tetrahedra is illustrated in Figure 7.11. I chose not to use the six-tetrahedron decomposi-

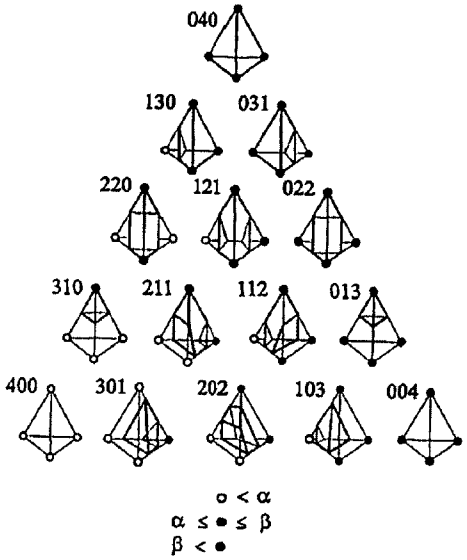


Figure 7.12: This table, from Nielson and Sung [1997], enumerates the fifteen cases of vertex labeling with respect to two isosurfaces. Each tetrahedral cell is partitioned into different materials as appropriate.

tion because it results in more tetrahedra and requires an edge along the long diagonal of the cube, which can lead to additional interpolation artifacts on the internal and external boundaries. Another option would be the tetrahedral lattice, a very elegant decomposition of space [Molino et al. 2003]. All elements in the lattice are congruent and very well-proportioned; however, I chose not to use this method because it is more complex to visualize and instantiate, since it does not partition into cubes. Also, a lattice tetrahedralization would only use a quarter of the regularly-sampled signed distance field grid values.

7.4.2 Layer Boundaries

In the next step, each tetrahedral cell is divided into tetrahedra of the appropriate materials, similarly to Nielson and Sung [1997], which is illustrated in Figure 7.12. However, I do not explicitly enumerate these cases in my implementation. If the distance values of all four vertices of a tetrahedral cell are within the range for a single layer, one tetrahedron of that material is created. If the vertices are within different layer ranges, the cell is split at a best-fit planar isosurface crossing along one of its edges, and then the two resulting cells are handled recursively. To ensure that the final mesh is well-defined, the faces of neighboring tetrahedral cells must be assigned the same

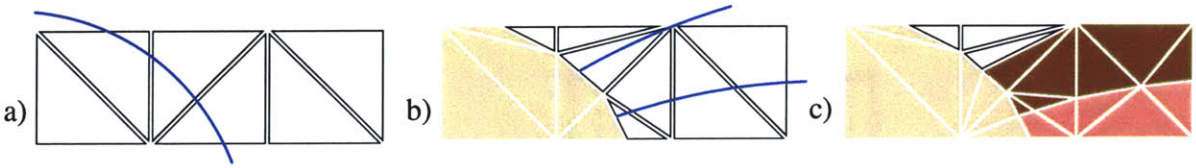


Figure 7.13: Multiple volume descriptions can be combined with the precedence construct and are tetrahedralized serially: a) cubic grid cells are split into tetrahedral cells, b) tetrahedral cells are split at the isosurface crossings of the first volume and materials are assigned, and c) unassigned volume is split at the isosurface crossings of subsequent volumes. Note that previously assigned volumes may require further splitting to handle non-manifold intersection points.

triangular decomposition. The following protocol for ordering edge splits based on vertex and isosurface identifiers guarantees a proper mesh with matching tetrahedral faces and no T-junctions:

1. If the tetrahedral cell spans more than one isosurface boundary, split an edge on the most interior isosurface first.

2. If multiple edges span a particular isosurface, examine the unique identifiers of the vertices to determine which edge to split first. When comparing e_1, the edge between vertices v_a and v_b with $\mathrm{id}(v_a) < \mathrm{id}(v_b)$, to e_2, the edge between vertices v_c and v_d with $\mathrm{id}(v_c) < \mathrm{id}(v_d)$:

   - If $\mathrm{id}(v_a) < \mathrm{id}(v_c)$ *OR* $(\mathrm{id}(v_a) = \mathrm{id}(v_c)$ and $\mathrm{id}(v_b) < \mathrm{id}(v_d))$, split edge e_1 first.
   - If $\mathrm{id}(v_a) > \mathrm{id}(v_c)$ *OR* $(\mathrm{id}(v_a) = \mathrm{id}(v_c)$ and $\mathrm{id}(v_b) > \mathrm{id}(v_d))$, split edge e_2 first.

This algorithm places no constraints on the thickness of layers or the number of isosurface crossings allowed per tetrahedral cell.

7.4.3 Precedence of Distance Fields

If a tetrahedral cell is not assigned material by the first volume description in a precedence operation (Section 4.2), the tetrahedralization algorithm proceeds to the next volume description. When a tetrahedral cell is split, tetrahedra assigned by previous volume descriptions that share the edge are also split to prevent T-junctions within the mesh at non-manifold interface intersections [Bloomenthal and Ferguson 1995]. Figure 7.13 illustrates the technique with a two-dimensional example. The set of tetrahedra that share a particular edge of a tetrahedral cell are efficiently collected by storing all tetrahedral edges in a hash table, indexed by the vertices.

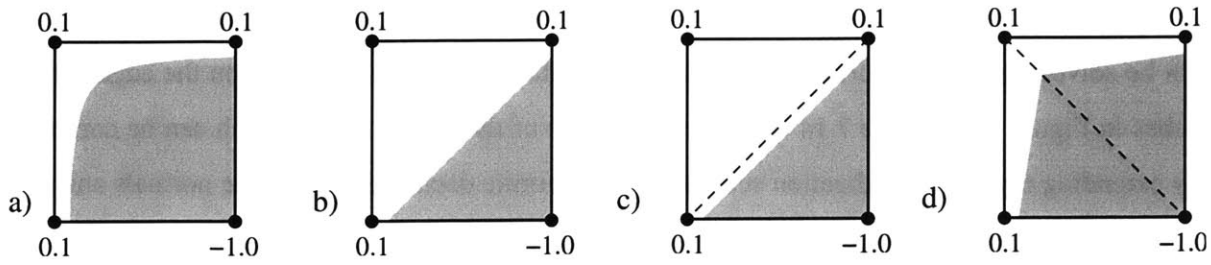


Figure 7.14: The sample distance field shown in this illustration can result in a sawtooth artifact when the zero isosurface is tetrahedralized. The artifact is caused by the polygonal approximation of the a) exact tri-linear interpolation. Using Marching Cubes results in b) a beveled corner, and if the cubic cells are first split into tetrahedral cells the result is c) or d), depending on the placement of the diagonal. Since the 5-way tetrahedral packing is alternated along the edge of the model, the extracted polygonal representation is also alternated, resulting in a sawtooth appearance.

7.4.4 Materials within a Layer

Procedural descriptions for the decomposition of material within a layer are evaluated after the entire volume has been tetrahedralized. The algorithm loops over all tetrahedra in the model that are assigned a composite material type, such as `WAVY_CHOCOLATE`.

1. Evaluate the procedural material definition at each vertex of the tetrahedron. If all vertices return the same material, change the material type of this tetrahedron to that material.

2. Otherwise, choose two vertices of the tetrahedron that have different material type and perform binary search along the edge to find a material interface crossing along the edge. Split the tetrahedron, and all of the neighboring tetrahedra, at that point.

3. Repeat until no tetrahedra have composite material type.

See examples in Figures 4.4 and 6.1. Procedural subdivision of layers into materials can suffer from the same resolution dependency problems as implicit surfaces. The user should specify an adequate grid spacing (Section 7.1.1) to ensure that all important features of the material are captured.

7.4.5 Artifacts of Tetrahedralization

The structured mesh generation technique does not attempt to match the triangulation of the original object. This becomes a problem when the original mesh is insufficiently sampled and results

in topological artifacts such as webbing, holes or disjoint regions. One particular artifact that cannot be solved simply by increasing the sampling is the *sawtooth* artifact seen on the edges of the cubes in Figure 7.3. Figure 7.14 illustrates the cause of the jagged edge, which can be corrected by extending my tetrahedralization scheme to use Hermite data, such as surface normals and grid intersections, for a more precise reconstruction [Ju et al. 2002].

7.5 Chapter Summary

In this chapter I described how the objects designed with my procedural solid modeling language are initialized as tetrahedral meshes. Meshes created by these algorithms consist of well-shaped elements within thick layers of material. However, one of the main disadvantages of the structured mesh generation approach is the large number of elements created and the poor shape of elements where the surfaces (both air/material and material/material) pass near the signed distance field grid points. The poorly-shaped elements usually have at least one very short edge relative to the underlying grid. In order to effectively use a tetrahedral mesh in a physical simulation such as FEM, one must eliminate or improve as many of the poorly shaped tetrahedra as possible. In the next chapter I will explain various simplification and mesh improvement operations to prepare these models for both interactive and offline simulations.

Chapter 8

Simplification and Improvement of Tetrahedral Models for Simulation

Three-dimensional mesh simplification is a mature subfield of mesh processing that aims to reduce the number of elements in a polygonal representation while maintaining important features of the geometry. Most three-dimensional tetrahedral mesh generation techniques require a simplification or mesh improvement stage to prepare a tetrahedral model for efficient simulation and interaction. As part of implementing the procedural solid modeling system described in this dissertation, I have developed a simplification algorithm that both reduces the number of tetrahedra in the model to allow online or interactive manipulation and removes the most poorly shaped tetrahedra to allow stable physical simulations such as the Finite Element Method (FEM).

The basic approach is to eliminate the poorly shaped elements (Section 8.1) while simplifying the model using edge collapses and other mesh operations, such as vertex smoothing, tetrahedral swaps, and vertex addition (Sections 8.2 and 8.3). The algorithm preserves both surface detail and the topology of internal and external material boundaries. In Section 8.4, I compare my algorithm to the Progressive Mesh technique [Hoppe 1996]. Though not an optimal remeshing of the original geometry, my solution quickly reduces the size of the model for similar results. Finally, in Section 8.5, I present the results of the algorithm on a variety of inputs, including models with more than a million tetrahedra. In practice, the algorithm reliably reduces meshes to contain only tetrahedra that meet minimum solid angle requirements.

8.1 Goals and Requirements

The simplification and mesh improvement algorithm is an important stage in the procedural solid modeling system. The input meshes shown in this chapter are generated with the grid-based mesh generation technique described in the previous chapter, although the algorithm can also be applied to meshes generated with other techniques. The grid resolution is set high enough to adequately capture the details of the input surface; however, the initial models have too many tetrahedra and the shape of many of the tetrahedra is poor, especially in places where the isosurface passes near the underlying rectilinear grid, making physical simulations impossible. There are a number of objectives, described below, that must be addressed in order for tetrahedral meshes to be useful in an interactive volumetric modeling system and behave correctly during simulation.

8.1.1 Reduce the overall number of tetrahedra

Generally, the running time of a physical simulation is polynomial in the number of elements. The FEM simulation engine for our system [Müller et al. 2001, Müller et al. 2002] operates at interactive rates with models of 5,000 or fewer tetrahedra, so this is often the target tetrahedral count for simplification. A mesh with this few tetrahedra, and fewer triangular boundary faces, will probably not preserve the fine details of the initial surface. However, a higher resolution triangular surface may be animated by interpolating the results of a simulation on a lower resolution tetrahedral model. Offline simulations can handle significantly larger models, but the problems of poorly shaped tetrahedra still remain.

8.1.2 Maintain the shape and topology of boundary surfaces

One of the main goals of any simplification algorithm is to preserve the shape of boundary surfaces. However, some surface details must be sacrificed to achieve the target tetrahedral count. It is important to maintain both external (material/air) and internal (material/material) boundaries, but they may have different preservation requirements.

In my simplification strategy, the topology of the object and the topology of the materials within the object are strictly preserved. For example, if the object is covered by a thin layer of material, the simplified object should also have this layer. A naive implementation might allow the thin

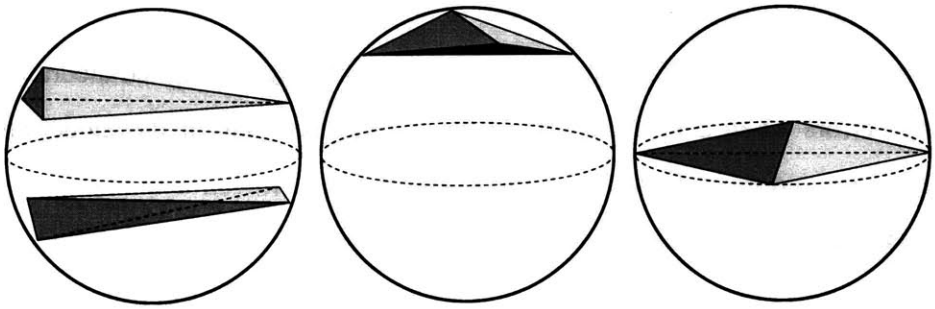


Figure 8.1: Illustration of several types of poorly-shaped tetrahedral elements, from Shewchuk [1997]. Unlike the two-dimensional case, it is not sufficient to judge element shape quality by the ratio of shortest and longest element edges. While this test detects needle and wedge tetrahedra, it fails for sliver tetrahedra. A better metric in three dimensions is the minimum solid angle.

layer to *collapse*, so that the interior material becomes visible. Other examples of lossy internal boundary simplification are shown and discussed by Ju et al. [2002].

Noise and alignment errors in the acquisition of three-dimensional scanned meshes often lead to higher genus models than the original object. Recent work by Guskov and Wood [2001] on the topological simplification of surface meshes addresses this problem and presents techniques for reducing the genus of the model to give a better representation for the same number of elements. Surface models can be pre-processed using these techniques prior to the application of the tetrahedralization and simplification algorithms. Alternatively, topological simplification of the tetrahedral model could be performed.

8.1.3 Improve the shape and proportion of each tetrahedron

Simulations such as FEM move vertices of the mesh relative to each other and place restrictions on the shape of volumetric elements in a particular model. Errors related to the finite element approximation increase for elements with extremely large dihedral angles, and the stiffness matrix is severely constrained when these angles are very small [Babuska and Aziz 1976, Krizek 1992]. In general, such tetrahedral elements are said to have poor *shape*, as illustrated in Figure 8.1. To ensure mathematical stability of the simulations and guarantee that the volume of each element remains positive, the initial model should consist of only well-shaped tetrahedral elements. For example, if a mesh contains a *sliver* tetrahedron, the slightest translation of one of its vertices can

cause the sign of its volume to change.

Tetrahedra with negative volume overlap the volumes of neighboring tetrahedra. These overlaps can cause errors when determining the exterior triangular skin of the model and result in incorrect renderings and missed collisions. For example, a tetrahedron with negative volume may have neighbors on all sides (and therefore none of these sides will be included in the triangle mesh) but have a vertex located outside of the computed triangle mesh. Adding constraints to the simulation to prevent inverted volumes introduces non-linearities to the system, making it expensive and difficult to solve. It is much more efficient to begin the simulation with well-proportioned elements that can withstand considerable deformation without introducing negative volume tetrahedra.

In two dimensions, the quality of a triangle's shape or proportions can be measured as a ratio of longest to shortest edges. However, in three dimensions, a tetrahedron with similar edge lengths is not guaranteed to have good shape; for example, a *sliver* tetrahedron. Liu and Joe [1994] evaluate several more effective three-dimensional metrics including: surface area to volume ratio, minimum dihedral angle (the angle between two faces of a tetrahedron), and minimum solid angle (the steradian measurement of unit sphere area covered by the three-dimensional angle at a tetrahedral vertex). They conclude that these metrics are equally effective in judging element quality. I have chosen to measure the quality of a tetrahedron's shape by its minimum solid angle.

While obtaining reasonable tetrahedral shape is the most important objective, I do not place a hard requirement on the minimum solid angle; rather, the algorithm works to improve element shape as much as possible, while also maintaining boundary quality and reducing the number of elements.

8.1.4 Maintain a reasonable distribution of elements throughout the volume

Simulations on uniform density meshes, in which all elements have similar volume and are nearly equilateral, will be very accurate and stable, but slow since there are so many elements. The same simulations will run faster on a mesh with gradation in size, with larger tetrahedra on the interior of the model and in areas of low detail, and smaller tetrahedra near the surface details.

The use of large elements on the interior of a mesh can yield dramatic simulation speedups, but if they are too large, the model may be over-constrained and will not behave realistically. For example, soft material deformations might look polygonal rather than continuous and the choice

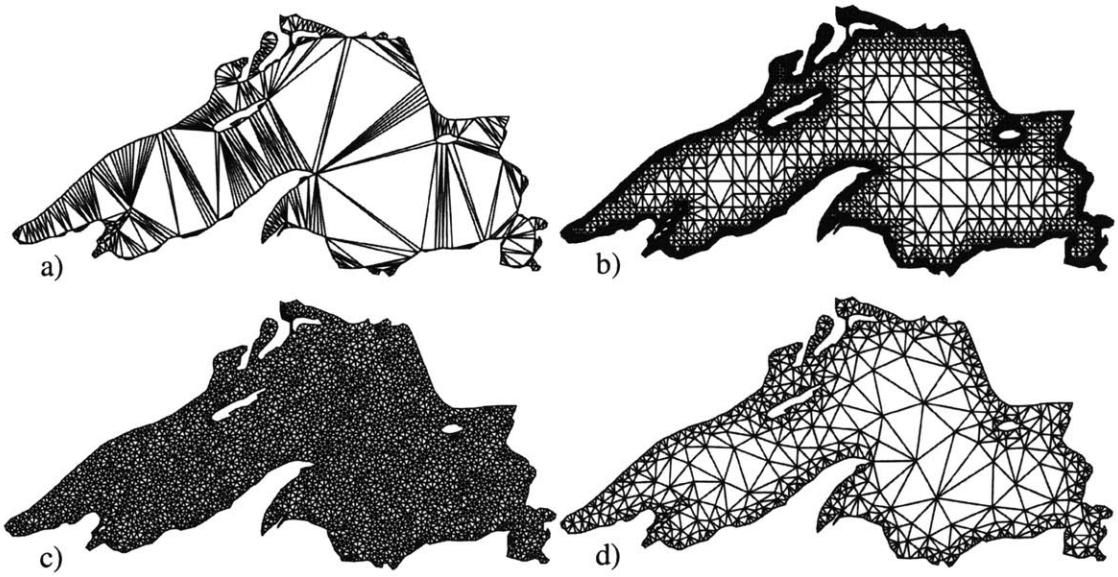


Figure 8.2: Sample two-dimensional triangular meshes of Lake Superior illustrate different element proportions and distributions. a) Triangulation with no additional interior points usually results in elements with greatly varying area and shape. The long narrow triangles might easily be deformed to have negative volume, resulting in numerical instabilities during simulation. b) A typical structured quad-tree mesh requires a very high density of triangles near the borders to guarantee that all boundary details are represented. Computation cycles are wasted on the elements near the boundary that have near-zero area. c) A uniform density meshing in which all triangles have nearly the same volume and are nearly equilateral. Simulations using this mesh will be very accurate and stable, but slow since there are so many elements. d) A compromise is reached that allows fewer triangles in the center of the mesh, but still represents the details of the boundary surface, and all triangles have reasonable proportions. (Meshes from [Shewchuk 1997]).

of fracture planes for a brittle material will be limited. These problems can be addressed during simulation by adaptively splitting tetrahedra that are under large amounts of strain. However, adaptive splitting is expensive and often too slow for an interactive simulation. To create the best mesh for a particular simulation, an upper limit on the element size may be specified.

Figure 8.2 illustrates several different element density and shape options for filling a complex boundary outline.

8.1.5 Offline Simplification

My simplification strategy was designed to be computed offline, as it only needs to be determined once per model and need not be interactive. However, to be useful in an iterative design process, the

results should be available in a few hours, even for the largest models. Also, the simplification must be robust. The input to the system is a manifold, consistently connected mesh with no negative volume tetrahedra; and the algorithm should never allow these properties to be violated. Finally, the simplification and mesh improvement strategy should be scalable, and handle very large models with more than a million tetrahedra.

8.2 Adaptive Distance Field

The *structured mesh generation* technique described in Chapter 7 was used to create most of the volumetric models presented in this document. A signed distance function is evaluated on a uniform, rectilinear grid and then each cell is tetrahedralized. To create high-resolution material boundaries, a high resolution distance field grid is required, which produces a large number of tetrahedra evenly distributed throughout the volume of the model without regard to the complexity of surface detail. However, if a material layer is thick relative to the grid, this leads to extraneous tetrahedral sampling within the layer.

By collapsing the distance field to an octree or *Adaptive Distance Field* [Frisken et al. 2000], the initial number of tetrahedra created is reduced, as shown in Figure 8.3. Prior to tetrahedralization, grid cells that are adequately represented by interpolation of the coarser grid or do not contain an interface crossing are collapsed. The user can specify that certain air/material or material/material boundaries should be represented at a higher resolution and with more accuracy.

