A Procedural Approach to Authoring Solid Models

Barbara Cutler Julie Dorsey Leonard McMillan Matthias Müller Robert Jagnow

Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

We present a procedural approach to authoring layered, solid models. Using a simple scripting language, we define the internal structure of a volume from one or more input meshes. Sculpting and simulation operators are applied within the context of the language to shape and modify the model. Our framework treats simulation as a modeling operator rather than simply as a tool for animation, thereby suggesting a new paradigm for modeling as well as a new level of abstraction for interacting with simulation environments.

Capturing real-world effects with standard modeling techniques is extremely challenging. Our key contribution is a concise procedural approach for seamlessly building and modifying complex solid geometry. We present an implementation of our language using a flexible tetrahedral representation. We show a variety of complex objects modeled in our system using tools that interface with finite element method and particle system simulations.

Additional Keywords: volumetric modeling, signed-distance function, tetrahedral representation.

1 Introduction

Geometric models are a fundamental component in any graphics system. While there has been tremendous progress in the area of rendering over the past three decades, creating and acquiring high fidelity geometric models remains a challenging and tedious process.[1] Models are generally designed with high-end rendering in mind and can be difficult to modify and manipulate. Furthermore, as animation and simulation techniques become increasingly sophisticated and widely available, there is an increasing demand for models suitable for these purposes as well.

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, in which lighting, materials, objects, and even shading are specified procedurally [Hanrahan and Lawson 1990; Upstill 1990].

In this paper, we introduce a procedural modeling approach for authoring layered, solid models. We are especially interested in generating models that are suitable for both rendering and physical simulation. Just as computer graphics rendering systems provide a framework for light transport simulation, we envision an analogous framework for physical processes and other operators that modify and shape geometry.

There are many reasons to consider a procedural approach to surface creation and modification. A concise specification framework permits different simulation techniques — for example, ray tracing, radiosity, finite element method (FEM), and simplified spring-mass models — to be applied and compared. In addition, complicated processes can be described algorithmically. A procedural definition can be used as an intermediate format for capturing, editing, and replaying interactive editing sessions. It also provides a high-level abstraction, permitting a variety of different representations — for example, meshes and implicit functions — to coexist in the same environment, regardless of the underlying simulation system. Procedural models are advantageous in that they can be incrementally edited and refined based on artistic needs. Finally, powerful simulation tools, such as FEM or particle systems, can be embedded as modeling operators within such a procedural framework.

1.1 Related Work

Within traditional modeling systems, complex models are created by applying a variety of operations, such as constructive solid geometry(CSG) and freeform deformations, to a vast array of geometric primitives [Coquillart 1990; Payne and Toga 1992; Wyvill et al. 1999; Adzhiev et al. 1999]. In the hands of a talented artist, these systems produce intricate geometric models, but the process is extremely labor intensive. The range of tools available for specifying and editing shapes is also very limited. Surface representations can be locally deformed by simply modifying surface control points; however, tools for shaping geometry are rarely physically based, and the underlying geometry generally lacks information about the internal physical properties of the model, which would be necessary for creating complex deformations. In addition, such deformations can create self-intersections that are difficult to detect or prevent. Furthermore, performing topological changes to a model, such as drilling a hole through it, can be challenging using a surface description alone.

Another approach to creating models involves interactive sculpting, in which the user modifies a solid material with a tool [Wang and Kaufman 1995; Mizuno et al. 1998; Raviv and Elber 2000; Frisken et al. 2000]. Such systems are typically based on sampled volumetric representations (voxels or octrees), which can be costly to store and render interactively. Additionally, performing deformations within a grid-based representation requires shifting data over cell boundaries, which can be expensive and lossy. Unlike surfaces, which are merely hollow shells, volumetric representations can capture the internal material structure of a model. One of the main benefits of volumetric representations is that they support robust sculpting operations and simulations [Dorsey et al. 1999; O'Brien and Hodgins 1999]. However, volumetric models often lack visual fidelity because a high resolution volume is necessary to represent a complex model.

[1] The widespread use of the same small set of models, such as the Stanford bunny and the Utah teapot, attests to these difficulties.

3D 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. While such digitizers are useful for acquiring surface shape and appearance properties, they do not capture the internal structure of the geometry, which is often necessary for animation or simulation.

Procedural modeling techniques have proved to be valuable in several specific domains of computer graphics [Ebert et al. 1998]. Examples include plant modeling [Prusinkiewicz et al. 1988], solid texturing [Perlin 1985; Perlin and Hoffert 1989], displacement maps [Cook 1984], cellular texturing [Legakis et al. 2001], and urban modeling [Parish and Müller 2001]. One of the difficulties of procedural modeling is that the various techniques are domain specific. Additionally it can be difficult to precisely control the generation process to create a specific model. In our approach, we use a surface model as a starting point and use procedural techniques to generate a solid model. This provides a framework for the creation of a rich class of models, which are suitable for rendering and simulation.

1.2 Overview

Our procedural framework provides a controlled, systematic way to specify the geometric and material properties of a solid model and to vary these attributes as a function of time. We have developed a simple scripting language for authoring complex volumetric models and we show examples of its use. In our language, models are first initialized and then modified with a palette of physically inspired simulation operations. Model initialization is presented in Section 2 and the definition and use of simulation tools is described in Section 3. In Section 4 we present an implementation of the language using layered tetrahedral models, which we used to create the examples discussed in Section 5.

2 Model Specification

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. 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 through tomography or dissection approaches, but this can be inaccurate or destructive. Our modeling language is based on the observation that often the internal structure of an object can be inferred from a representation of its primary interface. Our basic building block is the *layered volume*. Within our framework, volumes can be combined, and layer composition can be controlled procedurally.