8.3 Algorithm

The basic idea of my iterative simplification algorithm is to identify and correct poorly-shaped tetrahedra, which can cause stability problems during simulation. Also, the algorithm merges very small volume tetrahedra, which cannot remain if the tetrahedral count is to be significantly reduced. Once the poorly-shaped and small-volumed tetrahedra have been identified, the algorithm eliminates or improves them by using one or more of the atomic mesh operations presented below. My solution is similar to the mesh improvement strategy described by Freitag and Ollivier-Gooch [1997]. However, by combining the improvement strategy with aggressive simplification,

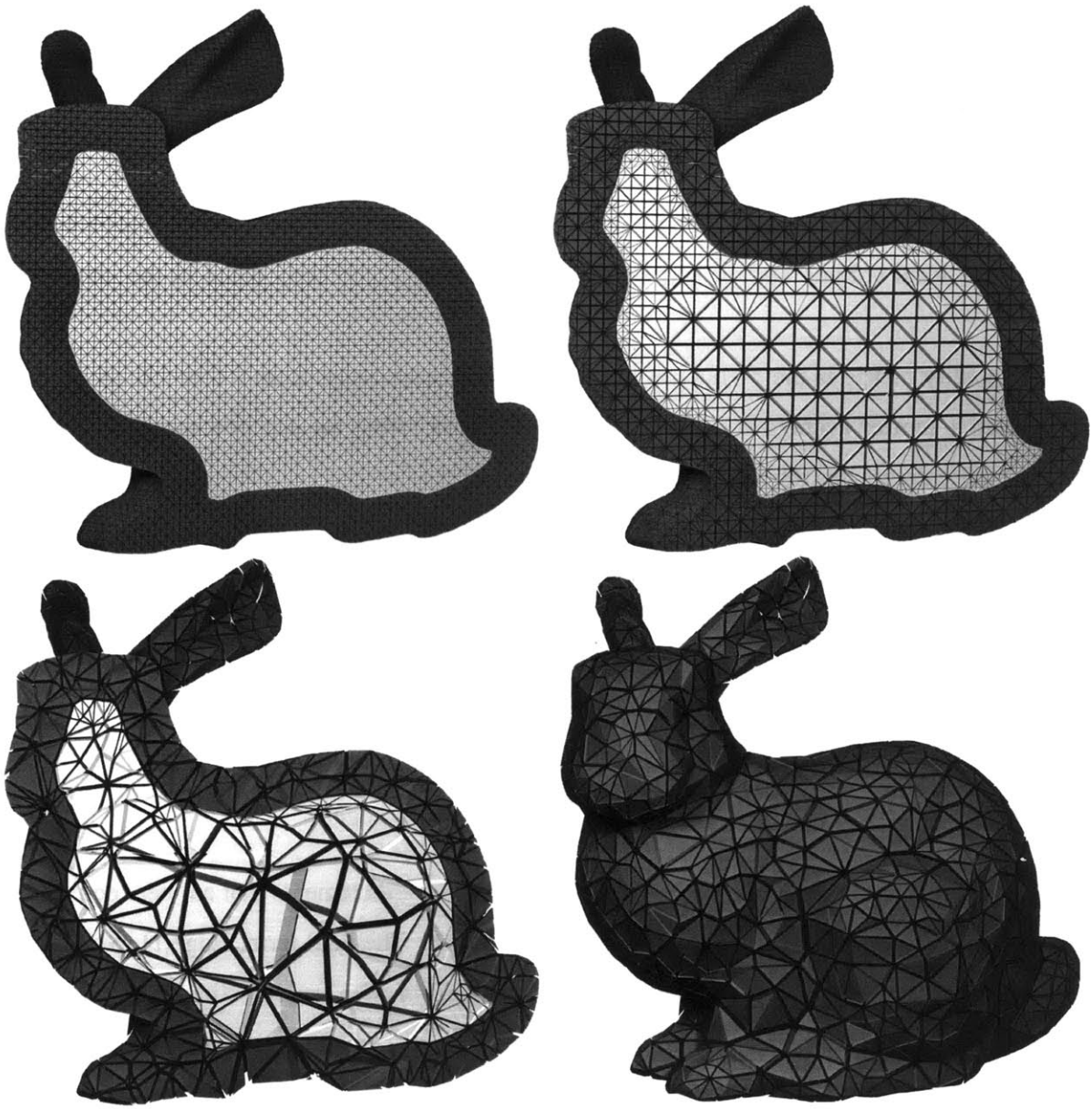


Figure 8.3: The mesh on the top left was created from a uniform distance field. The mesh on the top right was created from the same distance field after being collapsed to an octree, resulting in fewer than half as many tetrahedra. The meshes have similar boundary quality. Simplification is used to further reduce the size of the model, while preserving boundaries, as shown in the bottom images.

my method is more efficient and effective in practice.

To find the tetrahedra targeted for elimination or improvement, a quality metric q, ranging from 0.0 (very poor shape/volume) to 1.0 (near perfect shape/volume), is computed for each tetrahedron

t. The only user-specified parameter to the algorithm is the *target tetrahedral count*, although future extensions may add additional controls. The equations below reward tetrahedra that are close to equilateral (minimum solid angle ≈ 0.55 steradians) and have volume close to the *ideal volume* (total model volume / target tetrahedral count). If the final model can be represented with all similarly-volumed, near-equilateral elements, each element will have volume approximately equal to the ideal volume and all edges will have length approximately equal to the ideal edge length.

The quality metric is the product of three factors that each range from 0.0 to 1.0. A product rather than a sum or average is used to ensure that a poor rating for any one component of the quality metric dominates the score for the element.

$$
\begin{aligned}
\text{Quality}(t) &= \sqrt[3]{\text{Angle}(t) \cdot \text{Volume}(t) \cdot \text{Edge}(t)} \\
\text{Angle}(t) &= \text{minimum}\left(1, \sqrt{2 \cdot \text{min\_solid\_angle}(t)}\right) \\
\text{Volume}(t) &= \text{minimum}\left(1, \sqrt{\frac{\text{volume}(t)}{\textit{ideal volume}}}\right) \\
\text{Edge}(t) &= \text{minimum}\left(1, \frac{10 \cdot \textit{ideal edge length}}{\text{longest\_edge}(t)}\right)
\end{aligned}
$$

The first two terms, Angle(t) and Volume(t), simply reward elements that are near equilateral and have volume greater than or equal to the ideal volume. The final term, Edge(t), penalizes a tetrahedron whose longest edge is much greater than the *ideal edge length* ($\sqrt[3]{\text{ideal volume}}$). This specifically discourages the creation of large, poorly-proportioned tetrahedra that have reasonable volume and long edges. Such tetrahedra can be difficult to remove in later iterations. The "minimum" function is used to clamp each term at zero and prevent negative values.

Simplification and improvement is initially focused on the poorest 10% of all tetrahedra. The algorithm estimates a 10% quality cutoff value, gathers the tetrahedra with quality less than this value, randomly reorders the list, and then attempts to perform different types of atomic mesh operations on each tetrahedron. I describe these operations in detail later in this section. Some of the operations (edge collapses and vertex smoothing) can change the shape of the visible and internal boundaries, so they are only performed if within the current allowable weight, W.

W = *ideal edge length*
while (tetrahedral count $>$ *target tetrahedral count*)
    for i = 0 to 20
        q = estimate cutoff for poorest 10% of all tetrahedra
        Simplify_and_Improve(q, W)
    for i = 0 to 5
        Simplify_and_Improve(1.0, W)
    $W *= \sqrt[3]{\frac{\text{tetrahedral count}}{\textit{target tetrahedral count}}}$

Simplify_and_Improve (q, W)
    $T = \{ t \mid \text{Quality}(t) \leq q \}$
    randomly reorder T
    foreach $t \in T$, try these actions:
        - $3 \rightarrow 2$, $2 \rightarrow 3$, and $2 \rightarrow 2$ swaps
        - edge collapse
        - move a vertex
        - add a vertex

In general, the algorithm only performs operations that locally improve the model; specifically, the poorest quality tetrahedron after the operation should be no worse than the poorest quality tetrahedron before the operation. However, the algorithm may get stuck in a local minimum while using this greedy approach. So in the spirit of simulated annealing, I allow some small backward steps early in the computation and decrease the chance of taking these steps as the model approaches its target tetrahedral count.

After performing all available operations on the 10% poorest quality tetrahedra, the algorithm simplifies and improves all of the tetrahedra. This two-phase approach is important for efficiency early in simplification, to focus the computation where the model needs the most work. After each iteration, W, the allowable edge collapse weight, is increased (in smaller and smaller increments), until the target tetrahedral count is reached. A text visualization of initial and final element quality is shown in Table 8.4, and sample output from the initial iterations is shown in Table 8.8.

8.3.1 Swapping

The first actions attempted are tetrahedral swaps, which can dramatically improve tetrahedral shape by switching inter-element connectivity. I have implemented the two most common types of tetra-

Evaluate: 824834		lowest_quality: 0.000000			ten_percent_worst 0.078000			avg_quality: 0.322693		
>	0.0%					14:2		6:2		
10.0	0.1%				35:1	25:2	11:2			525:5
5.0	0.7%				36:2	1943:4	499:4	1119:5	2533:5	
1.0	2.1%				4066:3	3411:4	128:5	19:7	199:8	9576:9
0.5	7.1%				1370:3	1349:3	1971:6	12095:7	42131:7	11:7
0.25	13.3%			16:2	30935:3	41522:4	7917:5	2032:5	322:5	27166:5
0.1	20.9%			2276:2	10368:3	9421:3	6002:3	15089:4	126182:4	3178:5
0.05	22.0%		1314:1	16368:1	43236:2	39239:2	30563:3	39465:3	10817:3	587:3
0.01	23.8%	402:0	18918:0	91004:1	46278:1	16837:1	7806:2	8263:2	6146:2	383:2
0.001	9.9%	16308:0	26021:0	11842:0	6916:0	5904:0	3825:0	5316:0	5228:0	350:0
volume										
		2.0%	5.6%	14.7%	17.4%	14.5%	7.1%	10.1%	23.5%	5.1%
	angle	0.001	0.01	0.05	0.1	0.15	0.2	0.3	0.4	>

Evaluate: 100000		lowest_quality: 0.118466			ten_percent_worst 0.349000			avg_quality: 0.487434		
>	1.8%				9:1	64:1	123:2	627:2	717:3	229:3
10.0	2.3%				14:2	88:2	290:3	975:3	760:4	140:5
5.0	13.2%			12:2	464:3	1815:4	2848:4	5179:5	2453:6	478:7
1.0	11.7%			43:2	969:3	2480:4	2875:5	3860:6	1280:8	194:8
0.5	18.4%			142:2	2851:4	4947:4	4550:5	4569:6	1206:6	143:7
0.25	35.0%			545:3	6974:3	9779:4	7754:4	7897:5	1863:5	220:6
0.1	15.1%			155:2	3208:3	4104:3	3201:4	3376:4	966:4	109:5
0.05	2.5%			13:2	266:2	765:3	624:3	633:3	141:4	23:4
0.01	0.0%									
0.001	0.0%									
volume										
		0.0%	0.0%	0.9%	14.8%	24.0%	22.3%	27.1%	9.4%	1.5%
	angle	0.001	0.01	0.05	0.1	0.15	0.2	0.3	0.4	>

Table 8.4: Evaluation of mesh before (top) and after (bottom) the simplification and improvement algorithm. The quality of each tetrahedron is computed and dropped in a grid of buckets with volume on the vertical axis and solid angle on the horizontal axis. Ideally, after simplification and mesh improvement, all of the tetrahedra will move toward the bins in the upper right of this diagram, which are more equilateral in shape, and have volume close to the ideal element volume (1.0). Each bin displays the number of elements and ten times the average quality of those elements.

hedral swapping, shown in Figure 8.5, but other tetrahedral swaps are possible (see Freitag and Ollivier-Gooch [1997] for a complete list).

The $2 \rightarrow 3$ swap is performed by removing two adjacent tetrahedral elements and replacing them with three tetrahedra that share the edge connecting the two opposite vertices. The $3 \rightarrow 2$ swap is performed in reverse. Although the configuration with two elements contains fewer elements, the configuration with three elements is selected in some instances because it has a better local minimum element quality. The $2 \rightarrow 2$ edge swap is a special case swap which allows re-triangulation of the exterior surface mesh. A swap is not performed if it will introduce negative volume elements or if the air/material or material/material boundary surfaces are significantly modified.

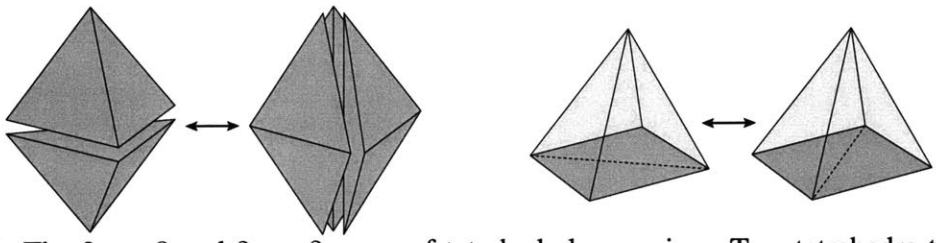


Figure 8.5: The $2 \leftrightarrow 3$ and $2 \leftrightarrow 2$ cases of tetrahedral swapping. Two tetrahedra that share a face may be replaced with three tetrahedra that share an edge (or vice versa). Two tetrahedra that share a face and have neighboring boundary faces that are nearly parallel can be replaced with two tetrahedra that essentially swap the edge between these two boundary faces. (Illustration redrawn from Freitag and Ollivier-Gooch [1997]).

8.3.2 Point Deletion

Next, the algorithm attempts to eliminate each target tetrahedron by performing a half-edge collapse, as illustrated in Figure 8.9. A half-edge collapse removes one vertex from the mesh and all tetrahedral elements surrounding the collapsed edge. Elements that pointed to the removed vertex are stretched to point instead to the vertex at the other end of the edge. There are two directions in which each of a tetrahedron's six edges may be collapsed. As illustrated in Figure 8.6, some of these collapses may be disallowed because they introduce negative volume tetrahedra or change the topology of the mesh[1]. Furthermore, some collapses are more desirable than others because they do a better job of preserving the boundaries or because they do not create lower quality tetrahedra.

The edge weighting function is used to prioritize edge collapses that best maintain surface details, and can be formulated in a number of ways [Garland and Heckbert 1997, Hoppe 1996]. I use a simple volume conservation metric that will preserve both exterior and interior boundaries. The change in volume, Δ volume, of each material is computed separately.

$$\text{Collapse Weight}(e) \quad = \quad \text{length}(e) \; + \; \gamma \cdot \sqrt[3]{\Delta \text{ volume}}$$

γ is a weighting factor which controls how much change in volume will be tolerated. Larger values result in better volume preservation (and likewise the retention of surface detail) but poorer element quality, while smaller values result in poorer volume preservation but improved element proportions. In the examples I use $\gamma = 20$. If no half-edge collapse has collapse weight less than

[1] Actually, multiple edge collapses are required to change the exterior surface topology, and the intermediate states are not consistent or manifold.

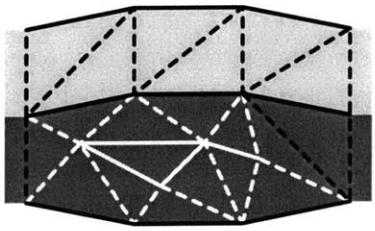


Figure 8.6: Two-dimensional illustration showing the types of edges that are considered for collapse during simplification. Interior edges (solid white) are completely surrounded by one type of material and can be collapsed, subject only to the criteria previously mentioned, since they do not affect the material boundary definitions. Boundary touching edges (dashed white) are surrounded by one material at their interior endpoint and two or more at their boundary endpoint. They should be collapsed to the boundary endpoint, and will have no effect on the material boundaries. Spanning edges (dashed black) are surrounded around the edge by a single material type, but have different materials at their endpoints and should never be collapsed, as this would cause a point of zero thickness in the material. Boundary edges (solid black) are part of the material/material or material/air interface definition. These edges can be collapsed as long as this interface is only minimally affected. This is measured by computing the distance from each of the edge endpoints to the new boundary mesh that would be created if the edge were collapsed.

the current allowable weight W, then no collapse will be performed on this tetrahedron, during this iteration.

More sophisticated surface preservation methods can be implemented to compare the resulting mesh after a proposed edge collapse to the original surface rather than the current intermediate model. This would prevent accumulated errors which might otherwise allow the mesh to shrink or deform noticeably.

8.3.3 Smoothing

If the procedures outlined above are unable to improve the shape of the targeted tetrahedron, the algorithm then attempts to adjust the position of its vertices. A simple, efficient, and surprisingly effective method is to move each vertex to the centroid of its connected vertex neighbors [Freitag and Ollivier-Gooch 1997], as shown in Figure 8.7. However, one must be careful that the new position does not introduce any tetrahedra of negative volume, and that the proposed movement does not unacceptably degrade air/material or material/material boundary surfaces. I monitor boundary surface quality with the volume preservation criteria described in the previous section.

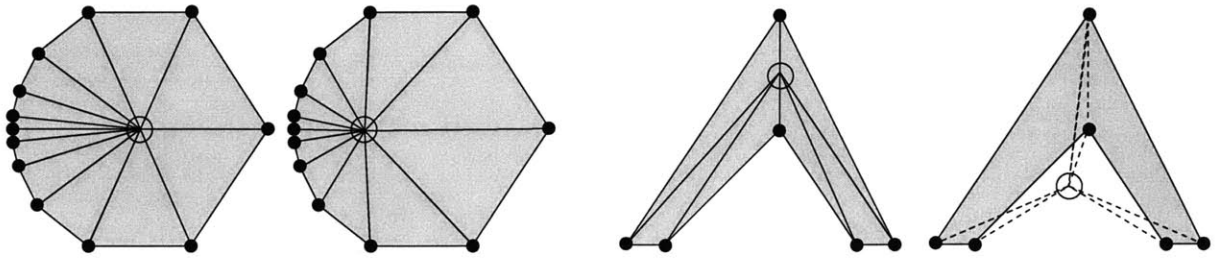


Figure 8.7: A simple vertex smoothing operation moves the position of the circled vertex to the average of all neighboring vertices (solid black circles), as shown on the left. However, the operation may not be performed if the new position results in negatively volumed elements, as shown on the right.

8.3.4 Point Addition

A final mesh operation option is the addition of *Steiner points* [Baker 1989, Conraud 1995, Shewchuk 1997, Shewchuk 1998, Cavalcanti and Mello 1999, Fleischmann 1999, Shewchuk 2000, Shewchuk 2002]. A new vertex can be added at the center of a tetrahedron, at the center of a boundary face or at the midpoint of a boundary edge. Although these actions will increase the total number of tetrahedra, they can provide significant help for improving the shape. I implemented this capability, but did not find it particularly helpful in the examples; it just created more simplification work. This operation is probably most useful when the initial model does not contain many more elements than the target count, and point addition is necessary to allow shape improvement. More investigation is needed to evaluate the conditions for which point addition is most effective.

8.3.5 Mesh Consistency

Throughout the different operations that are performed on the model, it is important to maintain a topologically consistent object. This includes updating tetrahedron-tetrahedron and tetrahedron-triangle neighbor pointers and invalidating cached values such as edge collapse weights and vertex normals. Consistency checks include ensuring that exactly two tetrahedra or one tetrahedra and one boundary triangle share each triangular face, that exactly two triangles share each surface edge, that at least three triangles meet at each surface vertex, that at least three elements meet at each edge, and that all tetrahedra sharing a vertex are connected by following tetrahedral neighbor pointers. Some of these criteria are guaranteed if all elements have positive volume, but because

```
simplify     current count:824834  target count:25000

 10%     10%     swap    swap    swap       edge      move     edge    step              after      time
qual    count    3->2    2->3    2->2   collapse    vertex       wt    back    none      count   h:mm:ss
-------------------------------------------------------------------------------------------------------
0.05    83069     873    3793    7731      34922      7288             2565     436     729471   0:03:57
0.10    74137    2178    2538    6394      33474      5685        1    1895     928     634331   0:07:25
0.14    65495    2579    1654    4910      29650      4975        1     934    1937     544765   0:10:28
0.18    55933    2494    1318    3085      23873      5190       17     740    3000     467562   0:13:10
0.22    47499    1770    1347    2002      18309      4945       94    1409    3727     403208   0:15:34
0.25    40387    1212    1616    1008      14787      4251      146    2660    2938     343753   0:17:54
0.26    35194     827    2144     549      13959      4856      178    3666    2152     286780   0:20:34
0.28    28948     912     454     468       8646      4658      551    5245    1954     248336   0:22:29
0.31    25381     700     183     400       5284      4521     1140    7237    1786     223600   0:24:08
0.32    22984     531     127     285       3413      4334     1741    8713    1835     206551   0:25:44
0.33    21279     373      91     183       2346      3553     2130    9574    1834     194548   0:27:10
0.34    19782     267      81     135       1695      2765     2485    9781    1625     185749   0:28:28
0.34    18922     207      69     111       1187      2271     2854    9859    1773     179580   0:29:39
0.34    18340     188      61      59        848      1703     3190   10164    1765     175225   0:30:44
0.35    17890     133      48      71        566      1207     3453   10377    1800     172261   0:31:45
```

Table 8.8: After each iteration of the Simplify_and_Improve subroutine, the program prints a line of statistics. Above are the first 15 lines of output for a simplification run to reduce a mesh with over 800,000 tetrahedra to 25,000 elements. The columns from left to right are: 10% quality threshold estimate and the count of elements with quality less than the threshold; the number of tetrahedra improved by 3→2, 2→3, and 2→2 swaps, edge collapses and move vertex operations; the number of elements not modified because the edge collapse weight was too high, the action was a step backward (resulting in worse quality tetrahedra), or no action was available; the number of elements at the end of the iteration; and the total cumulative running time.

the initial model may contain zero volume elements and due to limited floating point resolution, these checks should be consistently performed. See also Section 6.3.4.

8.4 Comparison to Progressive Mesh Technique

Initially, the simplification phase of the system was implemented using the Progressive Mesh (PM) technique of Hoppe [1996] and its extension to tetrahedral models [Staadt and Gross 1998]. Each edge in the mesh is given a weight which indicates how much its collapse will affect various mesh properties such as number of elements, conformity to the original mesh and mesh normals, element shape, element coloring or materials, etc. The edges are stored in a priority queue, and after each edge is collapsed, neighboring edge weights are recomputed and re-heaped as necessary.

The PM technique is very elegant, however, it can be difficult to select an appropriate weight function for a particular application. I attempted to incorporate terms in the weighting function that would improve tetrahedral shape, but unfortunately these restricted the level of simplification reached by the algorithm, and did not guarantee that all poorly shaped tetrahedra were eliminated.

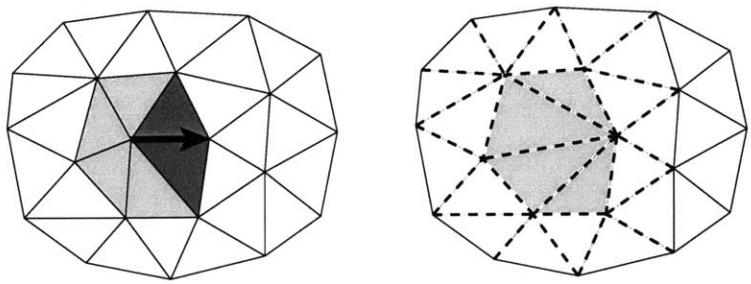


Figure 8.9: When an edge is collapsed (black arrow), the triangles or tetrahedra that share that edge are removed (dark gray) and the elements that touch the deleted vertex are deformed (light gray). If the edge weighting function for a Progressive Mesh includes data about element shape, all edges that share a vertex with a deformed element (dashed lines) must be recomputed. In this two-dimensional example, 27 edge weights are recomputed, and in three dimensions, on average, over one hundred edge weights must be recomputed.