Through a series of examples based on a simple model of a chocolate candy, we show that our language provides a natural and expressive way to construct volumetric models. We include fragments of code from the scripts used to generate the images in this paper. As a convention in our examples, we use all capital letters to indicate user-defined functions and materials. A simplified grammar for the language appears in the appendix.

2.1 Layers of Material

To construct a volume with an interesting internal structure, we build layers of material from the primary surface. In our first example, we begin with a simple candy-shaped surface mesh and add two layers of material to the exterior and one layer to the interior (Figure 1).

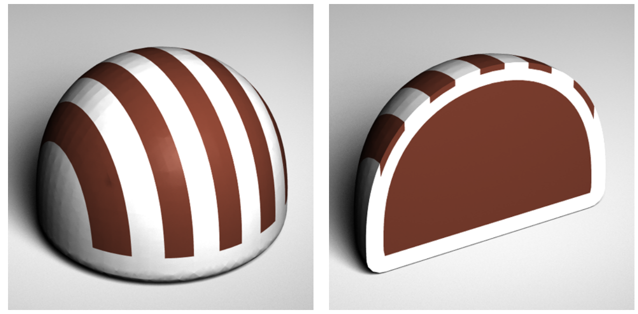


Figure 1: A chocolate candy created with two layers exterior to the original surface and one layer to fill the interior. The outermost layer has a procedural definition to create stripes of chocolate.

```
STRIPED_CANDY = volume {
  distance_field = surface_mesh {
    file = candy.obj }
  layers = {
    interior_layer {
      material = CHOCOLATE
      thickness = fill }
    exterior_layer {
      material = WHITE_CHOCOLATE
      thickness = 0.10 }
    exterior_layer {
      material = STRIPED_CHOCOLATE
      thickness = 0.05 } } }
```

Each layer has a material type and thickness. The type and thickness can be uniform or vary procedurally, which we discuss later. The thickness keyword `fill` can be used with a well-defined closed mesh 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.

2.2 Procedural Layer and Material Definitions

Materials are defined by a list of rendering and simulation parameters. We have a small library of built-in materials; additional materials can be defined within the script file as shown below. Default values are assigned to any unspecified parameters.

```
CHOCOLATE = material {
  color = { 0.31 0.17 0.15 }
  density = 1100   /* kg/m^3 */
  etc.  }
```

A layer need not be composed of a uniform material. The user can procedurally define a continuous variation of properties, such as wood grain or concrete particles, within a single material. This information can be used by a simulation and during rendering. Alternatively, a procedure can be used to subdivide the layer into distinct materials. Below is the specification used to create the striped layer of chocolate on the candy.

```
Material* STRIPED_CHOCOLATE(Vec3f &p) {
  if (p.y() < 0.2) return Lookup("WHITE_CHOCOLATE");
  if ((p.x() > -0.9 && p.x() < -0.7) ||
      (p.x() > -0.5 && p.x() < -0.3) ||
      (p.x() > -0.1 && p.x() <  0.1) ||
      (p.x() >  0.3 && p.x() <  0.5) ||
      (p.x() >  0.7 && p.x() <  0.9))
    return Lookup("CHOCOLATE");
  return Lookup("WHITE_CHOCOLATE"); }
```

Within a script, the user can define arbitrary C++ functions with default values for optional parameters. These functions are compiled and linked at runtime to interface with the core system. The `Lookup` function gives access to script file variable assignments. Function calls consist of the function name and a list of `name = value` pairs within curly braces. The arguments may appear out of order, or be left unspecified if optional.

2.3 Volume Specification

Many objects we would like to model are more complicated than simply layers of material constructed from a primary interface. Often these objects can be easily described as a collection of overlapping shapes. In our language, we use the precedence construct to combine volumes. In the example below, precedence is used to first create the volume for the almond, and then define the candy shape around the almond (Figure 2a). Subsequent shapes could be defined to fill the remaining unoccupied space.

```
ALMOND_CANDY = precedence {
  volume_1 = volume {
    distance_field = surface_mesh {
      file = almond.obj
    layers = {
      interior_layer {
        material = NUT
        thickness = fill } } }
  volume_2 = volume {
    distance_field = surface_mesh {
      file = candy.obj }
    layers = {
      interior_layer {
        material = CHOCOLATE
        thickness = fill }
      exterior_layer {
        material = WHITE_CHOCOLATE
        thickness = 0.10 }
      exterior_layer {
        material = STRIPED_CHOCOLATE
        thickness = 0.05 } } } }
```

The use of the precedence operator is particularly interesting when the surface meshes intersect. In Figure 2b 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 the outer layers to be wrapped around the protruding almond as shown in Figure 2c. To do this, we use a volume as the primary shape for a new volume. First, we use precedence to combine the almond shape with the interior layer of the chocolate. Then, we extract the outermost interface of the volume to use as the initializing surface for the second volume that adds the exterior layers of chocolate.

```
ALMOND_CANDY_2 = volume {
  distance_field = from_volume_surface {
    volume = precedence {
      volume_1 = volume {
        distance_field = surface_mesh {
          file = almond.obj }
        layers = {
          interior_layer {
            material = NUT
            thickness = fill } } }
      volume_2 = volume {
        distance_field = surface_mesh {
          file = candy.obj }
        layers = {
          interior_layer {
            material = CHOCOLATE
            thickness = fill } } } } }
  layers = {
    exterior_layer {
      material = WHITE_CHOCOLATE
      thickness = 0.10 }
    exterior_layer {
      material = STRIPED_CHOCOLATE
      thickness = 0.05 } } }
```

2.4 Signed Distance Field

Signed distance fields are a natural choice for describing and implementing the layers and volumes in our language. A signed distance field is a continuous scalar function defined throughout a volume, which can be used to compute offset isosurfaces while elegantly handling changes in topology and preventing self-intersection of the interfaces. In most cases, we initialize the distance field from a surface mesh using the method described in Section 4.2. Alternatively, we can create the field from an implicit surface or other

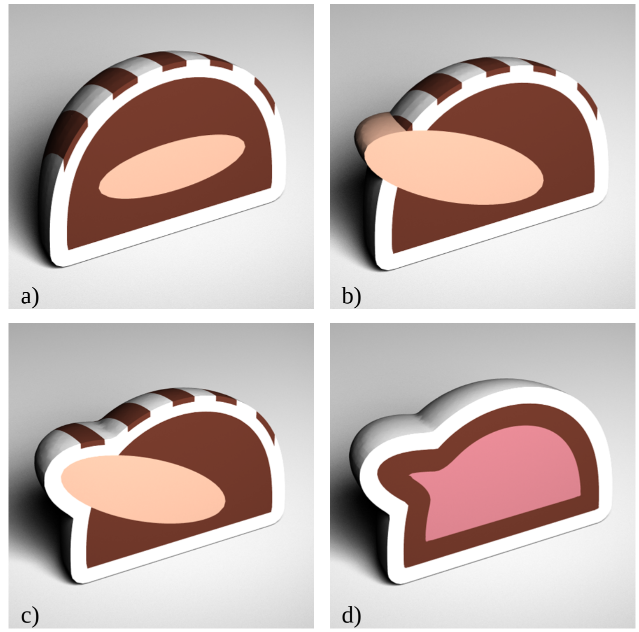


Figure 2: Specifying the interaction of two meshes allows many other possibilities: a) simple precedence to create the candy around an almond, b) & c) precedence with intersecting meshes, and d) union of the candy and almond meshes.

function. The layers of a volume are implemented as ranges of distance values.

Often the desired distance field is most easily described by combining distance fields using simple operators such as scaling, union (minimum), intersection (maximum), and subtraction [Ricci 1973; Frisken et al. 2000]. To demonstrate distance field composition, we use the union operator to combine the candy and almond surface meshes to produce the volume shown in Figure 2d. The zero isosurface of the resulting shape 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 } }
  layers = {
    interior_layer {
      material = CHOCOLATE
      thickness = 0.2 }
    interior_layer {
      material = PINK_FROSTING
      thickness = fill }
    exterior_layer {
      material = WHITE_CHOCOLATE
      thickness = 0.15 } } }
```

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 uniform thickness within each layer. However, it is often natural to describe layers that are thicker or thinner according to some pattern. To create interesting internal structures that have varying layer thicknesses, we can define non-Euclidean distance metrics by modifying the *interface 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 as shown in Figure 3a. Alternatively, the interface velocity can be defined procedurally: in Figure 3b the velocity is set by a random turbulence function, resulting in a bumpy appearance, and in Figure 3c a short procedure creates a diagonal swirl. The velocity can also be computed using visibility, accessibility, etc.

Interface velocity is implemented per distance field, and all layers within that field have a thickness pattern based on that velocity. Nesting volume specifications allows us to create a model with layers having different thickness patterns. For example, in Figure 3d we build a layer of bumpy frosting from a turbulent velocity field followed by a layer of foil wrapper with a diagonal pattern. This type of specification is common enough to warrant a syntactic sugar construct, which desugars velocities specified per layer into nested volume specifications.

```
LUMPY_CANDY = volume {
  distance_field = surface_mesh {
    file = candy.obj }
  layers = {
    exterior_layer {
      material = PINK_FROSTING
      thickness = 0.1
      velocity = BUMPY }
    exterior_layer {
      material = FOIL_WRAPPER
      thickness = 0.05
      velocity = DIAGONAL } } }
```

is equivalent to:

```
LUMPY_CANDY = volume {
  distance_field = from_volume_surface {
    volume = volume {
      distance_field = surface_mesh {
        file = candy.obj
        velocity = BUMPY }
      layers = {
        exterior_layer {
          material = PINK_FROSTING
          thickness = 0.1 } } }
    velocity = DIAGONAL }
  layers = {
    exterior_layer {
      material = FOIL_WRAPPER
      thickness = 0.05 } } }
```

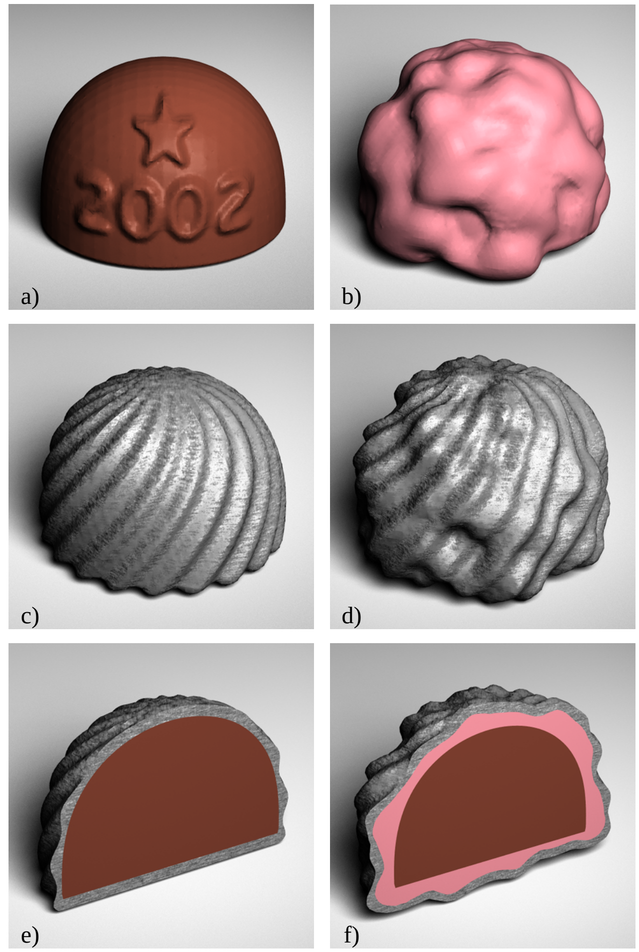


Figure 3: By modifying the interface velocity of the distance field, we can create layers with non-uniform thickness: a) painted velocity, b) turbulent velocity for a bumpy appearance c) procedurally created diagonal stripes, and d) a diagonal layer on top of the bumpy layer. Images e and f are cross-sections of c and d respectively.

3 Operations

In the previous section we discussed how our language is used to initialize a volumetric model. The advantages of these models become apparent when visualized and modified in complex ways. Many simulation techniques have been developed for sculpting and weathering [Dorsey et al. 1996; Dorsey et al. 1999; O'Brien and Hodgins 1999]. We have incorporated implementations of a few of these techniques into our system and provide user control of these tools through our language. The user is able to develop additional tools based on these packages or link to other simulation libraries.

3.1 Usability through Abstraction

One of the main obstacles the user must overcome in using a 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 first goal of our tool interface is to provide abstraction and standardization so the user of the tool can apply operations to the model without studying the details of the implementation. A simple interface between each simulation package and our system is established and a set of sample tools is created. Each tool defines default values for standard parameters such as position, orientation, size, and affected materials and calls one or more simulation packages. Using the sample tools as a guide, the user can create new tools.

We have linked our system to a flexible FEM simulation. We apply a distribution of forces to our model and the system computes the appropriate deformations and fractures. We can also control which materials are affected by the simulation; no other materials will be modified. Below we define a simple tool which applies a single hammer-like force to the model.

```
void HAMMER(Model *model,
            Vec3f position = Vec3f(0,0,0),
            Vec3f orientation = Vec3f(1,0,0),
            float magnitude = 1.0,
            float size = 1.0,
            List<Material*> *affects = NULL) {
  Vec3f force = orientation; force *= magnitude;
  AppliedArea *a = GaussSphere(position,size);
  FEM(model,a,force,affects); }
```

Below is an example use of this tool.

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

3.2 Defining Simulation Behavior

The power of a language for tool definition extends beyond copying and modifying existing tools. The language facilitates the specification of new types of behavior for the simulation. Particle systems have been used in many different applications to create a variety of effects that span a wide range of physical accuracy. The complexity of a particle system simulation depends on the definition of particle motion, interaction, and effects. Below we present the definition of a tool used to wash dirt from a statue.

```
void WASH(Model *model,
          int num_particles = 10000,
          float particle_life = 1) {
  Function *initialize = VerticalFall;
  Function *motion = Lookup("CLINGING");
  Function *action = Lookup("REMOVE_DIRT");
  ParticleSystem(model,num_particles,particle_life,
                 initialize,motion,action); }
```

The particle motion and action functions defined below each take two arguments: the particle to move, and the model with which it interacts. Motion functions that compute interactions between particles would also need the list of all particles as an argument.

```
void REMOVE_DIRT(Model *m, Particle *p) {
  Vec3f clean_color = Vec3f(1,1,1);
  List<Vertex*> vlist;
  float radius = 0.1;
  m->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); } }

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

In the `CLINGING` motion function, smaller values for the falling angle result in flow that behaves with greater surface tension.

3.3 Interactive Sculpting

Choosing the appropriate position, orientation, and radius for the types of tools described above can be tedious for complex models. Our language can also be used as an intermediate format for an interactive sculpting program. A simplified version of the volumetric model can be sculpted interactively and the actions saved. The logged actions can be edited by hand or simply appended to a script file that is run offline on the high resolution model.

4 Volumetric Representation

Our scripting language was designed to provide great freedom in model specification, independent of the underlying implementation of the volume data structures. In our implementation we use tetrahedral meshes to represent volumetric models. In this section we discuss some specifics of this implementation. Additionally, the system could maintain and convert between other volumetric representations that are more advantageous for certain operations.

4.1 Tetrahedral Mesh

Our volumetric representation consists of a set of tetrahedra, where each tetrahedron stores pointers to its four vertices and the four neighbors sharing its faces. Generally, neighbors are tetrahedra, but those tetrahedra with a face on the *visible interface* have a triangle neighbor that stores rendering information such as vertex normals and texture coordinates. The list of visible interface triangles forms a watertight mesh and is used for interactive display and offline rendering. Each tetrahedron stores its material type and any additional sub-tetrahedron material variations. We can also efficiently extract the set of faces that define the interfaces between different materials. These faces are necessary to accurately render refraction and translucency for non-opaque materials.

We have chosen a tetrahedral mesh because it offers many advantages in this application over other volumetric techniques, such as voxels or octree-based volumes [Wang and Kaufman 1995; Frisken et al. 2000]. With a tetrahedral mesh, we have a simple correlation between volume and surface, and the corresponding triangle mesh is easy to render on graphics hardware. The visible and interior interfaces can be represented at variable resolutions and model sharp creases in the geometry accurately. The data structure is inherently adaptive, allowing more tetrahedra in areas of high detail. Tetrahedral meshes are a simple extension of triangle meshes, and their geometric properties, such as simplification and subdivision, are well understood. Finally, many popular simulation techniques such as FEM are designed to work on tetrahedral meshes. Axis-aligned volumetric techniques such as voxels or octree-based distance fields are poorly suited to handle operations that deform or fracture the model.