Although I could adjust the weighting function to achieve some of the goals for specific models, I found it impractical to hand-tune parameters for each model. Moreover, since the PM technique only performs edge collapses, it cannot improve the element shape of meshes that are already near their target tetrahedral count or contain very poor inter-element connectivity — that is, too many or too few elements per vertex or edge.

Another problem with using the PM technique on tetrahedral elements is the large cost of re-computing edge weights and re-heaping the priority queue. On average over 100 edge weights must be recomputed and re-heaped for each edge collapse (see Figure 8.9). Edge weight computation is expensive since many element connection and consistency conditions need to be checked. The cost of actually performing the edge collapse is small compared to the cost of determining whether a particular collapse is legal and desirable. With the algorithm presented in Section 8.3, the edge weights are initially computed only for low quality tetrahedra, and it is very likely that if a legal collapse exists, it will be performed.

The increased cost of the optimal PM solution is apparent when performing extreme simplification on very large models. Unlike the PM, which is a total ordering of edge collapses, my method is a partial ordering, and in the early stages, many operations are performed in an arbitrary order during each iteration, which saves considerable computation. The per-iteration edge weight increase, ΔW, is decreased as the target tetrahedral count is approached. In the limit, as $\Delta W \rightarrow 0$, my method (restricted to half-edge collapses) is equivalent to the continuous simplification of a Progressive Mesh.

8.5 Performance

The simplification and mesh improvement implementation has been stable and effective, even on very large meshes as shown in Figures 8.10 and 8.12. Mesh statistics and running times are summarized in Table 8.11. Simplifications with smaller target tetrahedral counts have faster running times because the initial value for the edge weight cutoff, W, is higher, resulting in faster reduction of the tetrahedral count and fewer elements to process in later iterations.

In practice, all tetrahedra far exceed minimum quality requirements for simulations (Table 8.4). The simplified models have been successfully stress-tested in our interactive system by unsympathetic users and perform quite well [Müller et al. 2002]. Creating models for use in interactive systems is challenging because of the fixed time steps and minimum required refresh rates.

The implementation is efficient enough to be used during iterative design without being a major bottleneck. The offline simplification strategy could be adapted for interactive applications by restricting its use to local mesh improvement where simulation or sculpting have modified the mesh and created poorly-shaped tetrahedra.

8.6 Chapter Summary

In this chapter I presented the details of the tetrahedral mesh simplification algorithm. My strategy takes a novel approach in directly focusing mesh improvement operations on what I have defined to be the poorest quality elements. The technique, as currently formulated, could benefit from formal definition and more thorough studies of its effect on mesh quality.

Simplification is a necessary component in the overall procedural solid modeling framework. It provides a model at a resolution that supports interactive exploration and manipulation. In the next chapter I describe techniques for visualization and interaction with these models, which allow a user to perform physically inspired sculpting operations on complex objects designed in the modeling language.

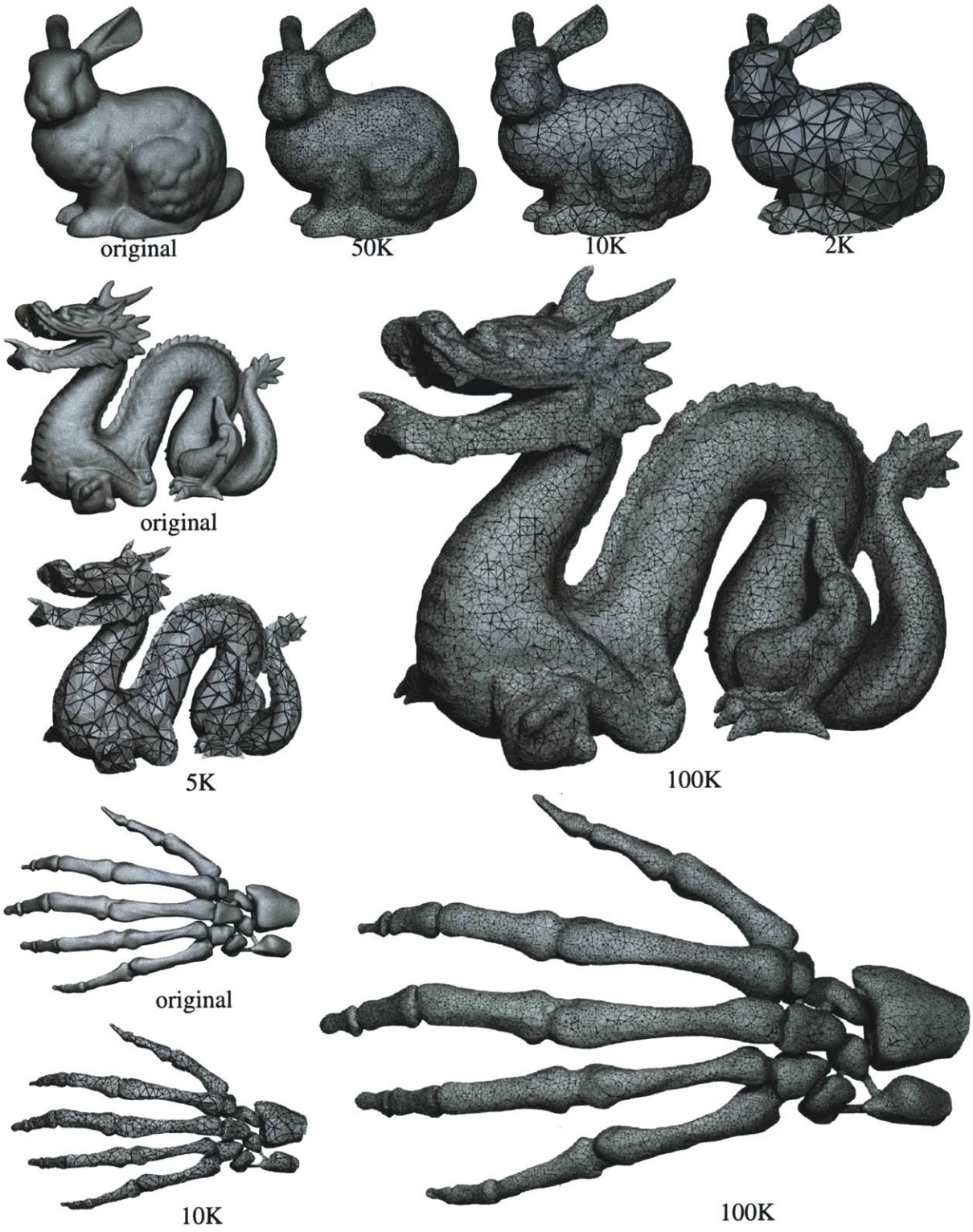


Figure 8.10: Renderings of representative results. The original bunny and dragon meshes are from Stanford University and the original hand mesh is from Clemson University.

Model	grid	uniform mesh		adaptive mesh		quality before		target		after simp. and impr.		quality after		running time (h:m:s)
Bunny	89x88x69	236,635	v	112,326	v	0.22	avg			280	v	0.59	avg	
		133,656	f	102,742	f	0.05	10%			352	f	0.41	10%	0:21:17
		1,050,903	t	453,804	t	0.00	low	1,000	t	996	t	0.22	low	
										541	v	0.58	avg	
										674	f	0.40	10%	0:18:52
								2,000	t	1,999	t	0.25	low	
										1,277	v	0.55	avg	
										1,490	f	0.39	10%	0:22:18
								5,000	t	4,999	t	0.22	low	
										2,493	v	0.54	avg	
										2,736	f	0.38	10%	0:31:44
								10,000	t	9,999	t	0.12	low	
										12,556	v	0.51	avg	
										13,876	f	0.35	10%	0:57:28
								50,000	t	49,998	t	0.09	low	
		454,809	v	232,328	v	0.32	avg			1,849	v	0.53	avg	
Dragon	171x120x77	355,296	f	297,994	f	0.08	10%			3,402	f	0.38	10%	0:36:53
		1,875,113	t	824,834	t	0.00	low	5,000	t	5,000	t	0.23	low	
										6,757	v	0.50	avg	
										11,854	f	0.36	10%	0:50:14
								20,000	t	19,999	t	0.21	low	
										16,198	v	0.48	avg	
										27,520	f	0.34	10%	1:01:33
								50,000	t	50,000	t	0.12	low	
										30,130	v	0.49	avg	
										48,120	f	0.35	10%	0:42:10
								100,000	t	100,000	t	0.12	low	
		2,311,453	v	458,571	v	0.27	avg			1,976	v	0.58	avg	
Hand	439x309x151	1,106,992	f	602,902	f	0.06	10%			3,654	f	0.42	10%	1:24:28
		5,902,522	t	1,596,397	t	0.00	low	5,000	t	5,000	t	0.25	low	
										3,835	v	0.55	avg	
										7,084	f	0.41	10%	1:19:25
								10,000	t	9,999	t	0.22	low	
										10,547	v	0.53	avg	
										18,728	f	0.39	10%	1:42:27
								30,000	t	30,000	t	0.23	low	
										17,194	v	0.51	avg	
										30,162	f	0.38	10%	1:59:05
								50,000	t	49,995	t	0.17	low	
										33,201	v	0.49	avg	
										56,914	f	0.36	10%	2:19:19
								100,000	t	99,997	t	0.14	low	
		487,371	v	221,039	v	0.16	avg			3,361	v	0.51	avg	
Gargoyle	207x86x74	353,336	f	279,272	f	0.04	10%			6,016	f	0.36	10%	0:37:11
		2,044,684	t	792,125	t	0.00	low	10,000	t	9,997	t	0.22	low	
										9,493	v	0.50	avg	
										16,130	f	0.35	10%	0:56:34
								30,000	t	29,999	t	0.20	low	
										15,644	v	0.49	avg	
										26,208	f	0.34	10%	0:48:46
								50,000	t	49,999	t	0.13	low	
										51,518	v	0.53	avg	
										63,040	f	0.36	10%	0:40:21
								200,000	t	199,999	t	0.05	low	

Table 8.11: Mesh statistics for each model shown in Figures 8.10 and 8.12. The simplified meshes were generated from the corresponding adaptive mesh on a Pentium 4 Xeon 2.4 GHz machine with 4 GB of RAM.

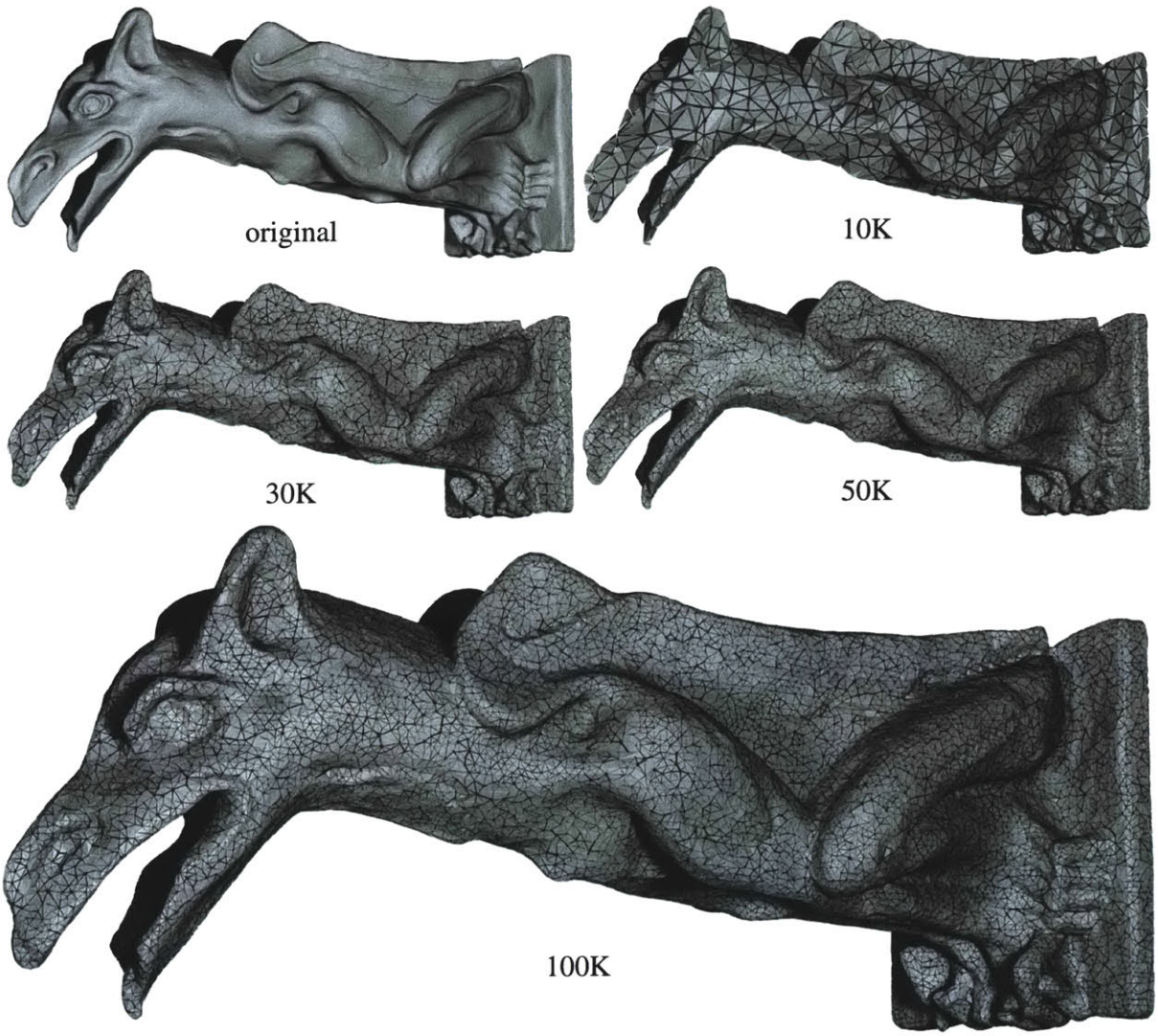


Figure 8.12: Renderings of the gargoyle model at several resolutions.

Chapter 9

Visualization and Interaction

A solid modeling system that is useful for designing (rather than simply generating) volumetric objects must support interactive visualization and exploration. In particular, the user should be able to quickly visualize the visible air/material exterior surface, the material/material boundaries that define the internal structures, and the density and shape of the volumetric elements that form the object. Also, it is important that the user be able to interactively view variations within the materials to verify that the material parameters, which will be used in simulations and high-quality offline renderings, have been set appropriately. Physical simulation is a powerful modeling resource, but to truly be an artist's tool for authoring models, it must be controllable through an intuitive and responsive interface. In this chapter, I describe several visualization and interaction techniques, including triangle rendering style (Section 9.1), normal interpolation (Section 9.2), texture mapping (Section 9.3), tetrahedral visualization (Section 9.4), and the interactive tool interface (Section 9.5). These techniques were developed in collaboration with Rob Jagnow and Matthias Müller.

9.1 Triangle Rendering Style

The primary mode of display for the interactive system draws only the surface triangles — that is, the unpaired faces of the tetrahedral mesh. The list of triangles is maintained during sculpting and simulation with minimal overhead. Therefore, rendering a complex volumetric model is only as expensive as rendering the surface mesh, because the complexity of the internal structures do not affect the rendering time. The material/material tetrahedral boundary faces can also be efficiently stored and maintained to correctly model and render translucent materials.

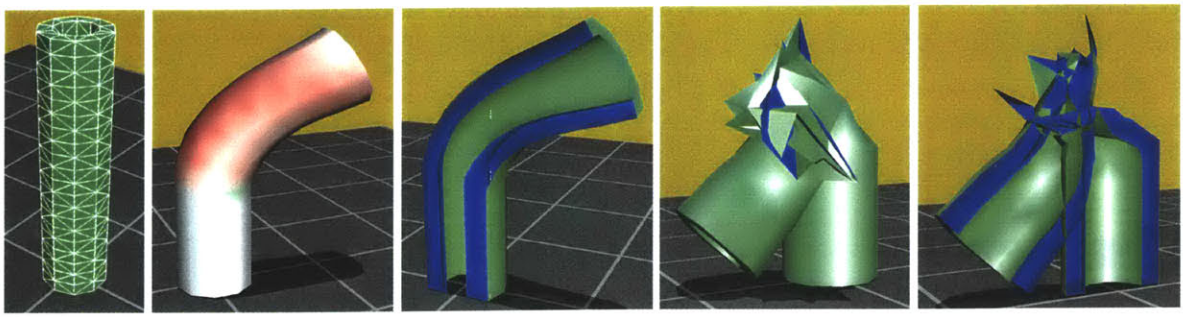


Figure 9.1: A simple jello tube rendered in wireframe style, stress visualization, and clipped to show thickness. If the blue-colored, interior surfaces are visible during normal display mode, the model contains negative volume tetrahedra, which indicates that the simulation has violated the mesh consistency properties.

Normally the surface triangles are colored or textured to represent a particular material, but we may also visualize other model properties. For example, the tensile and compressive stresses computed by the Finite Element Method (FEM) simulation module may be displayed.

The user can control the position of one or more clipping planes to slice through the object and view complex surface structures, such as holes or knots. For visualization, back-facing polygons are not culled, but instead are colored bright blue, highlighting the difference between interior and exterior surfaces (Figure 9.7 b and c). This visualization style is very useful when debugging a volumetric model or simulation. If the blue side of the triangle is ever visible without the clipping plane, then the model must contain a negative-volume tetrahedron that has pierced through the exterior boundary. However, it is important to note that the lack of visible blue faces does not guarantee that all tetrahedra have non-negative volume; problems may be present on the interior of the mesh. These triangle rendering styles are illustrated in Figure 9.1.

9.2 Smooth and Sharp Edges

Interpolated triangle normals improve the appearance of polygonal approximations of curved surfaces, especially for low-resolution meshes [Woo et al. 1999]. One interpolation approach involves averaging the normals across all faces and edges. A better approach distinguishes between smoothly curving regions and sharp edges, as shown in Figure 9.2, by using a *sharpness threshold angle*. Instead of sharing a single normal among all triangles at each vertex, each triangle stores three normals for its vertices. These normals are computed efficiently as follows: For each vertex,

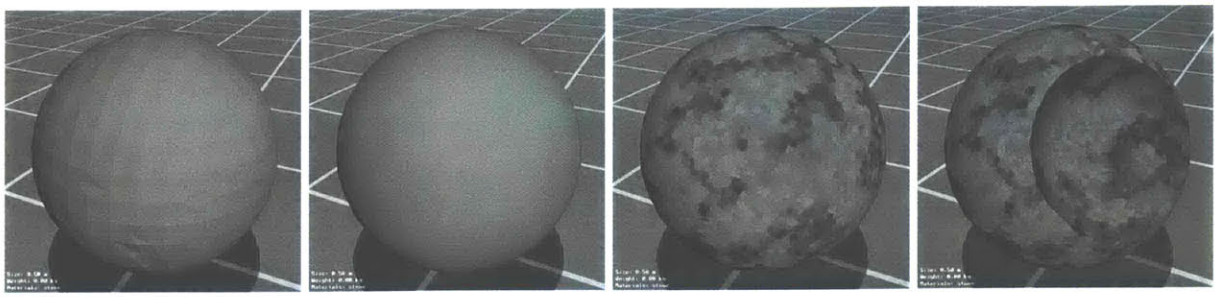


Figure 9.2: Screen shots showing the different rendering options in our interactive system. From left to right: flat shaded polygons, interpolated normals, texture map computed from solid stone texture, and sharp edges and recomputed textures after a CSG sphere subtract tool operation.

the triangles that share that vertex are collected and ordered in a ring around the vertex using the triangle neighbor information. An edge between a pair of faces in this ring is sharp if the dihedral angle is less than the sharpness threshold[1]. Finally, for each set of triangles between sharp edges, a weighted average of the face normals is computed and assigned to each triangle.

This algorithm for computing smooth normals with sharp edges is linear in the number of vertices and triangles. However, the overhead of collecting neighbors and checking normals is too expensive to recompute for each redraw during an interactive simulation such as rigid body animation. Fortunately, it is simple to use the transformation matrix of the rigid body to also transform the cached smooth normals.

However, this rendering strategy is not perfect. The interpolated appearance is highly dependent on the connectivity and proportion of the surface triangles, and certain geometry may be ambiguous or impossible to shade, as shown in Figure 9.3. In particular, the initial tetrahedralizations, generated as described in Chapter 7, usually contain sliver triangles and thus may have appearance artifacts when rendered with smooth normals.

9.3 Texture Mapping

Two-dimensional textures can add complex surface detail to a scene with limited geometry. Similarly, solid textures are an important component of volumetric models. For example, materials with spatially-varying properties, such as wood or stone, are often represented with *procedural solid textures* — functions that return a color when queried with a three-dimensional point.

[1] We have found that a threshold of 150 degrees works well for most models.