4.2 Evaluating the Signed Distance Field

We synthesize tetrahedral models from triangle meshes by evaluating the signed distance field (discussed in Section 2.4) on a uniform 3D grid. The system determines a default grid based on the bounding box of the function or surface mesh, but it can be overridden by the user in the script file. We compute the distance value at each grid point using the Fast Marching Level Set method described by Sethian [1999], which elegantly avoids self-intersections when computing isosurfaces.

Given surface S, a 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. We initialize a band of *known* vertices near the original surface by iterating over the faces in the surface mesh and *rasterizing* each face F into the volume grid. For all grid points p near F, we update $f_S(p)$ iff $|f_F(p)| < |f_S(p)|$. To compute $f_F(p)$, the signed distance from point p to F, we 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 must be obtained by averaging the normals of adjacent faces.[2] After all faces have been rasterized, the function f_S is defined in the proximity of S. We propagate the distance of each known vertex to its neighbors, which are then marked *trial*. The trial vertices are stored in a priority queue by magnitude, and starting with the smallest distance, they are marked known and propagated to their neighbors until the necessary layer thickness has been defined.

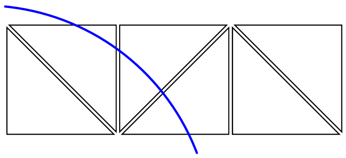
4.3 Tetrahedral Mesh Generation

Once the signed distance function has been initialized, we use a standard method for creating tetrahedral meshes — a *structured method* based on an axis-aligned grid or octree [Yerry and Shephard 1984; Wyvill et al. 1986; Lorensen and Cline 1987; Bloomenthal 1994]. 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 mesh. In our application, we handle large scanned meshes and matching the surface is

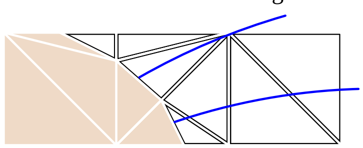
[2] Nooruddin and Turk [2000] present an alternative approach for obtaining the signed distance field which does not require a watertight mesh.

usually unnecessary and even undesirable. Both structured and unstructured mesh generation techniques are usually used in conjunction with mesh simplification and optimization, which we discuss in Section 4.4.

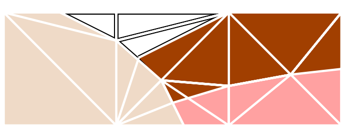
First each *cubic grid cell* is divided into five *tetrahedral cells*, alternating the orientation of the central tetrahedron so that diagonals match on neighboring cubic cells. We chose not to use the six-tetrahedron decomposition because it results in more tetrahedra and requires interpolation along the long diagonal of the cube, which can lead to additional artifacts on material interfaces. We illustrate our technique with a 2D example.



Each tetrahedral cell is then divided into tetrahedra of the appropriate materials, similar to Nielson et al. [1997]. 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, we split the tetrahedral cell into two cells by splitting one of its edges at an interface crossing (neighboring tetrahedral cells sharing that edge are also split) and recurse. A protocol for ordering edge splits based on vertex and interface identifiers guarantees a proper mesh with matching tetrahedral faces. This algorithm places no constraints on the thickness of layers or the number of interface crossings allowed per tetrahedral cell.



Using precedence to combine volume descriptions (Section 2.3) can introduce non-manifold interfaces [Bloomenthal and Ferguson 1995]. If a tetrahedral cell is not assigned material by the first volume description in a precedence operation, we proceed to the next volume description. When we split a tetrahedral cell, we also split tetrahedra assigned by previous volume descriptions that share the edge to be split. This operation prevents T-junctions at non-manifold interface intersections and is performed efficiently using a hash table of all tetrahedral edges.



After the volume has been tetrahedralized, tetrahedra for layers with procedural descriptions (Section 2.2) are subdivided as necessary to correctly assign materials.

4.4 Simplification of Models for Simulation

The structured mesh generation technique described in Section 4.3 produces a large number of tetrahedra and *poorly shaped* tetrahedra [Shewchuk 1998] when an interface passes very close to the grid points. Many simulation techniques require tetrahedra to be well-proportioned, which is often measured by the minimum solid angle [Fleischmann et al. 1999]. We have several methods to reduce the overall number of tetrahedra and improve their shape.

To obtain a high resolution interface, we require a high resolution grid; however, if a material layer is thick relative to the grid, this leads to extraneous tetrahedra within the layer. An adaptive octree approach dramatically reduces the initial number of tetrahedra produced, as illustrated in Figure 4. Similarly to Frisken et al. [2000], we compute the signed distance field on a uniform grid, then collapse grid cells that are accurately represented by interpolation or do not contain an interface crossing. We restrict the grid cell collapses such that cells sharing faces are no more than one level different in the octree. This restriction bounds the minimum solid angle of intermediate tetrahedral cells. Additionally, the user can specify that certain interfaces must be represented at a higher resolution and with more accuracy.

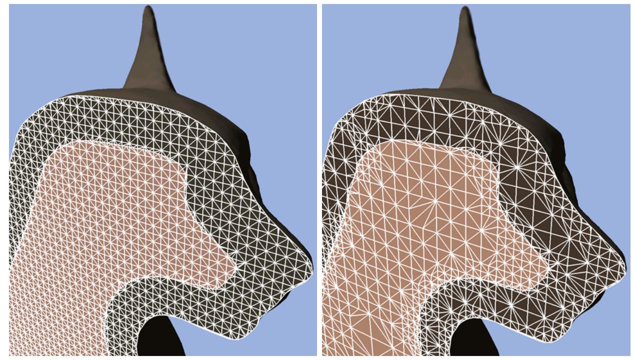


Figure 4: The mesh on the left was created from a uniform distance field. The mesh on the right was created from the same distance field after adaptive refinement, resulting in less than half as many tetrahedra. The meshes have similar interface quality. Simplification can be used to further reduce the size of the model.