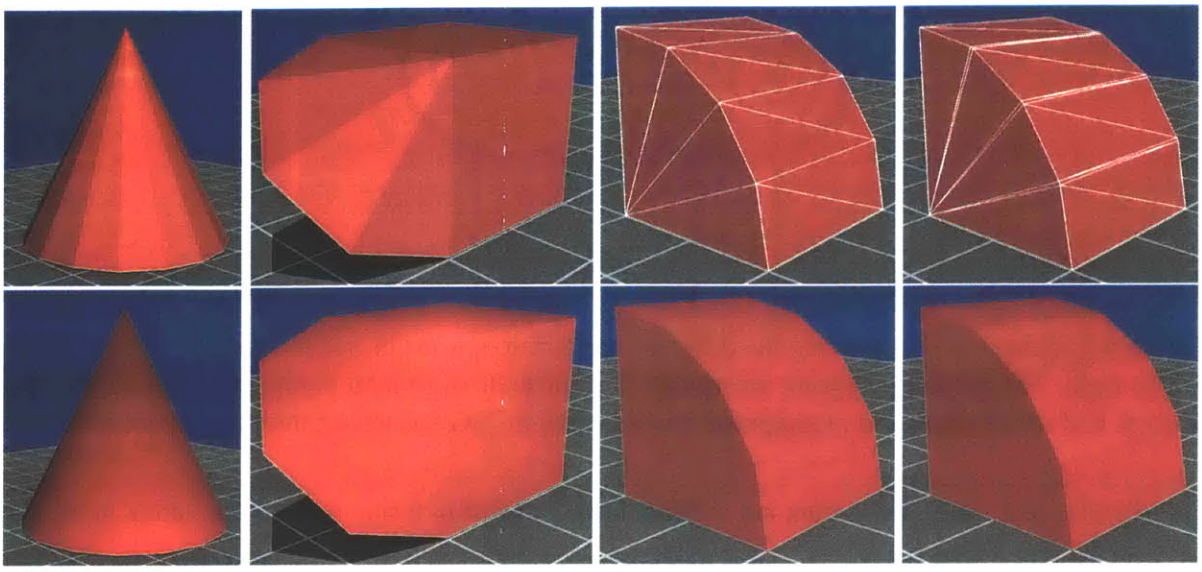


Figure 9.3: These four surface meshes illustrate some of the artifacts that may appear with the smooth normal algorithm. The top row shows the models flat shaded while in the bottom row the models are rendered with interpolated smooth normals at each vertex for each triangle. There is no assignment of normals for which interpolation along the faces of a cone, shown in the left image, is correct. The second pair of images shows a cone-shaped wedge attached to the corner of a cube. Since there is only one sharp edge at this corner, the sharpness is incorrectly ignored. The final two pairs of images show similar shapes, but the addition of sliver triangles in the rightmost model results in subtle facets rather than a smooth curve.

However, the user does not need to wait for an offline rendering to visualize the surface appearance of the model, but can instead explore these textures interactively. The system uses the same shaders, parameters and shader interface as in the offline ray tracer, so that the interactive texture will match the offline texture. Because procedural solid textures can be expensive to evaluate, the results must be cached in texture maps for efficiency.

Usually three-dimensional objects are texture-mapped with a single image that is "wrapped" around the object, often with distortions due to stretch. The texture coordinates for each triangle on the mesh are carefully assigned to grab the right portion of the image. Down-sampled versions of the image, called *mipmaps*, prevent aliasing artifacts when the texture is viewed at low resolutions [Williams 1983].

The texture wrapping method works well for static objects or objects with deformation only. However, when the model is sculpted or fractured, texture must be assigned to the newly exposed faces by evaluating the procedural solid texture. To allow the addition of arbitrary texture patches,

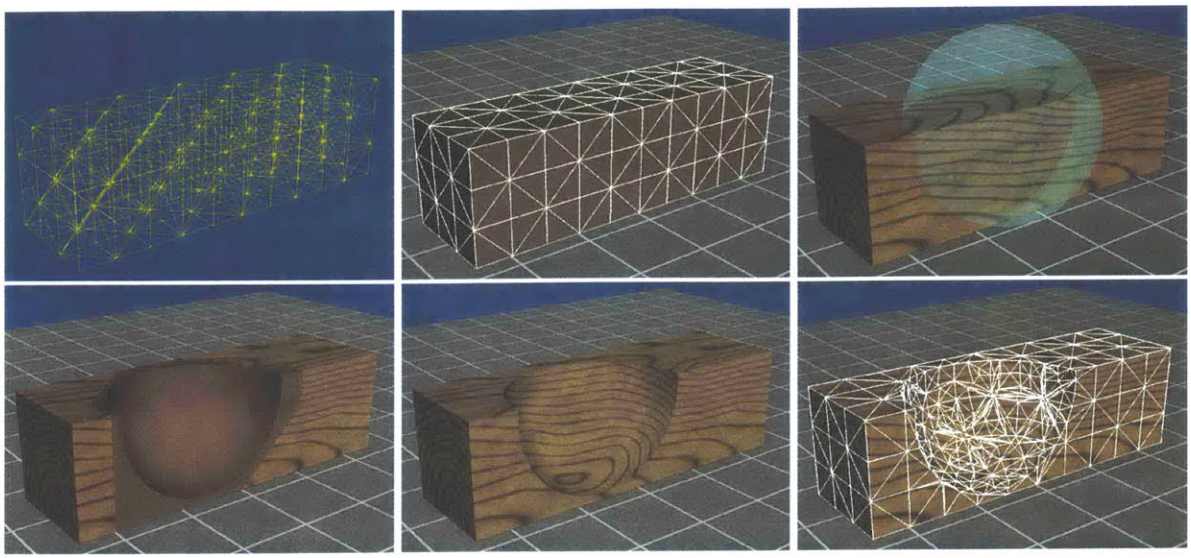


Figure 9.4: A simple rectangular block becomes more convincing when textured with procedural wood grain. When the CSG sphere tool is used to remove a chunk of the model, untextured triangles are created and revealed. After a short delay, the new faces are textured to match the rest of the model.

the texture for each triangle on the exterior interface is stored in a texture chart, similar to that used by Sander et al. [2000] for normal maps. New patches are added to store the texture for these new faces, and the patches for deleted faces are garbage collected for later reuse (see Figure 9.4).

In our implementation, the chart is a 1024x1024 texture that is subdivided into 32 rows of small (16x16) texture triangles, 14 rows of medium (32x32) texture triangles, and 1 row of large (64x64) texture triangles. Overall, the chart stores 5024 texture triangles. The implementation could be extended to include multiple charts (depending on the capabilities and size of texture memory on the graphics card) and to dynamically adapt the distribution of texture triangle sizes to the needs of the current model. In particular, more sophisticated per-triangle texture mapping strategies are possible to minimize texture distortion and efficiently cluster and pack the triangles within a larger rectangular texture [Rocchini et al. 1999, Cignoni et al. 1999].

Each surface triangle is assigned a right isosceles triangular patch in the texture chart based on the length of its longest edge relative to the bounding box of the scene. Triangles with longer edges receive larger patches to ensure that the texture density across the mesh is similar (Figure 9.5). If there are no available texture triangles of the ideal size, another size is substituted. To minimize distortions due to stress, the vertex opposite the longest edge of the triangle is assigned to the right

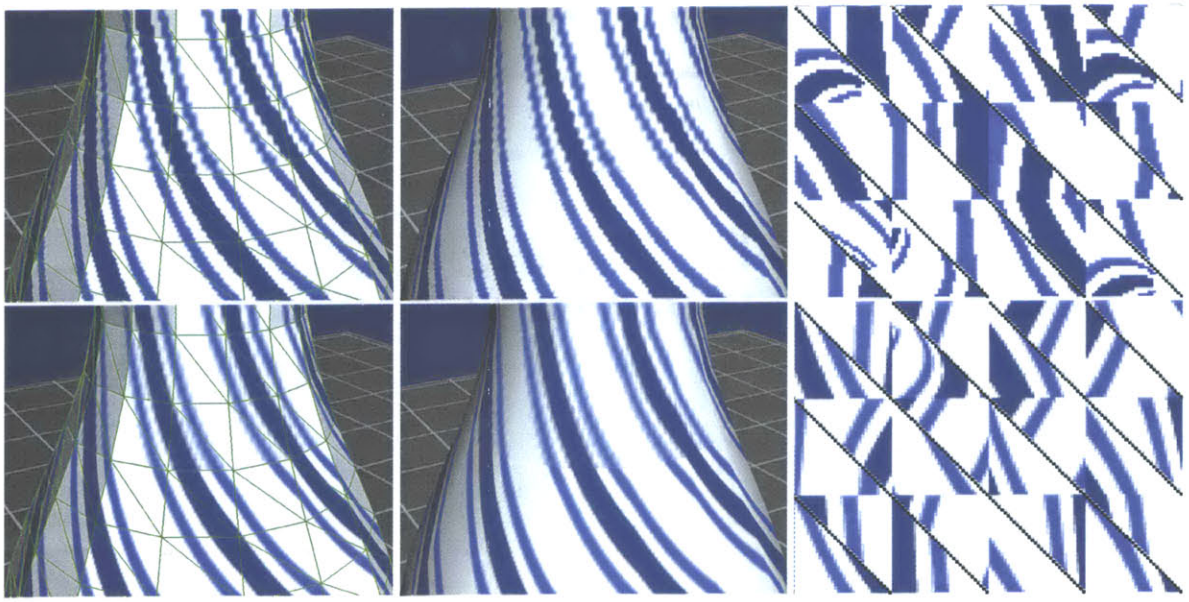


Figure 9.5: In this illustration I show a close-up of the vase decorated with a swirl of stripes. As the neck of the bottle tapers, the triangles become smaller and the seam between texture patches of different sizes is quite visible. The bottom row of images is rendered with anti-aliased textures to de-emphasize the "jaggies" in the low resolution textures. On the right is a small portion of the corresponding texture chart.

angle of the triangular patch. To prevent flipping of the texture orientation during deformation (if the longest edge of the triangle changes) this assignment must be done only once. When a triangle is initialized, the vertices are arranged so that the first indexed vertex is opposite the largest edge.

The texture coordinates within each triangular patch must be assigned with care so that colors from neighboring triangles in the chart, which are probably not neighbors in the mesh, do not leak over the boundaries during interpolation [Cignoni et al. 1999]. Additionally, mipmaps must be disabled since they filter the texture independent of the triangle patch boundaries. If a wider gap is left between neighboring patches in the texture packing, the first one or two mipmap levels can be enabled, without yielding artifacts.

Because the models are subjected to rigid body transformations and deformations, it is important to maintain both model and texture coordinates. The model coordinate indicates the current physical location of the vertex, while the texture coordinate stores the original position of the vertex, which is used to index into a solid texture. If a single position is used for both coordinates the object will probably look fine in static images, but when animated the texture will disturbingly *slide through* the object rather than stretch along with the object, as illustrated in Figure 9.6. A

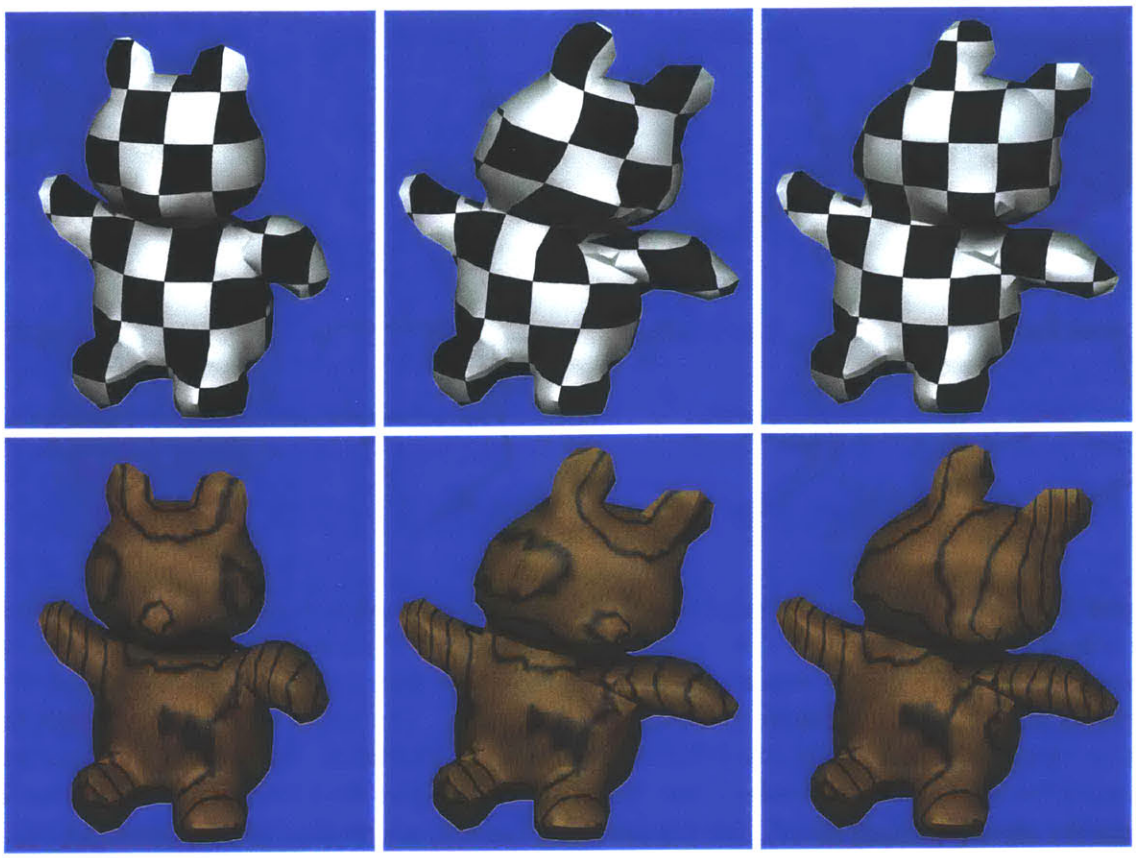


Figure 9.6: This example illustrates the need for two sets of vertex coordinates. The left images show the initial model, the middle images show the texture deforming with the model as expected, and the right images show the object incorrectly sliding through the texture, if the original vertex positions are not used as texture coordinates.

similar solution was used by Wyvill et al. [1987] for the texturing of soft "blobby" objects.

9.4 Tetrahedral Rendering Style

Tetrahedra can be very difficult to visualize in static two-dimensional images. Since tetrahedra are the core element in the modeling system, it is important to have techniques to view the meshes interactively, allowing inspection of the shape and proportion of the individual elements, the internal structures, the density of tetrahedra, and the quality of the boundary surfaces. Figures 9.7 and 9.8 illustrate a few different tetrahedral rendering options, some of which make use of an interactively positioned clipping plane.

One style is to render wireframe tetrahedra — that is, all of the edges in the model. This visu-

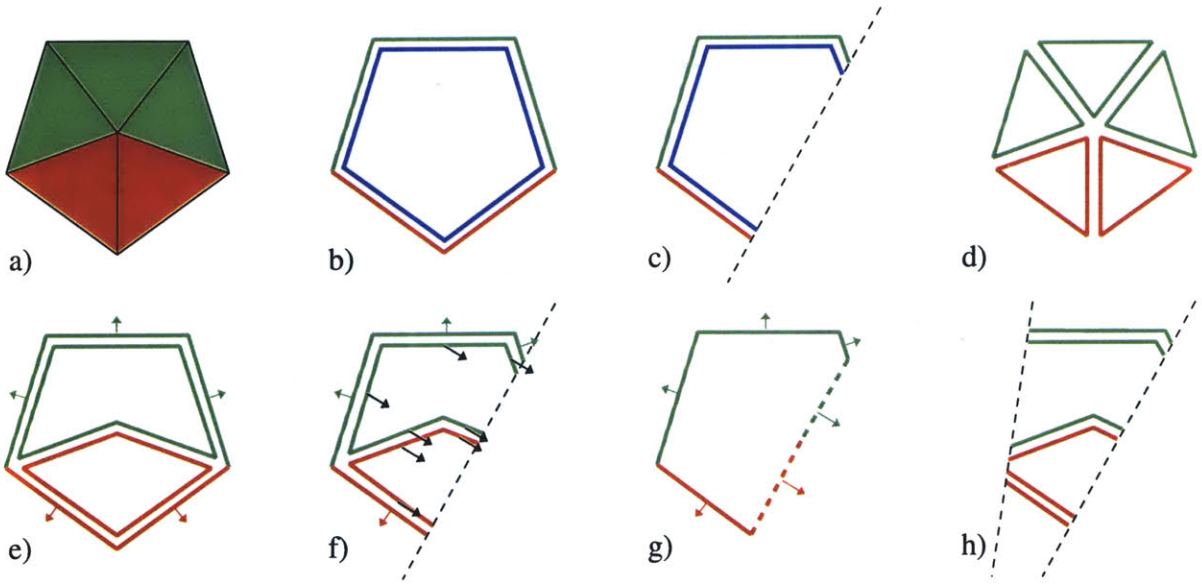


Figure 9.7: Two-dimensional illustrations of the different tetrahedral rendering options applied to the simple model shown in a). Two sided lighting is used to b) color the back faces of surface triangles blue, which can be viewed by c) cutting the object with a clipping plane. The individual elements can be visualized with d) crack-style rendering. The shape of the internal structures can be viewed by rendering e) two-sided internal material interfaces. The internal surfaces are f) assigned the normal of the clipping plane (black arrows), to give the illusion of a g) ghost surface where the object has been sliced. The illusion is broken if h) a second clipping plane is added.

alization can provide the general impression of the density of the mesh, but unfortunately does not reveal much about the shape or material type of the individual elements. I did not experiment with any translucent rendering styles, but this would be a possibility, and is worth exploring. Another option is to draw the tetrahedra with cracks between neighboring faces (Figure 9.7 d). To do this, new vertex positions are computed for each tetrahedron, and four triangles are drawn using these new vertices:

$$
\begin{aligned}
v_{center} &= 0.25 \cdot (v_a + v_b + v_c + v_d) \\
v_a{}' &= 0.8 \cdot v_a + 0.2 \cdot v_{center} \\
v_b{}' &= 0.8 \cdot v_b + 0.2 \cdot v_{center} \\
v_c{}' &= 0.8 \cdot v_c + 0.2 \cdot v_{center} \\
v_d{}' &= 0.8 \cdot v_d + 0.2 \cdot v_{center}
\end{aligned}
$$

With this definition, the width of the crack between tetrahedra automatically scales with the size of the elements, which is visually more appealing than a uniform crack. Tetrahedral meshes can also be exported to triangle meshes with the crack visualization style enabled, and then rendered

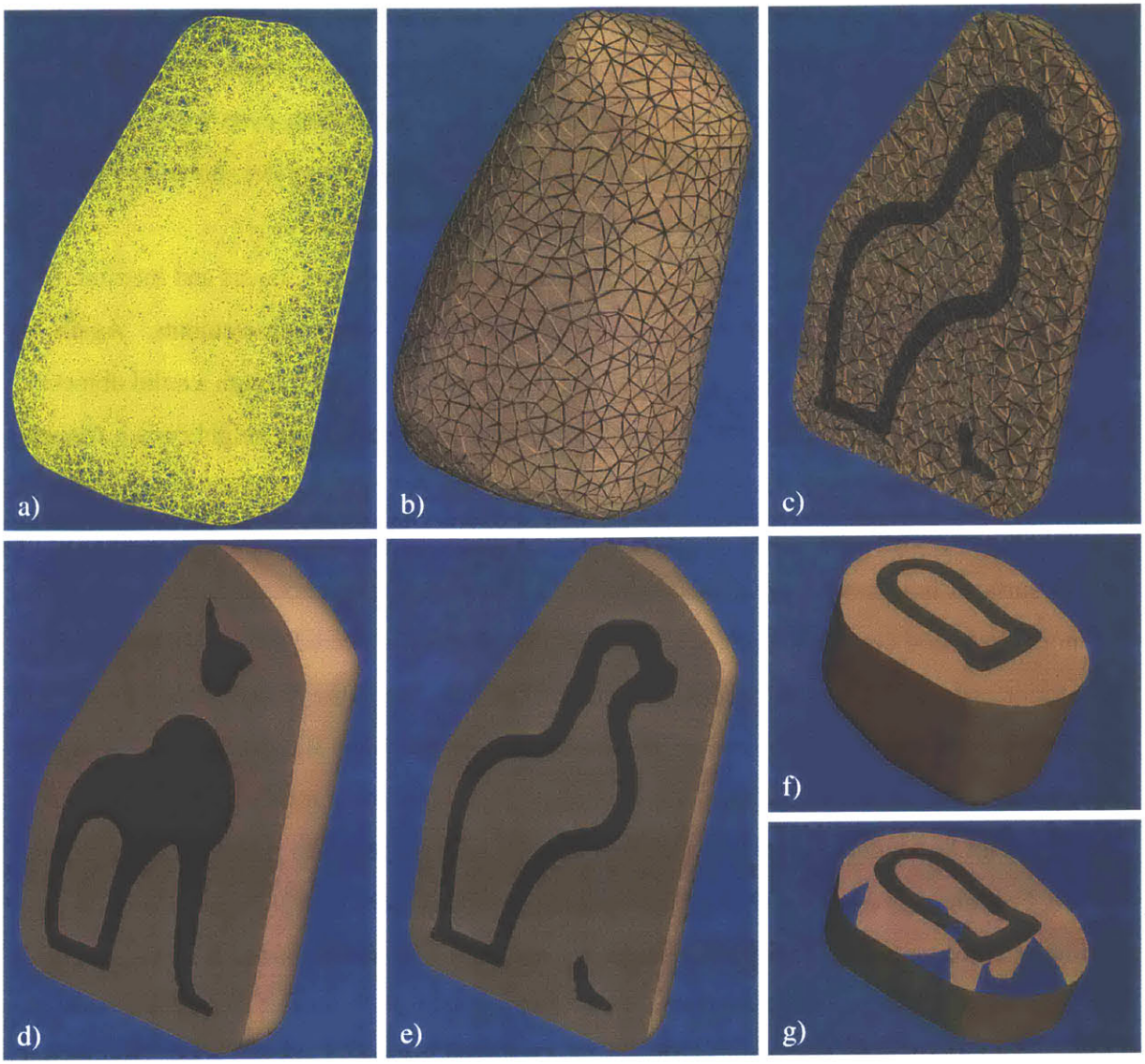


Figure 9.8: A wireframe rendering of the 49,995 tetrahedra cat mesh, shown in a), is not very informative. To better visualize the shape and material of the elements, we can b) render the tetrahedra with cracks between them and c) move a clip plane through the model interactively. To view just the structure, we can render only the air/material and material/material interfaces to create d), e), & f), the illusion of a slicing through a solid material. The illusion is broken if g) a second clip plane is added.

in a conventional ray tracer as seen in Chapter 8.