After the initial tetrahedralization, we use a combination of simplification and mesh improvement techniques [Hoppe 1996; Staadt and Gross 1998; Trotts et al. 1999; Cignoni et al. 2000]. We found it difficult to define an appropriate edge collapse weighting function (used in the Progressive Mesh techniques) that simultaneously solved our goals. Our solution is similar to the mesh improvement strategy described by Freitag and Ollivier-Gooch [1997] and has been efficient and effective in practice.

First, we compute a quality metric (ranging from 0 to 10) for each tetrahedron t, which can vary depending on the exact requirements of the simulation we plan to run. The equations below reward tetrahedra that are close to equilateral (minimum solid angle $\sim$0.55 steradians) and have volume close to the ideal volume (total model volume / desired tetrahedral count). We use $\alpha = 0.7$.

$$\begin{aligned}
\text{Quality}(t) &= \alpha * \text{A}(t) + (1-\alpha) * \text{V}(t) \\
\text{A}(t) &= 10 * \min\left(1, \sqrt{2 * \text{min solid angle}(t)}\right) \\
\text{V}(t) &= 10 * \min\left(1, \sqrt{\frac{\text{volume}(t)}{\text{ideal volume}}}\right)
\end{aligned}$$

We target the removal or improvement of low-quality tetrahedra while maintaining the visible and interior interfaces (using, e.g., quadric error [Garland and Heckbert 1997] or volume preservation). Our simplification strategy is outlined in the following psuedocode.

```
for q = 0 to 10
    T = { all tetrahedra with Quality(t) ≤ q }
    for each t in T
        try these actions:
            • 3 → 2, 2 → 3, and 2 → 2 tetrahedral flips
            • half edge collapses
            • move each vertex to the average of its neighbors
```

We choose not to perform an action if the interface is unacceptably degraded, or if the minimum quality of the affected tetrahedra after the action is lower than the minimum quality before the action. If a

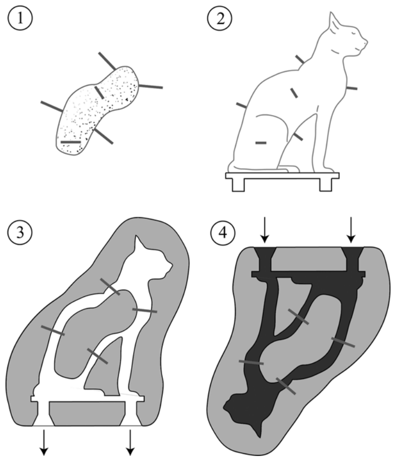


Figure 5: Lost wax casting. Adapted from Hodges [1970].

stopping criterion (such as a desired number of tetrahedra) has not been met, the interface requirements are reduced and the process is repeated.

5 Results

In this section we present three illustrative examples from our system. We describe our artistic intentions for each model based on its environment and history.

5.1 Lost Wax Casting

The *lost wax casting process* is a common technique for creating bronze statues (Figure 5). 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 hardened clay mold. Finally, when cool, the brittle clay is chipped away to reveal the bronze statue.

The original cat surface has sharp edges and areas of high curvature, 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, we use the convex hull of the original surface as a second mesh.

```
BRONZE_CAT = precedence {
  volume_1 = volume {
    distance_field = surface_mesh {
      file = cat.obj }
    layers = {
      interior_layer {
        material = BRONZE
        thickness = 1 }
      interior_layer {
        material = FIRED_CLAY
        thickness = fill } } }
  volume_2 = volume {
    distance_field = surface_mesh {
      file = cat_hull.obj }
    layers = {
      interior_layer {
        material = FIRED_CLAY
        thickness = fill }
      exterior_layer {
        material = FIRED_CLAY
        thickness = 2.5 } } } }
```