To visualize the internal structures of models rendered with crack-style tetrahedra, I developed a simple technique to give the illusion of slicing through a solid tetrahedron with the clipping plane. Each tetrahedral face is drawn twice, once facing outward with vertex normals based on

the face normal, and once facing inward with the vertex normals *set to the clipping plane normal.* Back facing polygons are culled so that only one side of each tetrahedral face is visible. Unfortunately, drawing the faces of all tetrahedra is expensive for large models, even when compiled to an OpenGL display list [Woo et al. 1999]. To render the simplified cat model shown in Figure 9.8 with crack-style tetrahedra, nearly 400,000 triangles must be drawn.

However, often the user just wants to visualize the different materials, layers and internal structures of the object, and does not need to view the individual volumetric elements. Again, by carefully specifying the OpenGL vertex normals, the illusion of slicing through a solid object can be created. Each tetrahedral face that is on the air/material interface (its neighbor is a triangle) or on a material/material interface (its neighbor is a tetrahedron with a different material type) is drawn twice (Figure 9.7 e). As the clipping plane is manipulated interactively, the shading for the interior surfaces is adjusted to give the illusion that a correctly-shaded ghost surface appears where the model has been clipped (Figure 9.7 f and g). The shading is altered efficiently by compiling the interior surface triangles to a display list without normals. Given the current clipping plane and camera position, an appropriate normal is assigned before executing the display list. This technique yields a very convincing *interactive* illusion of slicing through a solid object. Unfortunately, texture cannot be rendered with this approach and the user is limited to a single planar clipping surface, because only a hollow shell of the volume is drawn (Figure 9.7 h).

Figure 9.8 d, e, f, and g shows the cat model rendered with double-sided internal interfaces. The model has 4,064 exterior triangles that are rendered with lighting and smooth normals. An additional 14,628 tetrahedral faces on the internal material interfaces are rendered. The responsiveness of the system as the user adjusts the clipping plane, compared to that of the crack-style display mode, is much improved, as expected.

9.5 Interactive Tool Interface

One priority in designing the system was developing a simple and consistent user interface. First the user chooses a volumetric model, which has been designed, constructed, labeled with material properties, and simplified offline (see earlier chapters). A text display in the lower left of the screen reminds the user of the model's size (largest dimension), weight, and materials. The user can check

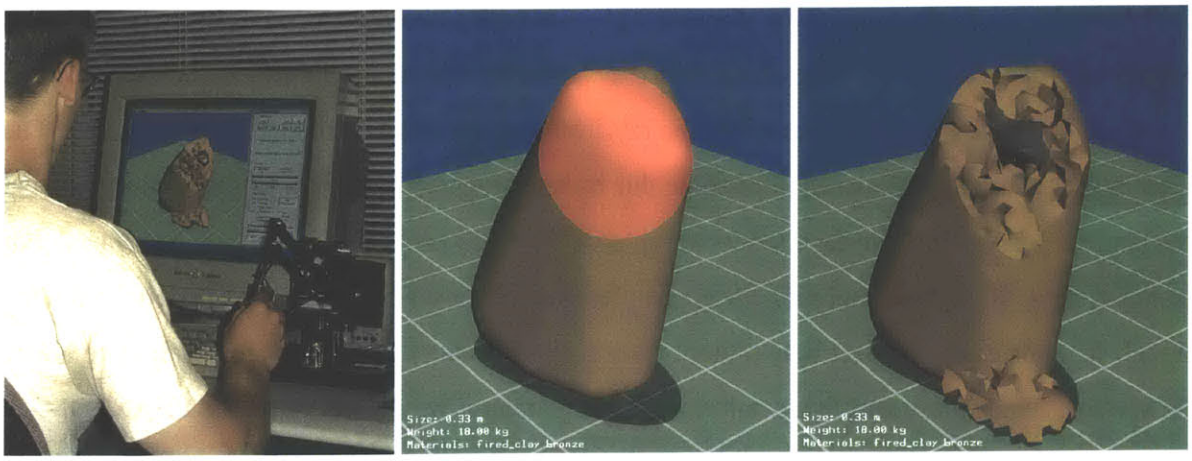


Figure 9.9: The photo on the left shows a user interacting with a volumetric model with the *PHANTOM*^{TM} three-dimensional force feedback input device [SensAble Technologies]. The middle image shows a visualization of the current Chisel tool position, which highlights the region of the model that will be affected by the impact force of the tool. The result of the tool action is shown in the right image.

the plausibility of the model before performing sculpting or simulation operations. For example, the user has a chance to correct the scale of the object before running a deformation simulation on a ten-foot-tall, one-ton chocolate bunny.

Our system can be used to compare a number of different physical simulations — thus, the user is also asked to select an appropriate type for the sculpting to follow, so that the necessary simulation data structures can be initialized. The choices currently implemented include rigid body, rigid body with deformation and fracture [Müller et al. 2001], linear FEM, non-linear FEM, warped-stiffness FEM [Müller et al. 2002], or none. The user can also enable collision detection and/or dynamic behavior, and adjust the magnitude of gravity. If desired, the user can specify a high-resolution surface mesh that will be animated by interpolating the results of a simulation computed on the low-resolution tetrahedral mesh.

The user selects a tool from the palette, and adjusts various standard tool parameters, discussed in Chapter 5. The tools currently available include *CSG Sphere* for subtraction, *Lift* for dropping rigid bodies, *Grab* for pulling and stretching deformable objects, *Hammer* for impact forces, *Peel* for performing spring-mass simulation on thick triangles, and *Wash* and *Erode* for directing various particle flows. The current tool position and size is visualized in the display window, and a single key press toggles between inspection (rotation, translation, and scaling) and modification actions.

The problem of selecting a three-dimensional point with a two-dimensional device (a mouse) is common to most interactive three-dimensional systems. For the mouse interface, the *contact point* is defined on the first surface pierced by the ray from the eye point through the image plane. The depth buffer is used to efficiently determine this intersection and the center of the tool is placed at this contact point. This interface works acceptably for many tools, but is limiting for others. For example, the CSG Sphere tool will always be placed to remove a hemisphere of material from the model. In some cases the user may rather take a shallower cut, with the tool center floating above the surface. For a more satisfying solution to this problem, we incorporated the PHANTOM^{TM} three-dimensional input device [SensAble Technologies] into our system (shown in Figure 9.9).

Another general problem for interactive modeling systems stems from the limited dimensionality of standard graphics displays. For example, when using the FEM lift tool to "drop" objects, it can be difficult to interpret the height of an object above the ground plane. To assist the user's three-dimensional understanding of the scene, we render a simple shadow cast by the object(s) to the ground using a technique described by Blinn [1996]. All triangles in the model are rendered a second time, projected to the ground plane and drawn with 50% transparency. Examples of this effect can be seen throughout the chapter.

Finally, the OpenGL stencil buffer is used to create tool position and size visualizations. The CSG Sphere is shown in Figure 9.4 and the Hammer tool impact region is shown in Figure 9.9.

9.6 Chapter Summary

In this chapter I have presented techniques for the interactive visualization of three-dimensional volumetric models, specifically tetrahedral meshes. These methods were an integral part of the debugging process during the development of the modeling system described in this dissertation. Visualization and interaction is also a necessary component of the design process, to provide timely feedback about internal structures and material properties, so that they can be adjusted in future iterations.

Chapter 10

Results

Throughout this dissertation I have presented both simple and complex examples of objects that were constructed and manipulated with my procedural solid modeling system. In this chapter I present several additional examples. Each example is motivated with a history of environmental effects and sample real-world imagery. I also include several smaller examples to demonstrate the interactive portion of our system.

10.1 Weathered Gargoyle

The first example is a weathered gargoyle statue mounted on the exterior of a building, shown in Figure 10.1. Gargoyles are subjected to interesting flow patterns because they were originally used as decorative downspouts to direct rainwater away from exterior building facades. Long-term exposure causes a variety of effects on these exterior architectural details including discoloration, weakening, erosion, biological growth, and fracture due to the freeze/thaw cycle. In Chapter 5 I described a number of different weathering effects that were applied to this particular model. Two inspirational real-world examples for this object are shown in Figure 10.2.

This model was created by scanning a replica of a gargoyle statue that would have been mounted as shown in the pictures, and used to carry water from the gutters. Water would flow along the back of the creature, through a hole in the back of its head and out of its mouth. However, while the wings do form a channel for the water to flow horizontally, the head of this gargoyle replica is solid. I could have constructed a more faithful representation of this scene by first performing the necessary CSG sculpting actions to create an exit for the water. This modification

Figure 10.1: This gargoyle model was inspired by the photographs shown in Figure 10.2. Details of the weathering operations performed on this model are given in Chapter 5. The Gothic cathedral elements in the background were modeled by Stephen Duck.

Figure 10.2: The photograph on the left from Crist [2001] shows a gargoyle downspout directing rainwater away from the building. In the right image, from Benton [1997], staining and lichen growth from years of exposure are apparent on the gargoyle and the other architectural elements.

would result in different flow patterns and an opportunity for a visually stunning animation. Additionally, the weathering operations were only applied to the gargoyle; but clearly the entire scene could also benefit from the added realism of these effects.

The modifications were initially sketched on a low resolution version of the model containing approximately 50,000 tetrahedra. The high resolution mesh used in the offline simulations consisted of approximately 500,000 tetrahedra. Once the design was finalized, processing for all

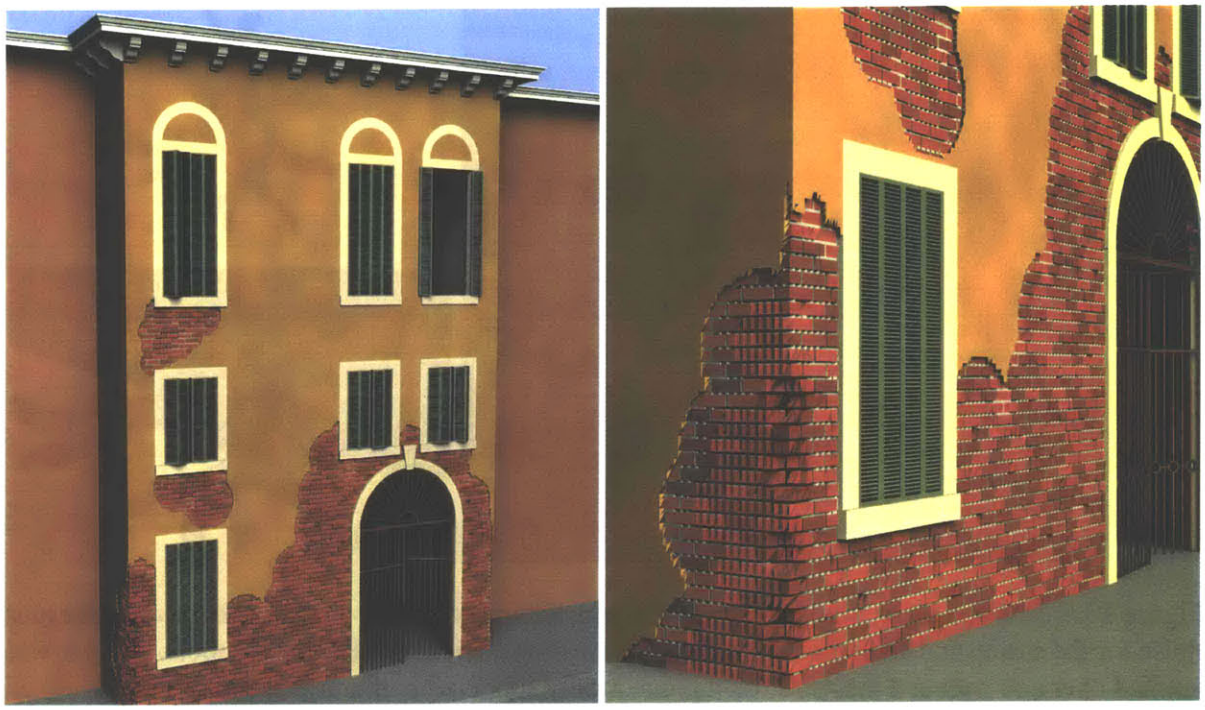


Figure 10.3: The crumbling plaster and brick surface of this model was motivated by Venetian facades similar to those shown in Figure 1.2. The close-up on the right reveals the regular tetrahedralization of this model, since it was sculpted with an early version of the system that lacked support for tetrahedral refinement. The gate, cornice and shutters were modeled by Stephen Duck and added during the final rendering.

stages of this model (initialization, simplification and simulation) was completed overnight.

10.2 Venetian Facade

Another example of weathering I wanted to simulate was the crumbling plaster and brick shown in Figure 1.2. To create the synthetic model shown in Figure 10.3, a custom distance field is created from a simple CAD model. Two layers of material are applied to the model — a thin layer of colored plaster covering a thicker procedural layer of brick and mortar. The model was not simplified from its initial grid tetrahedralization, simply because the simplification code was not complete when this model was designed and executed. A simple ellipsoidal CSG subtraction tool is used to loosely copy the weathering pattern from a photograph of a similar building. The materials react differently based on their proximity to the center of the tool: plaster is removed most easily, followed by mortar, and finally brick.

Figure 10.4: Inspired by examples in the real world, a tree model was created using a cylinder plus noise for the stump and an extruded hand-drawn two-dimensional image for the roots. A layer was applied over and between the roots, then procedurally subdivided into bricks and dirt.

This model contains many possibilities for future work, the most obvious being addition of the thick triangles described in Section 6.2.4, to represent a layer of peeling and blistering paint. The model contains approximately one million tetrahedra, so interactive sculpting was slow and tedious, taking roughly an hour to complete. Although simplification of the initial mesh would help, the general problem stems from the large scale of the model relative to the fine level of detail desired for the output. Delayed-element instantiation, view-dependent refinement, multi-resolution modeling or a hybrid volume/image-based data structure could be used to solve this problem in the future and more satisfactorily capture the surface details necessary for close-up renderings.

10.3 Displaced Brick Paving

In the next example, I model a tree in an urban setting surrounded by brick paving. As the tree roots grow and expand, the bricks surrounding the tree are shifted and pushed upward, but maintain their shape. Figure 10.4 shows a photograph of a similar tree in the real world. The materials in this model have different behaviors and thus the boundaries between the materials must be represented within the tetrahedral model to allow correct simulation. A voxel representation would be inadequate.

The tree model is constructed by combining distance fields for the trunk and root shapes. The trunk is created from an implicit function for a cylinder that flares at the base, plus turbulence to imitate the rough surface of bark. The roots are modeled with a custom distance field extruded from a two-dimensional user sketch, shown in Figure 10.4. More visually-compelling root systems can be created with procedural techniques such as L-systems [Prusinkiewicz 1986], but for this application the extrusion approach is simple and gives direct control of root shape and placement to the user. A thick layer is applied over and between the roots using the precedence operator described in Chapter 4. The layer is procedurally divided into bricks and dirt to form the paving pattern. Here is the script used to produce the initial model:

```
URBAN_TREE = precedence {
  volume_1 = volume {
    distance_field = union {
      distance_field_1 = TRUNK
      distance_field_2 = 2D_EXTRUDE {
        file = roots.ppm } }
    interior_layers = {
      layer {
        material = TREE
        thickness = fill } } }
  volume_2 = volume {
    distance_field = GROUND_PLANE
    interior_layers = {
      layer {
        material = BRICK_PAVING
        thickness = 0.075 }
      layer {
        material = DIRT
        thickness = 1.00 } } } }
```

After simplification, the model contains approximately 200,000 tetrahedra. To displace the brick paving around the tree, I created a new tool to translate upward the vertices of all tree tetrahedra. An offline finite element simulation solves for the static equilibrium positions of the remaining vertices. The results are shown in Figure 10.5. The bricks maintain their rectilinear shape, because the brick material has a large value for Young's Modulus, the elasticity parameter; the dirt between and beneath the bricks deforms easily because of its relatively smaller value. Appropriate values for these materials can be obtained from standard references [Anderson 1989]. The simulation performed in this example took less than an hour to finish. The most time-consuming aspect of

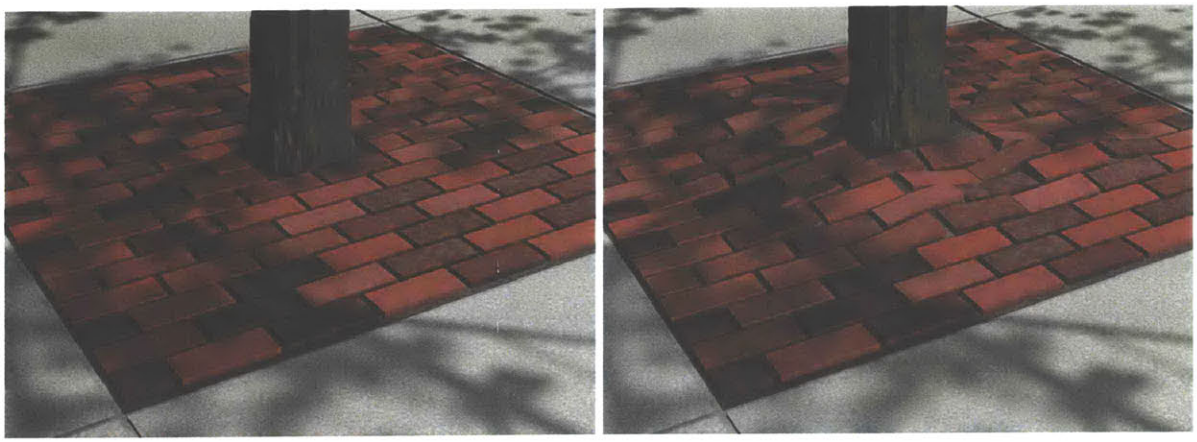


Figure 10.5: To simulate tree growth, the vertices of all tree tetrahedra are translated upward which deforms the dirt and bricks around the roots. A simple tree model was placed above the stump outside of the rendering window to cast shadows of branches and leaves and complete the scene.

this example was arranging the scenery for the final rendering.

As with all synthetic environments, there are many additional features and simulations that could have been incorporated: the raised brick corners are more exposed to pedestrian traffic and will become worn and broken; the dirt will wash away from between the raised bricks; moss will grow between the bricks on the shady side of the tree; the bark will peel; cracks will form in the concrete paving; etc. Furthermore, although this simple physically-inspired arrangement produced the intended results, a more physically-correct representation would model root expansion instead of lifted vertices. Also, most brick sidewalks are packed more tightly with a minimal seam of sand or dirt, and the bricks are held together by friction, rather than the deformable clay-like material in this model. Unfortunately, it would be difficult to correctly model this scene since contact forces and collisions between many rigid bodies are a challenge to model accurately. All of these features are possible within the context of the language, and would only require small extensions to the infrastructure and/or optimizations of the simulation engine.

The simulation computed in this example is textbook, but the results are exciting because they demonstrate the interaction of different materials arranged in a complex pattern. Constructing interesting tetrahedral models for finite element systems has long been a challenge, but procedural solid modeling offers new possibilities.

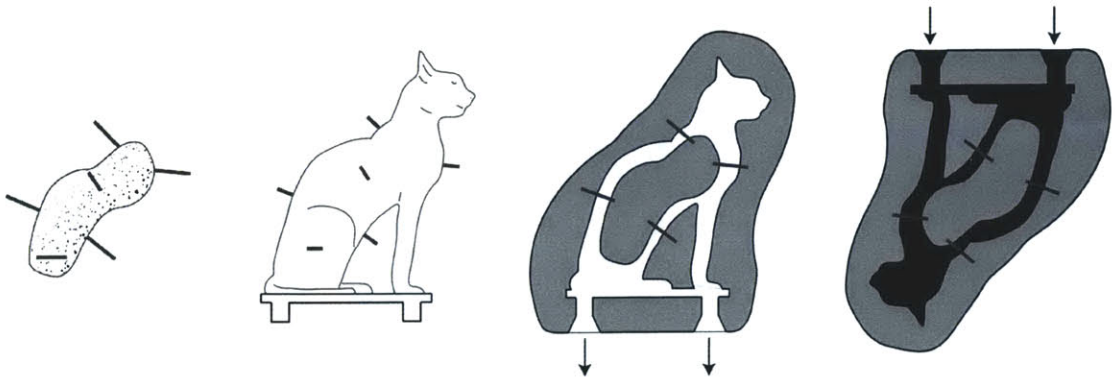


Figure 10.6: This illustration, redrawn from Hodges [1970], presents the first four steps of the lost wax casting method for creating bronze statues. The steps are: the clay core, the wax statue formed over the core, covered with clay and heated to remove the wax, and finally, filled with molten bronze. The bronze pegs embedded in the clay core hold it in position when the wax is removed.

10.4 Lost Wax Casting of an Egyptian Cat Statue

In the final complex example, I explore the multi-layered object that is constructed at an intermediate stage during the lost wax casting method for creating bronze statues. This is an ideal example with which to demonstrate the system, as layer boundaries of this object are derived from the shape of the final sculpture, but have very different shapes.

An overview of the lost wax casting method is shown in Figures 10.6 and 10.7. A roughly-shaped clay core is covered with malleable wax, in which the shape and details of the final sculpture are formed. When the wax sculpture is finished, a thick layer of clay is spread over the wax. The model is slowly heated to allow the wax to drip from the clay mold, and then the mold is fired in a kiln. Molten bronze is poured into the clay mold, and finally, when cool, the brittle clay is broken away to reveal the new bronze statue. After a thorough cleaning, the sculpture is treated to achieve a particular finish patina.

I constructed a model of this multi-layered object just before the outer layer is broken away, and then simulated the removal of the clay and polishing of the raw bronze. The original cat surface has sharp edges and areas of high curvature, particularly near the ears, but the outer clay layer does not contain such detail. In the physical process, the artist applies a thicker layer of clay to the concave portions of the model. To model this process, I used the convex hull of the original surface as a second mesh. Below is the script used to generate the initial model:

Figure 10.7: These photographs of the lost wax casting process from Langland [1999] follow the process after the stages illustrated in Figure 10.6. The outer layer of clay is broken away with a hammer to reveal the sculpture. After polishing and sandblasting, the raw metal is bright and shiny. Different chemicals are applied to the sculpture to achieve a particular patina and the model is sealed with a coat of wax.

```
BRONZE_CAT = precedence {
  volume_1 = volume {
    distance_field = surface_mesh {
      file = cat.obj }
    interior_layers = {
      layer {
        material = BRONZE
        thickness = 1 }
      layer {
        material = FIRED_CLAY
        thickness = fill } } }
  volume_2 = volume {
    distance_field = surface_mesh {
      file = cat_hull.obj }
    interior_layers = {
      layer {
        material = FIRED_CLAY
        thickness = fill } }
    exterior_layers = {
      layer {
        material = FIRED_CLAY
        thickness = 2.5 } } } }
```

The model is interactively sculpted, first with the hammer tool described in Chapter 5, to break away the outer layer of clay. By specifying that only the fired clay is affected, fractures are prevented from occurring within the bronze material. This is done by assigning all bronze tetrahedra to the same side for each fracture operation (see Section 6.3.2 and Figure 6.9). The tool is used

repeatedly on different portions of the model. A sample call is given below:

```
HAMMER {
  model = BRONZE_CAT
  position = { 1.08 0.79 0.29 }
  orientation = { -0.32 -0.26 -0.91 }
  affects = { FIRED_CLAY } }
```

Next, the statue is polished with a sphere-shaped tool that cleans and brightens the bronze material. This tool performs a CSG subtraction operation to remove clay left on or around the model (Section 6.3.3). The polish tool also increases the shininess of nearby tetrahedra, by blending with the SHINY_BRONZE material.

```
void POLISH(Model *model,
            Vec3f position = Vec3f(0,0,0),
            float size = 1) {
  AppliedArea *a = Sphere(position, size);
  List<Material*> affects(Lookup("FIRED_CLAY"));
  CSG_Subtract(model,a,affects);
  List<Tetra*> lst;
  model->CollectTetras(lst,position,size);
  for (int i = 0; i < lst.numElements(); i++) {
    Tetra *t = lst.getElement(i);
    t->BlendMaterial(Lookup("SHINY_BRONZE"),
                     position,size); } }
```

The low-resolution cat model has approximately 50,000 tetrahedra. Collision detection and tetrahedral refinement were disabled for the interactive session to improve responsiveness. Visualizations of the internal structure of this model are shown in Chapter 9 along with images of the sculpting process. The tool operations were automatically logged to a script file and then replayed on a model with 300,000 tetrahedra. The interactive session was completed in less than half an hour. The high-resolution simulation finished in a few hours. A sequence of ray-traced frames from the offline simulation are shown in Figure 10.8.

The ears of the cat statue are very thin with sharp creases along the edges. A high-resolution distance field grid is required to mesh this surface without introducing holes. Even with sufficient samples to guarantee the correct topology, the ears have nicks along the edges due to the sawtooth artifact discussed in Section 7.4.5. With a very high resolution initial grid, I was able to reduce these artifacts to the scale of the noise in the original scanned model. The octree distance field grid,

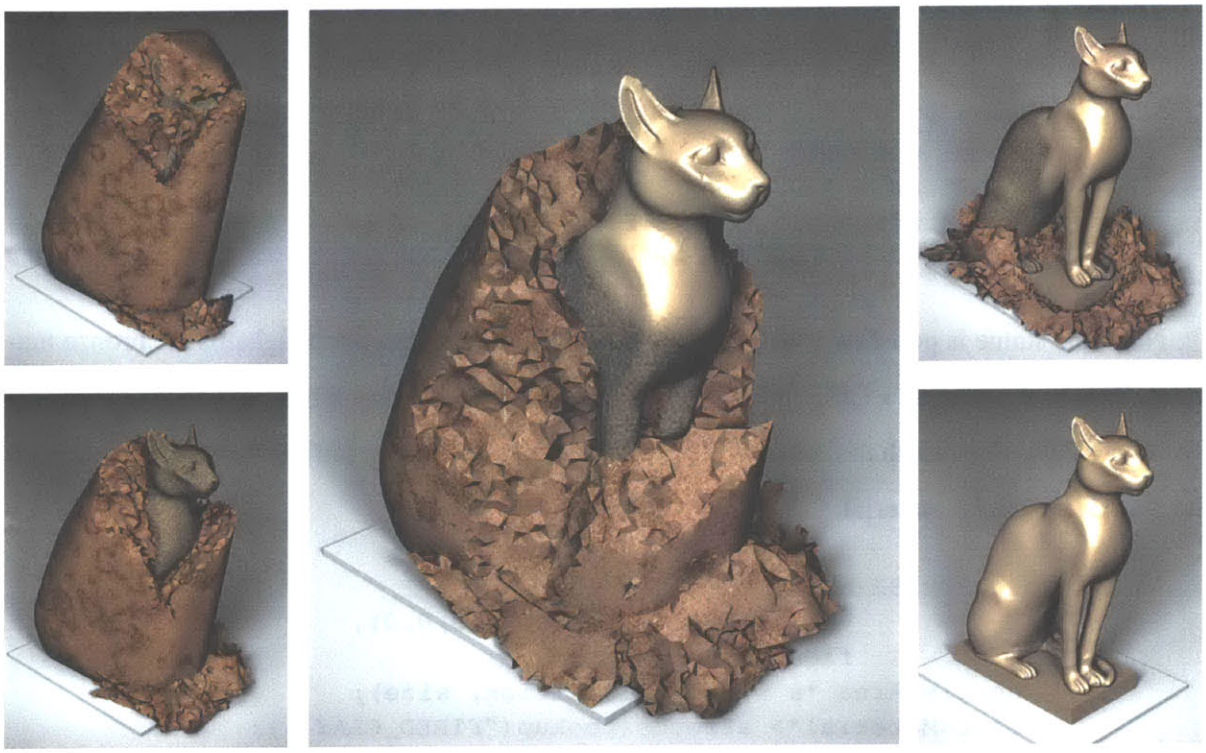


Figure 10.8: A sequence of images from the bronze statue simulation. The outer layer of fired clay is broken away using a hammer tool. A polish tool is used to clean and shine the model.

described in Section 8.2, is critical in this example to ensure that the number of elements initially created (prior to simplification) is reasonable.

The fractured clay surfaces betray the polygonal nature of the tetrahedral representation. A more highly refined mesh could better represent these details in close-up views; but unfortunately, the sole FEM module implemented in our system has been optimized for interactive simulations with smaller models and is not capable of processing such models. A better solution would be to develop a hybrid data structure to allow these details to be represented with a different technique, such as displacement maps.

There is no guarantee that the simulation performed on the low resolution model is a good predictor of the same operations performed on the high resolution model. In particular for this example, since collision detection and element refinement are enabled only for the offline simulation, the results are difficult to precisely control. An interesting area for future work would be to apply a simulation computed on a low resolution model to models of varying levels of detail.

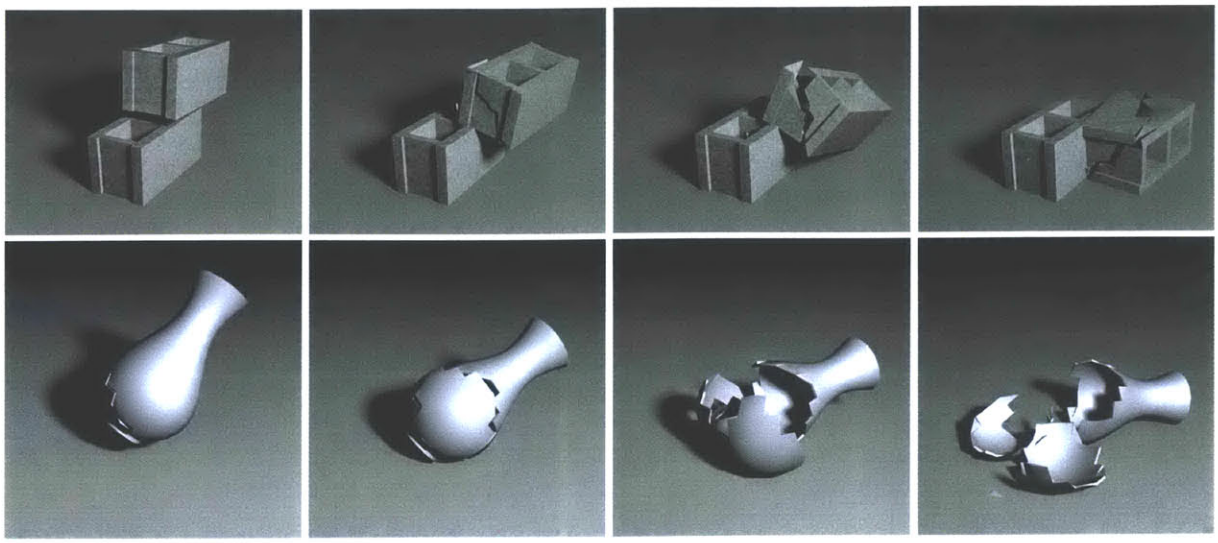


Figure 10.9: The models in these two sequences were simulated interactively with collision detection and fracture, and then ray-traced offline. The top row shows two concrete blocks, one dropped on top of the other, which contain 412 tetrahedra each. The bottom row shows a ceramic vase with 1,440 tetrahedra.

10.5 Interactive Fracture and Deformation Simulations

The interactive sculpting and simulation system was a joint effort with Matthias Müller and Rob Jagnow. The previous examples were all targeted for offline processing after initial online "sketches" were designed on low-resolution models. In this section I show examples of real-time manipulation executed in our system, which appeared in Müller et al. [2001] and Müller et al. [2002]. Interactive simulation is important in two major fields of graphics: in games, where stability is foremost — for example, the game is not allowed to crash and must recover gracefully from any inconsistencies; and in movies, where artists want control over all aspects of the simulation and need quick feedback to allow iterative design. While realism is still necessary, both of these applications will trade a fair amount of physical accuracy to maintain interactive updates.

Figure 10.9 shows two examples of interactive fracture. The user can design an animation by iteratively adjusting the initial conditions and material properties. The concrete blocks were created in the language by carefully selecting a grid spacing to avoid sawtooth artifacts (Section 7.4.5). I attempted to create a tetrahedral mesh of the vase with my infrastructure, but the thinness of its shell required a very high resolution, and it was much easier to simply generate the tetrahedral mesh as a surface of revolution.

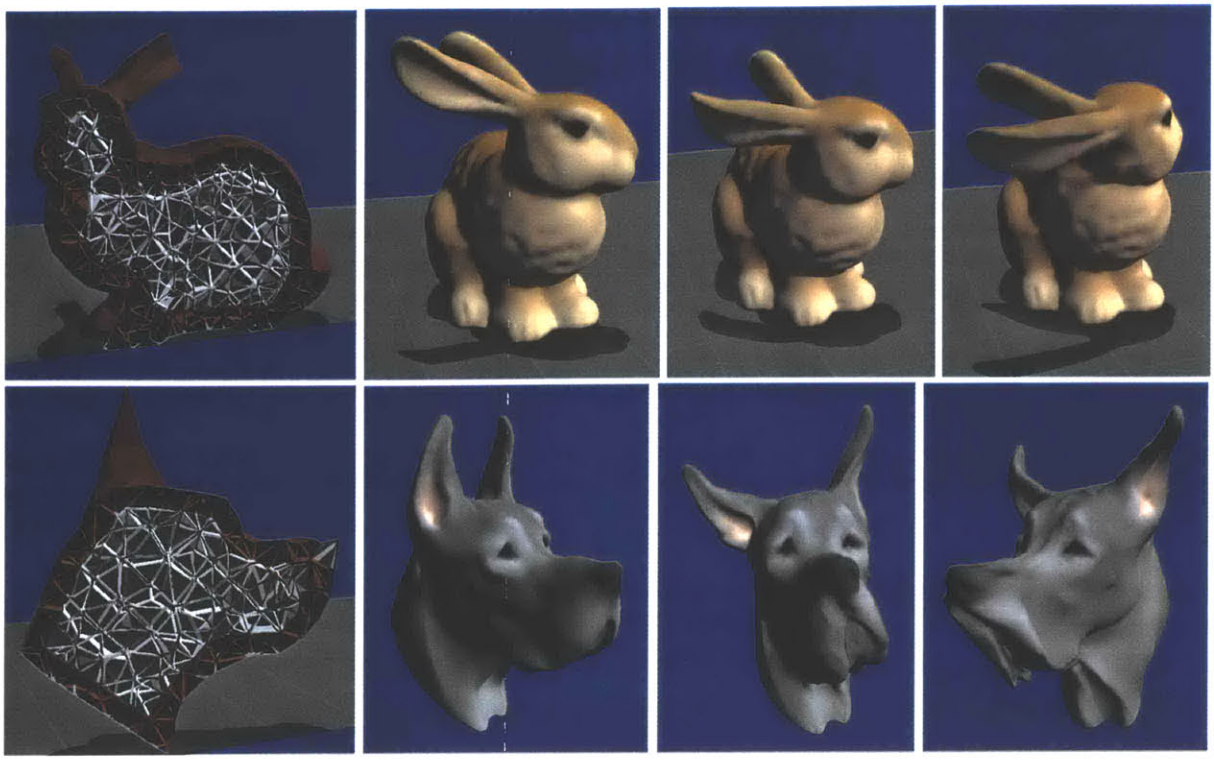


Figure 10.10: These examples demonstrate the deformation of multiple materials within a single object. Both models consist of a rigid bone core with a soft deformable skin covering.

Figure 10.10 shows two multi-layered examples that were quickly constructed from scanned meshes using the layered volume component of the language. These models would have been much more time-consuming to create without the language. The script used to initialize and simplify the dog model is shown below.

```
GREAT_DANE = volume {
  distance_field = surface_mesh {
    file = dane.obj }
  interior_layers = {
    layer {
      material = SKIN
      thickness = 0.02 }
    layer {
      material = BONE
      thickness = fill } } }

simplify {
  model = GREAT_DANE
  desired_tetra_count = 2000 }
```

A special interactive tool was created to rigidly move the bone core in the head of each model, while an FEM simulation computed the dynamic response for the deformable skin, resulting in wiggly skin and floppy ears. The material properties of the dog ears and nose were stiffened for a more realistic appearance.

The bunny consists of 4,995 tetrahedra. The simulation animates 851 tetrahedra for the ears and the skin of the head. The dog model contains 1,997 tetrahedra, 1244 of which are skin tetrahedra that are animated by the simulation. The original scanned surface geometry, textured by Rob Jagnow, is animated from the deformation computed on the low-resolution tetrahedral model.

10.6 Chapter Summary

In this chapter I have presented a range of interesting examples to demonstrate the procedural modeling components discussed throughout this dissertation. In the next chapter I will further discuss the benefits and limitations of the modeling language and its implementation, and summarize the thesis.

Chapter 11

Conclusions and Future Work

In this dissertation I presented a system for the procedural design of layered solid models and the details of an implementation for both interactive and offline manipulation of these objects. In Section 11.1, I further analyze the results presented in the previous chapter and discuss the benefits and limitations of my work. In Section 11.2, I outline some of the possible directions for future work. And finally, a summary of the thesis is given in Section 11.3.

11.1 Discussion

My research has spanned a number of subfields of computer graphics and mesh processing. In addition to studying these areas and creating applications that draw on these diverse fields, I have developed solutions to a number of important problems. I have demonstrated the features of the solid model authoring language in many examples throughout this document. Several other users have used the system to construct models for use in their own research, and I hope to make the system available to a larger audience. Below I discuss the benefits and limitations of the modeling language and its implementation.

11.1.1 System Benefits

Procedural modeling approaches can create fantastic geometry [Musgrave et al. 1989, Ebert et al. 1998], but it is difficult to make the model look like a particular object — for example, it is difficult to create a fractal mountain that looks just like Mount Everest. At the opposite extreme, manually

specifying every detail of a complex model can be impossibly tedious and error-prone. Either design strategy may be used in the language, or a combination of the two. The language allows specification of the same model in a number of different ways. For example, an experienced procedural modeling programmer could design a detailed solid object with the language using only implicit surfaces and procedural definitions for the decomposition of materials. Alternatively, an artist with access to a three-dimensional scanner and experience manipulating surface meshes would use the language to compose a similar scene from multiple scanned input shapes and only use available procedural components (such as `STRIPED_CHOCOLATE` or `BRICK_AND_MORTAR`) as black boxes, without understanding their details.

Specifically, the language introduces the following new components for authoring solid models: the application of layers of material to an input shape; no minimum thickness requirement for material layers; procedural control over layer thickness; multiple materials in the same model; the use of multiple input shapes with several options for specifying the materials in the intersecting regions; the use of high-resolution scanned meshes; multiple simulations on a model; and the ability to develop new simulation operators (Chapters 4 and 5).

The most important advantage gained from using a procedural approach to solid modeling is the ability to iteratively design complex objects. A short script in the language, along with the relevant input surface meshes, is a complete description of the object that can revised by the user and reprocessed to achieve a particular result. The procedural components used to create one object can be reused or modified to create new models. The language and core infrastructure are implemented in C++, a common language for many other graphics and simulation tasks, allowing their use as modules of the modeling system.

The tetrahedralization strategy described in Chapter 7 is robust and efficient and can produce complex models with more than one million tetrahedra. The implementation can handle minor self-intersections of the input surface (Section 7.2.4), unlike the Delaunay or Advancing Layers methods. The simplification and mesh improvement algorithm (Chapter 8) reliably reduces these large meshes to allow stable interactive manipulation with simulations such as the Finite Element Method (FEM). The construction of complex, layered tetrahedral models suitable for simulation was previously a challenging task. I formulated the simplification algorithm so that it requires just one input parameter, the desired final tetrahedral count, and therefore it is very easy to use.

In developing the modeling language and core implementation, I created an interactive infrastructure to visualize the internal structures of tetrahedral meshes (Section 9.4), which facilitates iterative design. A procedural definition for solid materials can also be used to texture map the model during interactive modifications (Section 9.3).

11.1.2 System Limitations

The current system implementation does have a number of limitations, some of which are addressed as areas of future work.

One unfortunate limitation of procedural techniques is that it takes skill and experience to write concise and expressive procedural definitions for components such as solid textures, isosurface velocity, material decomposition and simulation effects. Guidelines for designing procedural textures are given in Apodaca and Gritz [1999], but most often making effective use of procedural modeling techniques simply takes practice and some extra insights about the material or effect to be reproduced. For these reasons, a sample-based approach may be the ideal interface for a novice modeler. For example, by using texture synthesis (Section 11.2.1), a small sample of the surface appearance can be used to generate a larger visually-indistinguishable patch of texture.

I chose to implement a tetrahedral mesh representation for the many reasons described in Chapter 6. However, as mentioned, the representation is not ideal for rendering or simulating many real-world objects and effects such as liquid, gas, smoke, and furry or fuzzy objects. Additionally, the implementation costs of constructing and maintaining a tetrahedral mesh suggest that voxel or octree-based strategies may be a better choice when simulations that deform or fracture the model are not utilized. I have taken care to keep the functionality of the language independent of the underlying volumetric data structure, so that future extensions may take advantage of alternate representations.

There are several limitations to the structured mesh generation technique I chose to implement (Chapter 7). The same grid alignment and spacing must be used for all distance fields within a volumetric model. However, in some cases it may be advantageous to merge models that have been tetrahedralized on different grids. Furthermore, the structured tetrahedralization method does not match the original surface, which can be a problem if the topology is not accurately captured, or the output mesh contains artifacts that may be mistaken for surface detail.

The system contains sculpting and simulation modules developed specifically for this project, including CSG manipulation, interactive FEM, and basic particle system operation. However, I have not attempted to integrate any stand-alone simulation implementations. Unforeseen difficulties may arise when such modules are added, requiring a restructuring of the operator language (Chapter 5) to handle other types of simulation.

I describe a strategy for interactively sketching simulations on a low-resolution mesh that can be replayed offline on a higher-resolution mesh (Section 5.4). Unfortunately, the results from the low-resolution simulation are not guaranteed to match even the coarse details of the high-resolution simulation. This is a general problem with multi-resolution modeling that must be addressed as large-scale simulations are developed.

11.2 Future Work

There are many possible avenues for future work. One extension that I am currently investigating is intricate solid material variations that can be used in rendering and physical simulations. In this section I also describe several additional areas for possible extensions to the system, including: improvements to the tetrahedralization and simplification algorithms, extensions to the language, implementation of volume addition, new simulation operators, establishing a library of the procedural components of the language, alternatives to tetrahedral meshes including hybrid data structures, and strategies for modeling large scenes.

11.2.1 Sample-Based Volumetric Material Variations

Material definition is an important part of this procedural modeling system. I am currently investigating methods for creating high-quality solid materials from sample real-world imagery. For example, extensive visual catalogs of architectural construction materials, such as wood, marble, granite and concrete, could be searched to select a particular appearance for the model [Juracek 1996, Studio Marmo 1998, Studio Marmo 2001, Juracek 2002] .

I plan to use these solid materials not only for rendering, but also for simulations, to model fine-grain variations that lead to important material-specific behaviors. Below I describe some background in texture synthesis and procedural shader parameter estimation, limitations of the current

technology, and examples of how these materials could be used within the modeling framework.

Background: Texture Synthesis

Texture synthesis is a field of computer graphics that aims to construct large non-periodic blocks of texture that are visually similar to small input samples. Interest in this field was initially sparked by crossover papers from image analysis, such as Heeger and Bergen [1995] and De Bonet [1997], which replicated various statistical properties from the input image. Dischler and Ghazanfarpour investigated a number of ways to transfer this work to three dimensions [Dischler and Ghazanfarpour 1994, Ghazanfarpour and Dischler 1996, Dischler et al. 1998]. Their work is exciting because it allows the construction of anisotropic solid textures, by specifying different sample two-dimensional images for each axis.

However, statistical analysis cannot capture the important features of structured textures such as brick, long blades of grass, or pebbles. A more general-purpose synthesis technique assumes that texture can be modeled as a Markov Random Field [Efros and Leung 1999, Wei and Levoy 2000]. This approach generates the output image, one pixel at a time, by finding a region in the input that best matches the currently assigned neighbors in the output. While the single-pixel-at-a-time method can synthesize a wide variety of textures, the algorithm is very slow. Some of the best and most efficient overall results to date are achieved by copying large overlapping patches of the input sample and choosing the optimal seam within the overlapped region [Efros and Freeman 2001, Kwatra et al. 2003].