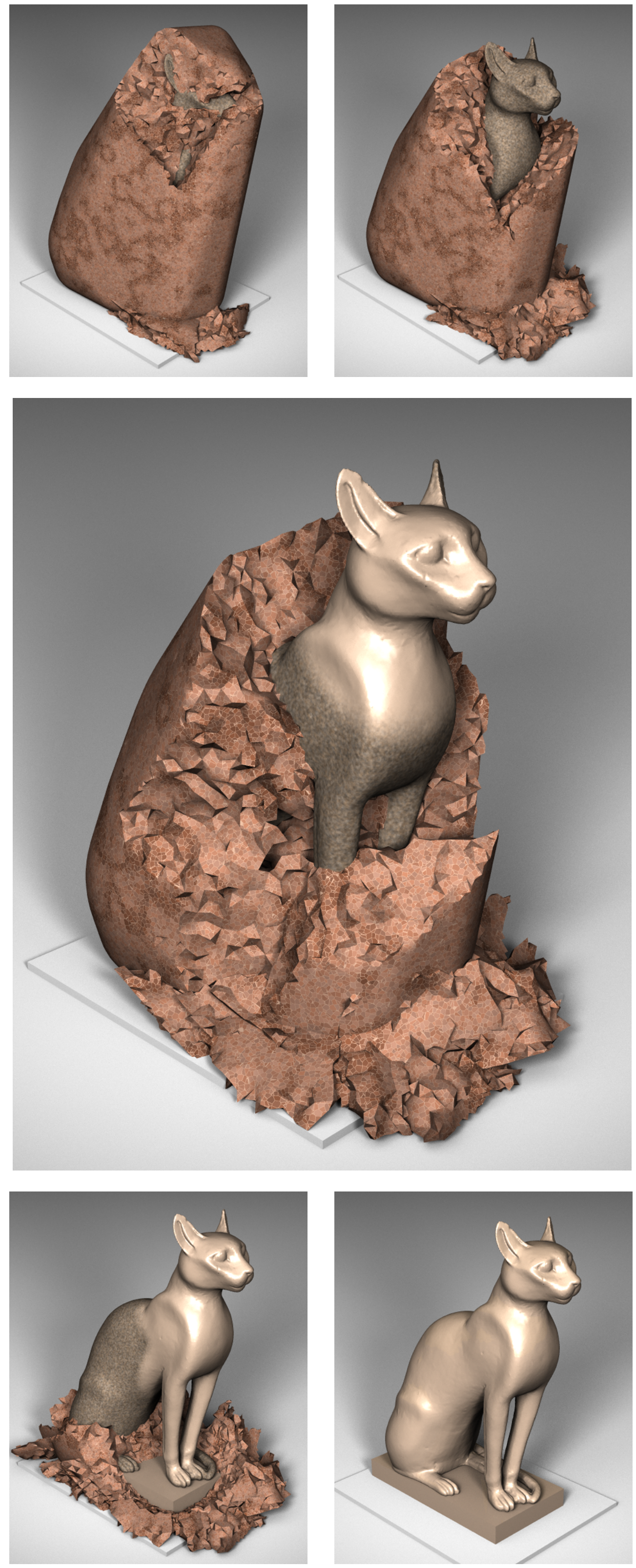


Figure 6: A sequence of images from our bronze statue simulation. The outer layer of fired clay is broken away using a hammer tool. A polish tool is used to clean and shine the model.

We used the hammer tool to break away the outer layer of clay, by specifying that only fired clay tetrahedra are affected. The tool is used repeatedly on different portions of the model. We also designed a polish tool to clean and shine the statue. This tool performs a CSG subtraction operation to remove clay left on or around the model. Subtraction is implemented in our system by subdivision and tetrahedron removal. 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); } }
```

We interactively sculpted a model of approximately 100,000 tetrahedra, and replayed the operations on a model with 300,000 tetrahedra [Müller et al. 2001]. A sequence from this simulation is shown in Figure 6.

5.2 Displaced Brick Paving

In our next example, we model a tree in an urban setting surrounded by brick paving. As the tree grows, the roots push upward, shifting the bricks. Here is the script we 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 } }
    layers = {
      interior_layer {
        material = TREE
        thickness = fill } } }
  volume_2 = volume {
    distance_field = GROUND_PLANE
    layers = {
      interior_layer {
        material = BRICK_PAVING
        thickness = 0.075 }
      interior_layer {
        material = DIRT
        thickness = 1.00 } } } }
```

We created an abstract tree model using our language: the trunk is represented with an implicit function for a cylinder plus turbulence, and the roots are procedurally created from a simple 2D sketch.

The brick paving is created with a procedural definition similar to the striped chocolate definition in Section 2.2. The simplified model has approximately 200,000 tetrahedra.

To displace the brick paving around the tree, we created a tool to translate upward the vertices of all tree tetrahedra. The FEM system is used to solve for the static equilibrium positions of the remaining vertices. The results are shown in Figure 7. The bricks maintain their rectilinear shape because the brick material has a large value for 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].

Figure 7: We simulate tree growth by translating all tree vertices upward and deforming the dirt and bricks around the roots.

5.3 Weathered Statue

In Figure 8, we show the layering of weathering effects on 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 building foundations. Long term exposure causes a variety of effects on exterior architectural details including discoloration, weakening, erosion, biological growth, and fracture due to the freeze/thaw cycle. The model shown here was created from a scanned mesh as one layer of stone.

We use several tools built on our particle system that use different procedures for particle motion and action. First, we apply an even layer of dirt to the model and use the wash tool to remove dirt according to rain flow. The FEM hammer tool is used to break off the ear and a corner of the wing. Next, an erosion tool moves particles toward exposed areas of the mesh where a small sphere of material is removed. Finally, we apply a biological growth tool similar to the wash tool, but with minimal particle motion, resulting in lichen-colored discoloration on the top-facing surfaces. Below is the script used to modify the model, which after simplification contained approximately 500,000 tetrahedra.

```
DIRT {
  model = GARGOYLE
  color = { 0.5 0.5 0.5 } }
WASH {
  model = GARGOYLE
  num_particles = 200000
  particle_life = 1.0 }
HAMMER {
  model = GARGOYLE
  position = { -0.78 1.22 0.77 }
  orientation = { -0.23 -0.47 0.85 } }
HAMMER {
  model = GARGOYLE
  position = { -2.53 1.03 1.06 }
  orientation = { 0.56 -0.19 -0.80 } }
ERODE {
  model = GARGOYLE
  num_particles = 2000 }
LICHEN {
  model = GARGOYLE
  num_particles = 40000 }
```

6 Discussion and Future Work

We have presented a procedural framework for specifying layered solid models and applying a series of simulation operations to them. Our 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 our high-level framework. These models can then be easily modified using procedural simulation tools.

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

Our system has been used to successfully construct models for a wide range of rendering, simulation, and animation applications. We have built small models, with a few hundred tetrahedra, for use in real-time animation research [Müller et al. 2001], as well as large models with millions of tetrahedra for off-line weathering and erosion simulations. In fact, models at either resolution can be constructed from essentially the same script.

In the future, we plan to expand our language to incorporate new modeling and simulation tools. We would like to alternate between the various phases of modeling and simulation more readily. We would also like to add better procedural support for volume generation, perhaps incorporating support for materials, such as cement-based products, which have intricate internal structures.