Background: Parameter Estimation

Approaching the texture synthesis problem from a different angle, a number of researchers study specific textures that can be modeled by classes of procedural shaders. If an appropriate shader can be constructed, then the generation of texture becomes simply an evaluation of this function.

In an example from material science, structural engineers estimate the strength of concrete from the size and distribution of aggregates in a two-dimensional slice [Taylor 1983, Hagwood 1990, Bentz et al. 1995, Bentz et al. 1996]. Similarly, Lefebvre and Poulin [2000] analyze the periodic nature of wood and brick textures to deduce the correct parameters of these relatively simple procedural shaders. Finally, Bourque and Dudek [2002] present a framework for choosing, from

a small library of procedural textures, a particular shader and the appropriate parameter assignments to best match the input sample. Unfortunately the search space for this general technique is massive, and the method cannot generalize for textures not present in the library.

Extending the Current Technology

There are a number of limitations that must be addressed before texture synthesis can provide the rich materials necessary for focal objects in photorealistic renderings. Most techniques are demonstrated on very small samples, often just 64x64 pixels, and the algorithms do not scale well for significantly larger images. Some of the samples I would like to convert to three-dimensional solid materials are up to 850x625 pixels. Additionally, many texture synthesis techniques require specification of an intrinsic *neighborhood size*, but selecting an appropriate value for this parameter can be difficult. Often a brute-force strategy is used — try several sizes and choose the one that produces the best results. Appropriate neighborhood size selection becomes even more critical when high-resolution images with hierarchical texture variation are used as input.

I have implemented a parallel version of the pixel-at-a-time synthesis strategy. I visualize the inner workings of this computation, to allow analysis and improvement of the algorithm. My preliminary results are comparable to the solid texture synthesis results of Wei [2001]. I also plan to investigate acceleration techniques for pixel-at-a-time texture synthesis [Ashikhmin 2001, Zelinka and Garland 2002] and hierarchical methods to reproduce both the large scale structure and small scale details of the input textures.

Integration with Simulation

To use synthesized solid textures in simulation, a data structure that stores sub-tetrahedral material variations must be utilized. These textures can denote variations in the material properties (such as transparency, shininess, density or fracture threshold), which are used in rendering and simulation.

11.2.2 Improved Tetrahedralization and Simplification Algorithms

The grid-based tetrahedralization approach described in Chapters 7 and 8 has been used to construct large and detailed volumetric models from complex scanned meshes. However, there are several ways the algorithm can be improved.

First, I would like to augment the model construction with Hermite data as in Ju et al. [2002] to reduce the artifacts near sharp edges in the model (Section 7.4.5). A related problem is the prohibitively high-resolution grid required to guarantee the correct topology of certain objects. If the combined tetrahedralization and simplification algorithm can be extended to match the original surface mesh (assuming it contains no self-intersections), a lower-resolution grid would be sufficient. Likewise, a tetrahedralization strategy that matches the triangulation of the input mesh could be used to build models with touching or interlocking parts such as gears and chains, or a human figure with hands on hips or legs crossed.

The octree grid for tetrahedralization is currently constructed using a bottom-up strategy, which begins with a uniform high-resolution grid and collapses cells in smooth regions of the distance field (Section 8.2). However, for improved memory performance a top-down approach, similar to Perry and Frisken [2001], which selectively refines a medium resolution grid, could be developed to work with the level sets infrastructure.

In Section 9.2, I describe the normal interpolation strategy that automatically detects sharp features in the model. In addition to several ambiguous cases for static models, additional artifacts may be encountered when performing significant deformation. In particular, if the dihedral angle between two surface triangles crosses the *sharpness threshold*, the edge will switch from "sharp" to "smooth" or vice versa. Similarly, a high-resolution sphere with many surface elements will appear uniformly smoothed, but as it is simplified to an extremely low-resolution version, some of the edges will be incorrectly labeled "sharp." The simplification algorithm could be extended with information about this rendering strategy and an increase in the cost of changes to the mesh that introduce errors in the interpolated surface normals. Alternatively, the tetrahedralization and simplification algorithms could be responsible for assigning and maintaining normals, which would then be explicitly stored with the mesh.

11.2.3 Language Extensions

The modeling language has been implemented with a high-level scripting syntax and optional user-defined C functions to represent the procedural components. The functions are extracted from the script, then compiled and linked at runtime with the core system. Writing interesting procedural definitions for material or simulation behavior requires knowledge about the simulation modules

and data structures used in the core implementation.

As the language evolves, it may be beneficial to define additional abstractions and re-implement the modeling language. I considered implementing the language with Python, an interpreted, interactive, object-oriented language [Beazley 2001]. Given that the core of the modeling system is implemented in C++, this option was attractive because Python can be *extended* with modules from C or *embedded* within a C program. Unfortunately, significant effort is required to write conversion functions for sharing data between Python and C modules. Additionally, the Python→C and C→Python call overhead is not insignificant and should not be included in the innermost loop of a computationally-intensive program.

Alternatively, a language in which procedures are first-class data types, such as Scheme [Abelson and Sussman 1996], would make specification of the particle system operators (Section 5.3) more concise. Furthermore, I would like to reformulate the language to alternate more seamlessly between the model initialization phase in Chapter 4 and the sculpting and simulation phase in Chapter 5 — for example, allowing a layer of material to be applied after an erosion simulation. See also Figure 3.1.

11.2.4 Volume Addition

Unfortunately, the addition of material layers at an intermediate stage of modeling (for example, between physical simulations, as suggested in the previous section) is beyond the current implementation. In fact, the system lacks a basic CSG-style material addition tool. General-purpose addition of volume to a tetrahedral mesh is challenging, because the new volume must match the triangulation of the existing mesh, which is not the case in structured mesh generation (Section 7.4). Even if the new volume is tetrahedralized on the same grid as the original geometry, the meshing will not match the current model if it has been simplified or deformed. The addition operation could be performed by resetting the entire model to the grid, but this is inefficient for small edits, and impossible for an interactive setting. These difficulties can be resolved by using a tetrahedralization strategy that matches the existing triangulation, or perhaps by developing a hybrid data representation between tetrahedral meshes and axis-aligned grids.

11.2.5 New Simulation Operators

The particle system tools described in Chapter 5 modify the color of the stone gargoyle and the polish tool increases the shininess of the bronze cat (Figure 10.8), but these modified material properties are only used during rendering, and only effected a per-vertex change. More dramatic material modifications could also affect future simulations by changing material properties such as stiffness, elasticity or the fracture threshold. There are many possibilities for new physically-based simulations that model material change due to water, heat, and chemical reactions. Examples include the interesting work by Carlson et al. [2002] on melting phenomena and by Paquette et al. [2002] on different peeling effects.

To incorporate these simulations into the system, hybrid data structures that store and manipulate sub-tetrahedral material variations are needed. These data structures would allow opportunities for simulations on materials that have intricate internal structures, such as wood or cement-based products. I believe all physical simulations would benefit from an authoring language similar to mine, not just those that are best modeled with tetrahedral meshes.

11.2.6 Library of Modeling and Simulation Operations

Many additional weathering effects could be applied to the examples in Chapter 10. Unfortunately, I was limited to the operators that I was able to construct from the simulations available in our system. With more time, and additional knowledge about the physical behavior of the different materials, I could have developed additional effects. Furthermore, creating convincing procedural definitions for real-world materials such as wood, concrete, marble, or granite takes significant effort from a skilled programmer. Often it requires specific knowledge about the composition of the material.

Fortunately, once a procedural definition is finished, it can be reused and shared with other artists. For example, many RenderMan shaders are available free or for purchase in online repositories [Lancaster]. Likewise, an important task for future work is to establish a library of the procedural components of the solid modeling language, including material decomposition, layer thickness patterns, geometry modification tools, and particle flow and interaction functions.

11.2.7 Hybrid Data Structure

The tetrahedral data structure described in Chapter 6 was chosen for its flexibility in representing geometry and surface details, and its compatibility with simulations such as FEM. However, as mentioned above, in some cases an alternate representation or a hybrid between a tetrahedral mesh and another representation will best capture the desired volumetric properties.

For example, in the bronze cat statue (Section 10.4), the general shape of the broken clay was physically plausible, but the large polygonal faces betrayed the synthetic nature of the scene. This artifact of tetrahedralization could be disguised with a displacement map that adds random surface variations to the model. The effect would be especially convincing if a separate fine-scale simulation, based on the microscopic properties of the material, was used to create the displacement map. Similarly, an opacity mask for thick triangles can improve the shape of the elements along the edges of the blistered paint in Figure 6.4.

11.2.8 Strategy for Handling Large Scenes

Finally, I would like to develop a strategy for combining the results of several simulations in a large scene. For example, the plaster and brick facade model (Section 10.2) pushed the storage limitations of the representation. The sculpting efficiency of this model could be improved with delayed instantiation — that is, tetrahedralization of the visible portion of the model (the plaster), coupled with a procedural representation of the underlying layers (the bricks). Then, when bricks are uncovered, they are locally instantiated (tetrahedralized). This strategy would allow representation of a collection of such buildings along a Venetian canal.

Additionally, objects may interact with each other but be too complex to manipulate in a single simulation. This task could be approached using multi-resolution simulations.

11.3 Summary

I have presented a procedural framework for authoring layered solid models and applying a series of simulation operations to them. This approach allows complex volumetric models to be constructed from existing triangle meshes as well as from implicit functions and distance fields. These

different modeling approaches are handled seamlessly within a high-level framework. The models can then be easily modified using sculpting and simulation operators.

This is one of the first modeling systems where simulation is treated as a sculpting tool rather than merely for animation — an approach with tremendous potential. In general, the language provides both a higher level of abstraction for, and a convenient interface to, existing simulation environments. The scripting language is also valuable as an intermediate file representation for capturing the history of interactive sculpting operations.

The system has been used to successfully construct models for a wide range of rendering, simulation, and animation applications. It has been used to construct small models, with a few hundred tetrahedra, for use in real-time animation research, as well as large models with millions of tetrahedra for offline weathering and erosion simulations. In fact, models at either resolution can be constructed from essentially the same script.

Overall, I believe that a procedural interface between modeling and simulation is an important tool for the computer graphics community. The system provides a novel approach to authoring models, which has allowed a dramatic increase in modeling productivity and flexibility, smoothed transitions of models between simulation and rendering applications, and provided access to complex simulation systems to novice users.

Appendix A

Modeling Language Specification

Below is the specification for the procedural solid modeling language described in Chapters 4 and 5. Examples of models created with the language appear throughout this document.

A.1 Type Grammar

script : (*assignment* | *operation*)*

assignment : *identifier* = *value*

operation : *function* { *assignment** }

value : `NULL` | *integer* | *float* | *vec3f* | *string* | *substance* | *layer* | *distance_field* | *volume* | *function* | { *value** }

vec3f : { *float* *float* *float* }

layers : { *layer** }

distance_field : function: *vec3f* ⇒ *float*

substance : *material* | function: *vec3f* ⇒ *material* | `nothing`

thickness : *float* | `fill`

velocity : *float* | function: *vec3f* ⇒ *float* | function: *vec3f, distance_field* ⇒ *float*

appliedArea : `GaussSphere` (*vec3f* `position`, *vec3f* `orientation`) | `Sphere` (*vec3f* `position`, *vec3f* `orientation`) | `HalfSpace` (*vec3f* `position`, *vec3f* `orientation`)

A.2 Selected Modeling Functions

```
material    material              ( string name,
                                    float density = 1.0,
                                    etc. );

layer       layer                 ( substance material,
                                    thickness thickness = 1.0 );

distance_field  from_surface_mesh ( string file );

distance_field  from_volume_surface ( volume volume );

distance_field  union             ( distance_field distance_field_1,
                                    distance_field distance_field_2 );

distance_field  from_isosurface   ( distance_field distance_field,
                                    float isosurface = 0.0,
                                    velocity velocity = 1.0 );

distance_field  transformation    ( distance_field distance_field,
                                    vec3f translation = { 0.0 0.0 0.0 },
                                    vec3f rotation = { 0.0 0.0 0.0 },
                                    float scale = 1.0 );

volume      volume                ( distance_field distance_field,
                                    layers interior_layers = NULL,
                                    layers exterior_layers = NULL );

volume      precedence            ( volume volume_1,
                                    volume volume_2 );

volume      load                  ( string tetra_file );

void        save                  ( volume model,
                                    string tetra_file = NULL,
                                    string triangle_file = NULL );

void        simplify              ( volume model,
                                    int desired_tetra_count = 5000 );
```

A.3 Sculpting and Simulation Modules

```
void  CSG              ( volume model,
                         appliedArea tool,
                         material material,
                         materials affects );

void  FEM              ( volume model,
                         vec3f force,
                         vec3f area,
                         materials affects,
                         int collisiondetection,
                         float gravity );

void  ParticleSystem   ( volume model,
                         int num_particles,
                         function: model ⇒ particle initialize,
                         function: particle ⇒ int life,
                         function: model, particle, particles ⇒ void motion,
                         function: model, particle ⇒ void action );

void  Peel             ( volume model,
                         vec3f position,
                         float size );
```

A.4 Syntactic Sugar

```
VOLUME = precedence_layers {
  volume_1 = VOLUME_A
  volume_2 = VOLUME_B
  exterior_layers = LAYERS }
```

$\Longrightarrow$

```
INTERMEDIATE = precedence {
  volume_1 = VOLUME_A
  volume_2 = VOLUME_B }

VOLUME = precedence {
  volume_1 = INTERMEDIATE
  volume_2 = volume {
    distance_field = from_volume_surface {
      volume = INTERMEDIATE }
    exterior_layers = LAYERS } } }
```

```
VOLUME = volume {
  distance_field = FIELD
  interior_layers = LAYERS
  exterior_layers = {
    layer {
      material = MATERIAL_1
      thickness = THICK_1
      velocity = VELOCITY_1 }
    ...
    layer {
      material = MATERIAL_n
      thickness = THICK_n
      velocity = VELOCITY_n }
    layer {
      material = MATERIAL
      thickness = THICK
      velocity = VELOCITY } } }
```

$\Longrightarrow$

```
INTERMEDIATE = volume {
  distance_field = FIELD
  interior_layers = LAYERS
  exterior_layers = {
    layer {
      material = MATERIAL_1
      thickness = THICK_1
      velocity = VELOCITY_1 }
    ...
    layer {
      material = MATERIAL_n
      thickness = THICK_n
      velocity = VELOCITY_n } } }

VOLUME = precedence {
  volume_1 = INTERMEDIATE
  volume_2 = volume {
    distance_field = from_volume_surface {
      volume = INTERMEDIATE
      velocity = VELOCITY }
    exterior_layers = {
      layer {
        material = MATERIAL
        thickness = THICK } } } }
```

A similar desugaring rule applies for `interior_layers`.

```
from_surface_mesh {
  file = FILE
  velocity = VELOCITY }
```

$\Longrightarrow$

```
from_isosurface {
  distance_field = from_surface_mesh {
    file = FILE }
  isosurface = 0.0
  velocity = VELOCITY }
```

```
from_volume_surface {
  volume = VOLUME
  velocity = VELOCITY }
```

$\Longrightarrow$

```
from_isosurface {
  distance_field = from_volume_surface {
    volume = VOLUME }
  isosurface = 0.0
  velocity = VELOCITY }
```

```
union {
  distance_field_1 = FIELD_1
  distance_field_2 = FIELD_2
  velocity = VELOCITY }
```

$\Longrightarrow$

```
from_isosurface {
  distance_field = union {
    distance_field_1 = FIELD_1
    distance_field_2 = FIELD_2 }
  isosurface = 0.0
  velocity = VELOCITY }
```

Bibliography

1. ABELSON, H., AND SUSSMAN, G. J. 1996. *Structure and Interpretation of Computer Programs*, 2nd ed. MIT Press.

2. ADLER, A. 1983. On the bisection method for triangles. *Mathematics of Computation*, 40(162) (April), 571–574.

3. ADZHIEV, V., CARTWRIGHT, R., FAUSETT, E., OSSIPOV, A., PASKO, A., AND SAVCHENKO, V. 1999. HyperFun Project: A framework for collaborative multidimensional F-rep modeling. In *Proceedings of Implicit Surfaces '99*, 59–69.

4. AGARWALA, A. 1999. *Volumetric Surface Sculpting*. Master's thesis, Massachusetts Institute of Technology.

5. AGRAWALA, M., BEERS, A. C., AND LEVOY, M. 1995. 3D painting on scanned surfaces. In *1995 Symposium on Interactive 3D Graphics*, 145–150.

6. ALIAS SYSTEMS. Maya. `http://www.alias.com`.

7. ANDERSON, H. L., Ed. 1989. *A Physicist's Desk Reference*, 2nd ed. American Institute of Physics, New York.

8. APODACA, A. A., AND GRITZ, L. 1999. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann.

9. ARNOLD, D. N., MUKHERJEE, A., AND POULY, L. 2000. Locally adapted tetrahedral meshes using bisection. *SIAM Journal of Scientific Computing*, 22(3), 431–448.

10. ASHIKHMIN, M. 2001. Synthesizing natural textures. In *2001 ACM Symposium on Interactive 3D Graphics*, 217–226.

11. AUTODESK, INC. AutoCAD. `http://www.autodesk.com`.

12. BABUSKA, I., AND AZIZ, A. 1976. On the angle condition in the finite element method. *SIAM Journal of Numerical Analysis*, 13(2), 214–226.

13. BAKER, T. J. 1989. Automatic mesh generation for complex three-dimensional regions using a constrained delaunay triangulation. *Engineering with Computers*, 5, 161–175.

14. BANK, R. E., SHERMAN, A. H., AND WEISER, A. 1983. Some refinement algorithms and data structures for regular local mesh refinement. 3–17.

15. BARAFF, D., AND WITKIN, A. 1998. Large steps in cloth simulation. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, 43–54.

16. BARAFF, D., WITKIN, A., AND KASS, M. 2003. Untangling cloth. *ACM Transactions on Graphics*, 22(3) (July), 862–870.

17. BEAZLEY, D. M. 2001. *Python: Essential Reference*, 2nd ed. New Riders Publishing.

18. BENSON, D., AND DAVIS, J. 2002. Octree textures. In *Proceedings of ACM SIGGRAPH 2002*, Computer Graphics Proceedings, Annual Conference Series, 785–790.

19. BENTON, J. R. 1997. *Holy Terrors: Gargoyles on Medieval Buildings*. Abbeville Press, Inc.

20. BENTZ, D. P., GARBOCZI, E. J., AND MARTYS, N. S. 1995. Multi-scale digital-image-based modelling of cement-based materials. In *Materials Research Society Symposium Proceedings*, vol. 370, 33–41.

21. BENTZ, D. P., GARBOCZI, E. J., AND MARTYS, N. S. 1996. Application of digital-image-based models to microstructure, transport properties, and degradation of cement-based materials. In *Modelling of Microstructure and Its Potential for Studying Transport Properties and Durability*, e. a. H. Jennings, Ed. Kluwer Academic Publishers, the Netherlands, 167–185.

22. BEY, J. 1995. Tetrahedral grid refinement. *Computing*, vol. 55, 355–378.

23. BLINN, J. 1996. *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*. Morgan Kaufmann Publishers, Inc., San Francisco, California.

24. BLOOMENTHAL, J., AND FERGUSON, K. 1995. Polygonization of non-manifold implicit surfaces. In *Proceedings of ACM SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, 309–316.

25. BLOOMENTHAL, J. 1994. An implicit surface polygonizer. In *Graphics Gems IV*. Academic Press, Boston, 324–349.

26. BOCCAZZI-VAROTTO, A. 1996. *Venice 360 Degrees*. Random House.

27. BOURQUE, E., AND DUDEK, G. 2002. Automated parameter estimation for procedural texturing. Poster Session at Eurographics Workshop on Rendering 2002.

28. BRIDSON, R., FEDKIW, R., AND ANDERSON, J. 2002. Robust treatment of collisions, contact and friction for cloth animation. *ACM Transactions on Graphics*, 21(3) (July), 594–603.

29. BUCHANAN, J. W. 1998. Simulating wood using a voxel approach. *Computer Graphics Forum*, 17(3), 105–112.

30. CARLSON, M., MUCHA, P. J., HORN, R. B. V., AND TURK, G. 2002. Melting and flowing. In *Proceedings of ACM SIGGRAPH Symposium on Computer Animation 2002*, 167–174.

31. CAVALCANTI, P. R., AND MELLO, U. T. 1999. Three-dimensional constrained delaunay triangulation: a minimalist approach. In *Proceedings of the 8th International Meshing Roundtable*, 119–129.

32. CHOPRA, P., AND MEYER, J. 2002. Tetfusion: An algorithm for rapid tetrahedral mesh simplification. In *Proceedings of IEEE Visualization 2002*, 133–140.

33. CIGNONI, P., MONTANI, C., ROCCHINI, C., SCOPIGNO, R., AND TARINI, M. 1999. Preserving attribute values on simplified meshes by resampling detail textures. *The Visual Computer*, 15(10), 519–539.

34. CIGNONI, P., COSTANZA, D., MONTANI, C., ROCCHINI, C., AND SCOPIGNO, R. 2000. Simplification of tetrahedral meshes with accurate error evaluation. In *IEEE Visualization 2000*, 85–92.

35. CONRAUD, J. 1995. Lazy constrained tetrahedralization. In *Proceedings of the 4th International Meshing Roundtable*, 15–26.

36. COOK, R. L. 1984. Shade trees. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, 18(3), 223–231.

37. COQUILLART, S. 1990. Extended free-form deformation: A sculpturing tool for 3D geometric modeling. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4), 187–196.

38. CRIST, D. T. 2001. *American Gargoyles: Spirits in Stone*. Clarksn Potter Publishers, New York, New York. Photographs by Robert Llewellyn.

39. CURTIS, C. J., ANDERSON, S. E., SEIMS, J. E., FLEISCHER, K. W., AND SALESIN, D. H. 1997. Computer-generated watercolor. 421–430.

40. CUTLER, B., DORSEY, J., MCMILLAN, L., MÜLLER, M., AND JAGNOW, R. 2002. A procedural approach to authoring solid models. *ACM Transactions on Graphics*, 21(3) (July), 302–311.

41. CYBERWARE. Model shop color 3D scanner. `http://www.cyberware.com`.

42. DACHILLE IX, F., QIN, H., KAUFMAN, A. E., AND EL-SANA, J. 1999. Haptic sculpting of dynamic surfaces. In *1999 ACM Symposium on Interactive 3D Graphics*, 103–110.

43. DAVIS, J., MARSCHNER, S. R., GARR, M., AND LEVOY, M. 2002. Filling holes in complex surfaces using volumetric diffusion. In *First International Symposium on 3D Data Processing, Visualization, and Transmission*.

44. DE BONET, J. S. 1997. Multiresolution sampling procedure for analysis and synthesis of texture images. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, 361–368.

45. DEBRY, D., GIBBS, J., PETTY, D. D., AND ROBINS, N. 2002. Painting and rendering textures on unparameterized models. In *Proceedings of ACM SIGGRAPH 2002*, Computer Graphics Proceedings, Annual Conference Series, 763–768.

46. DEBUNNE, G., DESBRUN, M., CANI, M.-P., AND BARR, A. H. 2001. Dynamic real-time deformations using space & time adaptive sampling. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, 31–36.

47. DEMKO, S., HODGES, L., AND NAYLOR, B. 1985. Construction of fractal objects with iterated function systems. In *Computer Graphics (Proceedings of ACM SIGGRAPH 85)*, 19(3), 271–278.

48. DEUSSEN, O., HANRAHAN, P. M., LINTERMANN, B., MECH, R., PHARR, M., AND PRUSINKIEWICZ, P. 1998. Realistic modeling and rendering of plant ecosystems. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, 275–286.

49. DISCHLER, J.-M., AND GHAZANFARPOUR, D. 1994. A geometrical based method for highly complex structured textures generation. *Computer Graphics Forum*, 14(4) (Oct.), 203–216.

50. DISCHLER, J. M., GHAZANFARPOUR, D., AND FREYDIER, R. 1998. Anisotropic solid texture synthesis using orthogonal 2D views. *Computer Graphics Forum*, 17(3), 87–96.

51. DORSEY, J., AND HANRAHAN, P. M. 1996. Modeling and rendering of metallic patinas. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, 387–396.

52. DORSEY, J., PEDERSEN, H. K., AND HANRAHAN, P. M. 1996. Flow and changes in appearance. In *Proceedings of ACM SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, 411–420.

53. DORSEY, J., EDELMAN, A., LEGAKIS, J., JENSEN, H. W., AND PEDERSEN, H. K. 1999. Modeling and rendering of weathered stone. In *Proceedings of ACM SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, 225–234.

54. EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND WORLEY, S. 1998. *Texturing & Modeling*, 2nd ed. Academic Press.

55. EFROS, A. A., AND FREEMAN, W. T. 2001. Image quilting for texture synthesis and transfer. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, 341–346.

56. EFROS, A. A., AND LEUNG, T. K. 1999. Texture synthesis by non-parametric sampling. In *IEEE International Conference on Computer Vision*, 1033–1038.

57. ENRIGHT, D., MARSCHNER, S., AND FEDKIW, R. 2002. Animation and rendering of complex water surfaces. In *Proceedings of ACM SIGGRAPH 2002*, Computer Graphics Proceedings, Annual Conference Series, 736–744.

58. EVISON, M. P. 1996. Computerised 3D facial reconstruction. *Assemblage: The Sheffield Graduate Journal of Archaeology*, 1.

59. FLEISCHMANN, P. 1999. *Mesh Generation for Technology CAD in Three Dimensions.* PhD dissertation, Technical University of Vienna.

60. FREITAG, L. A., AND OLLIVIER-GOOCH, C. 1997. Tetrahedral mesh improvement using swapping and smoothing. *International Journal for Numerical Methods in Engineering*, vol. 40, 3979–4002.

61. FREY, W. H., AND FIELD, D. 1991. Mesh relaxation: A new technique for improving triangulations. *International Journal for Numerical Methods in Engineering*, 31(6), 1121–1133.

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

63. GALYEAN, T., AND HUGHES, J. 1991. Sculpting: An interactive volumetric modeling technique. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, 25(4), 267–274.

64. GANOVELLI, F., AND C.O'SULLIVAN. 2001. Animating cuts with on-the-fly re-meshing. In *Eurographics 2001 - short presentations*, 243–247.

65. GANOVELLI, F., CIGNONI, P., MONTANI, C., AND SCOPIGNO, R. 2000. A multiresolution model for soft objects supporting interactive cuts and lacerations. *Computer Graphics Forum*, 19(3).

66. GANOVELLI, F., CIGNONI, P., MONTANI, C., AND SCOPIGNO, R. 2001. Enabling cuts on multiresolution representation. *The Visual Computer*, vol. 17, 274–286.

67. GARLAND, M. Qslim surface simplification software. `http://www-2.cs.cmu.edu/afs/cs/user/garland/www/quadrics/quadrics.html`.

68. GARLAND, M., AND HECKBERT, P. S. 1997. Surface simplification using quadric error metrics. In *Proceedings of ACM SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, 209–216.

69. GEORGE, P. L. 1999. Tet meshing: Construction, optimization, and adaptation. In *8th International Meshing Roundtable*, 133–141.

70. GHAZANFARPOUR, D., AND DISCHLER, J.-M. 1996. Generation of 3D texture using multiple 2D models analysis. *Computer Graphics Forum*, 15(3) (Aug.), 311–324.

71. GUSKOV, I., AND WOOD, Z. 2001. Topological noise removal. In *Graphics Interface*, 19–20.

72. HAGWOOD, C. 1990. A mathematical treatment of the spherical stereology. Tech. Rep. NISTIR 4370, National Institude of Standards and Technology.

73. HANRAHAN, P., AND HAEBERLI, P. E. 1990. Direct wysiwyg painting and texturing on 3D shapes. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4), 215–223.

74. HANRAHAN, P., AND LAWSON, J. 1990. A language for shading and lighting calculations. In *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, 24(4), 289–298.

75. HEEGER, D. J., AND BERGEN, J. R. 1995. Pyramid-based texture analysis/synthesis. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, 229–238.

76. HIROTA, K., TANOUE, Y., AND KANEKO, T. 1998. Generation of crack patterns with a physical model. *The Visual Computer*, 14(3), 126–137.

77. HODGES, H. 1970. *Technology in the Ancient World.* Alfred A Knopf, New York.

78. HOPPE, H., DEROSE, T., DUCHAMP, T., MCDONALD, J., AND STUETZLE, W. 1993. Mesh optimization. In *Proceedings of SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, 19–26.

79. HOPPE, H. 1996. Progressive meshes. In *Proceedings of ACM SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, 99–108.

80. HSU, W., HUGHES, J., AND KAUFMAN, H. 1992. Direct manipulation of free-form deformations. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26(4), 177–184.

81. IGARASHI, T., MATSUOKA, S., AND TANAKA, H. 1999. Teddy: A sketching interface for 3d freeform design. In *Proceedings of ACM SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, 409–416.

82. JAGNOW, R., AND DORSEY, J. 2002. Virtual sculpting with haptic displacement maps. In *Graphics Interface*, 125–132.

83. JAGNOW, R. 2001. *Virtual Sculpting with Haptic Displacement Maps.* Master's thesis, Massachusetts Institute of Technology.

84. JOE, B. 1995. Construction of three-dimensional improved-quality triangulations using local transformations. *Mathematics of Computation*, vol. 16, 1292–1307.

85. JU, T., LOSASSO, F., SCHAEFER, S., AND WARREN, J. 2002. Dual contouring of hermite data. In *Proceedings of ACM SIGGRAPH 2002*, Computer Graphics Proceedings, Annual Conference Series, 302–311.

86. JURACEK, J. A. 1996. *Surfaces: Visual Reserach for Artists, Architects, and Designers.* W.W. Norton & Company, Inc.

87. JURACEK, J. A. 2002. *Natural Surfaces: Visual Reserach for Artists, Architects, and Designers.* W.W. Norton & Company, Inc.

88. KRIZEK, M. 1992. On the maximum angle condition for linear tetrahedral elements. *SIAM Journal of Numerical Analysis*, vol. 29, 513–520.

89. KWATRA, V., SCHDL, A., ESSA, I., TURK, G., , AND BOBICK, A. 2003. GraphCut Textures: Image and video synthesis using graph cuts. In *Proceedings of ACM SIGGRAPH 2003*, Computer Graphics Proceedings, Annual Conference Series, 277–286.

90. LAIDLAW, D., TRUMBORE, W., AND HUGHES, J. 1986. Constructive solid geometry for polyhedral objects. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, 20(4), 161–170.

91. LAMORLETTE, A., AND FOSTER, N. 2002. Structural modeling of flames for a production environment. In *Proceedings of ACM SIGGRAPH 2002*, Computer Graphics Proceedings, Annual Conference Series, 729–735.

92. LANCASTER, T. Renderman® repository. `http://www.renderman.org/`.

93. LANGLAND, T. 1999. *From Clay To Bronze*. Waston-Guptill Publishers.

94. LEFEBVRE, S., AND NEYRET, F. 2002. Synthesizing bark. In *Proceedings of Eurographics Workshop on Rendering 2002*, 105–117.

95. LEFEBVRE, L., AND POULIN, P. 2000. Analysis and synthesis of structural textures. In *Graphics Interface*, 77–86.

96. LEGAKIS, J., DORSEY, J., AND GORTLER, S. J. 2001. Feature-based cellular texturing for architectural models. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, 309–316.

97. LEVINE, J. R., MASON, T., AND BROWN, D. 1992. *lex & yacc*, 2nd ed. O'Reilly & Associates, Sebastopol, California.

98. LEWIS, J.-P. 1989. Algorithms for solid noise synthesis. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, 23(3), 263–270.

99. LIN, M., AND GOTTSCHALK, S. 1998. Collision detection between geometric models: A survey. In *IMA Conference on Mathematics of Surfaces*.

100. LIU, A., AND JOE, B. 1994. On the shape of tetrahedra from bisection. *Mathematics of Computation*, 63(207) (July), 141–154.

101. LIU, A., AND JOE, B. 1994. Relationship between tetrahedron shape measures. *BIT*, vol. 34, 268–287.

102. LIU, A., AND JOE, B. 1995. Quality local refinement of tetrahedral meshes based on bisection. *SIAM Journal of Scientific Computing*, 16(6) (November), 1269–1291.

103. LIU, A., AND JOE, B. 1996. Quality local refinement of tetrahedral meshes based on 8 subtetra-

heron subdivision. *Mathematics of Computation*, 65(215), 1183–1200.

104. LOHNER, R. 1988. Generation of three-dimensional unstructured grids by the advancing front method. In *International Journal for Numerical Methods in Fluids*, vol. 8, 1135–1149.

105. LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3D surface construction algorithm. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, 21(4), 163–169.

106. MAUBACH, J. M. 1995. Local bisection refinement for n-simplicial grids generated by reflection. *SIAM Journal of Scientific Computing*, vol. 16, 210–227.

107. MAUBACH, J. M. 1996. The efficient location of neighbors for locally refined n-simplicial grids. In *5th International Meshing Roundtable*, 137–153.

108. MERILLOU, S., DISCHLER, J.-M., AND GHAZANFARPOUR, D. 2001. Corrosion: Simulating and rendering. In *Graphics Interface 2001*, 167–174.

109. MIZUNO, S., OKADA, M., AND ICHIRO TORIWAKI, J. 1998. Virtual sculpting and virtual woodcut printing. *The Visual Computer*, 14(2), 39–51.

110. MOLINO, N., BRIDSON, R., TERAN, J., AND FEDKIW, R. 2003. A crystalline, red green strategy for meshing highly deformable objects with tetrahedra.

111. MÜLLER, M., DORSEY, J., MCMILLAN, L., AND JAGNOW, R. 2001. Real-time simulation of deformation and fracture of stiff materials. In *Proceedings of Eurographics Workshop on Animation and Simulation 2001*, 113–124.

112. MÜLLER, M., DORSEY, J., MCMILLAN, L., JAGNOW, R., AND CUTLER, B. 2002. Stable real-time deformations. In *Proceedings of ACM SIGGRAPH Symposium on Computer Animation 2002*, 49–54.

113. MUSETH, K., BREEN, D. E., WHITAKER, R. T., AND BARR, A. H. 2002. Level set surface editing operators. *ACM Transactions on Graphics*, 21(3) (July), 330–338.

114. MUSGRAVE, F. K., KOLB, C. E., AND MACE, R. S. 1989. The synthesis and rendering of eroded fractal terrains. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, 23(3), 41–50.

115. NGUYEN, D. Q., FEDKIW, R., AND JENSEN, H. W. 2002. Physically based modeling and animation of fire. In *Proceedings of ACM SIGGRAPH 2002*, Computer Graphics Proceedings, Annual Conference Series, 721–728.

116. NIELSON, G. M., AND SUNG, J. 1997. Interval volume tetrahedrization. In *IEEE Visualization '97*, 221–228.

117. NOORUDDIN, F. S., AND TURK, G. 2000. Interior/exterior classification of polygonal models. In *IEEE Visualization 2000*, 415–422.

118. NORRUDDIN, F., AND TURK, G. 1999. Simplification and repair of polygonal models using volumetric techniques. Tech. rep., Georgia Tech GVU.

119. O'BRIEN, J. F., AND HODGINS, J. K. 1999. Graphical modeling and animation of brittle fracture. In *Proceedings of ACM SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, 137–146.

120. OPPENHEIMER, P. E. 1986. Real time design and animation of fractal plants and trees. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, 20(4), 55–64.

121. OWEN, S. J. 1998. A survey of unstructured mesh generation technology. In *7th International Meshing Roundtable*, 239–267. `http://www.andrew.cmu.edu/user/sowen/softsurv.html`.

122. PAQUETTE, E., POULIN, P., AND DRETTAKIS, G. 2002. The simulation of paint cracking and peeling. In *Graphics Interface*, 59–68.

123. PARISH, Y. I. H., AND MÜLLER, P. 2001. Procedural modeling of cities. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, 301–308.

124. PAYNE, B. A., AND TOGA, A. W. 1992. Distance field manipulation of surface models. *IEEE Computer Graphics & Applications*, 12(1), 65–71.

125. PEACHEY, D. R. 1985. Solid texturing of complex surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3), 279–286.

126. PERLIN, K., AND HOFFERT, E. M. 1989. Hypertexture. In *Computer Graphics (Proceedings of ACM SIGGRAPH 89)*, 23(3), 253–262.

127. PERLIN, K. 1985. An image synthesizer. In *Computer Graphics (Proceedings of ACM SIGGRAPH 85)*, 19(3), 287–296.

128. PERRY, R. N., AND FRISKEN, S. F. 2001. Kizamu: A system for sculpting digital characters. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, 47–56.

129. PFISTER, H., KAUFMAN, A., AND WESSELS, F. 1995. Towards a scalable architecture for real-time volume rendering. In *Proceedings of the 1995 Eurographics Workshop on Graphics Hardware*, 123–130.

130. PIRZADEH, S. 1993. Structured background grids for generation of unstructured grids by advancing front method. 257–265.

131. PIRZADEH, S. 1996. Three-dimensional unstructured viscous grids by the advancing-layers method. 43–49.

132. PRUSINKIEWICZ, P., AND LINDENMAYER, A. 1990. *The Algorithmic Beauty of Plants*. Springer-

Verlag.

133. PRUSINKIEWICZ, P., LINDENMAYER, A., AND HANAN, J. 1988. Developmental models of herbaceous plants for computer imagery purposes. In *Computer Graphics (Proceedings of ACM SIGGRAPH 88)*, 22(4), 141–150.

134. PRUSINKIEWICZ, P. 1986. Graphical applications of l-systems. In *Graphics Interface '86*, 247–253.

135. RAVIV, A., AND ELBER, G. 2000. Three-dimensional freeform sculpting via zero sets of scalar trivariate functions. *Computer-Aided Design*, 32(8–9), 513–526.

136. REEVES, W. T. 1983. Particle systems - a technique for modeling a class of fuzzy objects. In *Computer Graphics (Proceedings of SIGGRAPH 83)*, 17(3), 359–376.

137. RICCI, A. 1973. A constructive geometry for computer graphics. *The Computer Journal*, 16(2), 157–160.

138. RIVARA, M.-C., AND HITSCHFELD, N. 1999. LEPP-Delaunay algorithm: a robust tool for producing size-optimal quality triangulations. In *8th International Meshing Roundtable*, 205–220.

139. RIVARA, M.-C., AND LEVIN, C. 1992. A 3-D refinement algorithm suitable for adaptive and multi-grid techniques. *Communications in Applied Numerical Methods*, vol. 8, 281–290.

140. RIVARA, M.-C. 1996. New mathematical tools and techniques for the refinement and/or improvement of unstructured triangulations. In *5th International Meshing Roundtable*, 77–86.

141. ROCCHINI, C., CIGNONI, P., MONTANI, C., AND SCOPIGNO, R. 1999. Multiple textures stitching and blending on 3D objects. In *10th Eurographics Workshop on Rendering*, G. Ward and D. Lischinsky, Eds., 127–138.

142. SANDER, P. V., GU, X., GORTLER, S. J., HOPPE, H., AND SNYDER, J. 2000. Silhouette clipping. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, 327–334.

143. SCHROEDER, W. J., ZARGE, J. A., AND LORENSEN, W. E. 1992. Decimation of triangle meshes. *Computer Graphics*, 26(2), 65–70.

144. SEDERBERG, T. W., AND PARRY, S. R. 1986. Free-form deformation of solid geometric models. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, 20(4), 151–160.

145. SENSABLE TECHNOLOGIES. PHANTOM^{TM} haptics device and FreeForm® Modeling^{TM} system. `http://www.sensable.com`.

146. SETHIAN, J. A. 1999. *Level Set Methods and Fast Marching Methods*, 2nd ed. Cambridge University Press, Cambridge, United Kingdom.

147. SHEWCHUK, J. R. 1997. *Delaunay Refinement Mesh Generation.* PhD dissertation, Carnegie Mellon University, School of Computer Science. Technical Report CMU-CS-97-137.

148. SHEWCHUK, J. R. 1998. Tetrahedral mesh generation by delaunay refinement. In *Proceedings of the 14th Annual Symposium on Computational Geometry*, 86–95.

149. SHEWCHUK, J. R. 2000. Mesh generation for domains with small angles. In *Proceedings of the 16th Annual ACM Symposium on Computational Geometry*, ACM, 1–10.

150. SHEWCHUK, J. R. 2002. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications*, 22(1-2) (May), 21–74.

151. SIMS, K. 1991. Artificial evolution for computer graphics. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, 25(4), 319–328.

152. SIMS, K. 1994. Evolving virtual creatures. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, 15–22.

153. SMITH, J., WITKIN, A., AND BARAFF, D. 2000. Fast and controllable simulation of the shattering of brittle objects. In *Graphics Interface*, 27–34.

154. SMITH, A. R. 1984. Plants, fractals and formal languages. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, 18(3), 1–10.

155. STAADT, O. G., AND GROSS, M. H. 1998. Progressive tetrahedralizations. In *IEEE Visualization '98*, 397–402.

156. STANDER, B. T., AND HART, J. C. 1997. Guaranteeing the topology of an implicit surface polygonization. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, 279–286.

157. STRASSMANN, S. 1986. Hairy brushes. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, 20(4), 225–232.

158. STUDIO MARMO. 1998. *Natural Stone: A Guide to Selection.* W.W. Norton & Company, Inc. Text by Frederick Bradley.

159. STUDIO MARMO. 2001. *Fine Marble in Architecture.* W.W. Norton & Company, Inc. Text by Frederick Bradley.

160. TAYLOR, C. C. 1983. A new method for unfolding sphere size distributions. *Journal of Microscopy*, 132(1) (October), 57–66.

161. TERARECON, INC. Volumepro® 1000. `http://www.terarecon.com`.

162. TERZOPOULOS, D., AND QIN, H. 1994. Dynamic nurbs with geometric constraints for interactive sculpting. *ACM Transactions on Graphics*, 13(2) (Apr.), 103–136.

163. TROTTS, I. J., HAMANN, B., JOY, K. I., AND WILEY, D. F. 1998. Simplification of tetrahedral meshes. In *IEEE Visualization '98*, 287–296.

164. UPSTILL, S. 1990. *The Renderman Companion : A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley.

165. WANG, S. W., AND KAUFMAN, A. E. 1995. Volume sculpting. In *Symposium on Interactive 3D Graphics*, ACM Press, 151–156.

166. WEI, L.-Y., AND LEVOY, M. 2000. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, 479–488.

167. WEI, L.-Y. 2001. *Texture Synthesis by Fixed Neighborhood Searching*. PhD dissertation, Stanford University, Department of Electrical Engineering.

168. WILLIAMS, L. 1983. Pyramidal parametrics. In *Computer Graphics (Proceedings of SIGGRAPH 83)*, 17(3), 1–11.

169. WILLIAMS, L. 1990. 3D paint. In *1990 Symposium on Interactive 3D Graphics*, 24(2), 225–233.

170. WOO, M., NEIDER, J., DAVIS, T., SHREINER, D., AND OPENGL ARCHITECTURE REVIEW BOARD. 1999. *OpenGL® Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*, 3rd ed. Addison-Wesley.

171. WORLEY, S. P. 1996. A cellular texture basis function. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, 291–294.

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

173. WYVILL, G., WYVILL, B., AND MCPHEETERS, C. 1987. Solid texturing of soft objects. *IEEE Computer Graphics & Applications*, 7(12) (Dec.), 20–26.

174. WYVILL, B., GUY, A., AND GALIN, E. 1999. Extending the CSG tree. Warping, blending and boolean operations in an implicit surface modeling system. *Computer Graphics Forum*, 18(2), 149–158.

175. YERRY, M. A., AND SHEPHARD, M. S. 1984. Automatic three-dimensional mesh generation by the modified octree technique. *International Journal For Numerical Methods in Engineering*, 20, 1965–1990.

176. ZELINKA, S., AND GARLAND, M. 2002. Towards real-time texture synthesis with the jump map. In *Proceedings of Eurographics Workshop on Rendering 2002*, 99–104.

177. ZIENKIEWICZ, O. C., AND TAYLOR, R. L. 1989. *The Finite Element Method*, 4th ed. McGraw-Hill, New York.