Overall, we believe that a procedural interface between modeling and simulation is an important tool for our community. With our prototype framework, we have experienced 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.

7 Acknowledgments

We would like to thank Hugues Hoppe for helpful discussions, Justin Legakis for the use of his rendering software, and Stephen Duck for the architectural model in the gargoyle renderings. This work was supported by NSF grants CCR-9988535, CCR-0072690, and EIA-9802220 and by a gift from Pixar Animation Studios.

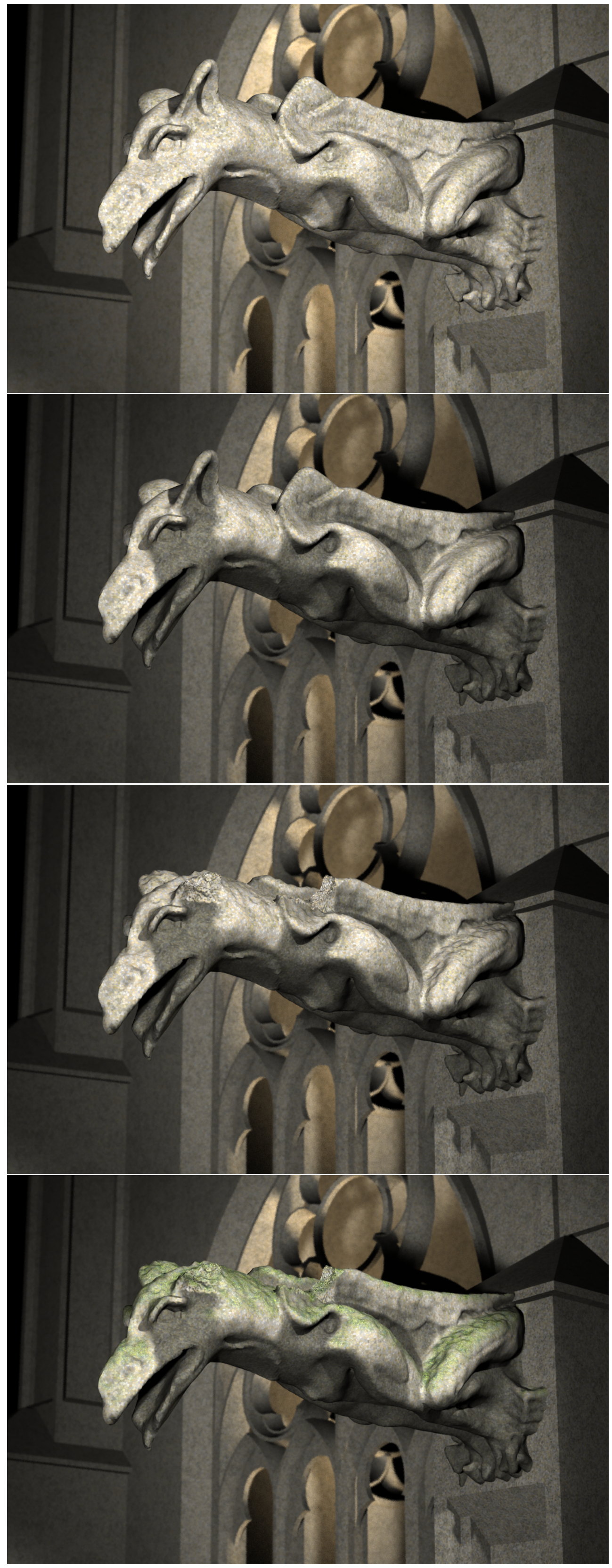


Figure 8: A sequence of renderings from the gargoyle simulation: the initial model made of fresh white stone; a layer of dirt is applied and partially washed away by rain; fracture removes the gargoyle's ear and wing, erosion affects top surfaces; biological growth.

A Modeling Language

Type grammar

script : (*assignment* | *operation*)*
assignment : *identifier* = *value*
operation : *function* { *assignment** }
value : *integer* | *float* | *string* | *material* | *layer* | *distance_field*
| *volume* | *function* | { *value** }
distance_field : function: *Vec3f* ⇒ *float*
substance : *material* | function: *Vec3f* ⇒ *material* | `nothing`
thickness : *float* | `fill`
velocity : *float* | function: *Vec3f* ⇒ *float*

Selected built-in functions

```
material material ( string name, float density = 1.0, etc. );
layer interior_layer ( substance material,
                       thickness thickness = 1.0 );
layer exterior_layer ( substance material,
                       thickness thickness = 1.0 );
distance_field surface_mesh ( string file,
                              velocity velocity = 1.0 );
distance_field union ( distance_field distance_field_1,
                       distance_field distance_field_2,
                       velocity velocity = 1.0 );
distance_field from_volume_surface ( volume volume,
                                     velocity velocity = 1.0);
volume load ( string file);
volume volume ( distance_field distance_field, layers layers);
volume precedence ( volume volume_1, volume volume_2);
```

References

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.

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

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

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.

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

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.

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

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.

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.

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.

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

FLEISCHMANN, P., KOSIK, R., HAINDL, B., AND SLBERHERR, S. 1999. Simple examples to illustrate specific finite element mesh requirements. In *Proceedings of the 8th International Meshing Roundtable*, 241–246.

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.

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.

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.

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.

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

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.

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.

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.

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

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.

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

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

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.

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.

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

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

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

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.

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.

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

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

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

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

TROTTS, I. J., HAMANN, B., AND JOY, K. I. 1999. Simplification of tetrahedral meshes with error bounds. *IEEE Transactions on Visualization and Computer Graphics*, 5(3), 224–237.

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

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

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

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.

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.