A Procedural Approach to Authoring Solid Models

Barbara Cutler Julie Dorsey Leonard McMillan Matthias Müller Robert Jagnow

Massachusetts Institute of Technology
Laboratory for Computer Science

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. Modeling operators, which may include simulations or sculpting operations, 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 demonstrate our language using a flexible tetrahedral representation. We show examples of our system interfacing with finite element and particle simulation tools to produce a variety of complex models.

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] In addition to the generation problem, models are also hard to modify and manipulate. Another difficulty is that models are generally designed with high-end rendering in mind. However, 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 paradigm 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 sculpt geometry.

There are many reasons to consider a procedural approach to surface creation and modification. A concise specification framework permits different simulation approaches – e.g. ray tracing, radiosity, finite element and simplified spring-mass models – to be applied and compared. In addition, complicated processes can be described algorithmically and codified. 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 – e.g. meshes and implicit functions – to coexist in the same environment, regardless of the underlying simulation system. Procedural models are also advantageous in that they can be incrementally edited and refined based on artistic needs. Finally, powerful simulation tools, such as finite element 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 modeling operations, such as CSG and freeform deformations, to a vast array of geometric primitives. While in the hands of a talented artist these tools are able to produce intricate geometric models, the model-creation 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 is 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 the model, such as drilling a hole through it, is challenging using just a surface description.

Another approach to creating models involves interactive sculpting, which is based on the notion of sculpting a solid material with a tool. Such systems are typically based on 3D grids; however, they can be costly to render, since they must be either ray traced or converted into a surface mesh using marching cubes [Lorensen and Cline 1987]. Additionally, deforming a grid-based representation is difficult since it involves expensive shifting of data over cell boundaries. One of the main benefits of volumetric representations is that they support robust sculpting operations and simulations [Wang and Kaufman 1995; Adzhiev et al. 1999; O'Brien and Hodgins 1999; Wyvill et al. 1999; Frisken et al. 2000].

Unlike surfaces, which are merely hollow shells, volumetric representations can capture the internal material structure of the model. However, volumetric models often lack 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 surface models, such as sculptures or mechanical parts, which would be difficult or impossible to create with interactive techniques. Such digitizers are useful for acquiring surface shape and appearance properties, but they do not capture the internal structure of the geometry, which is often necessary for animation or simulation.

Procedural modeling techniques have proven 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, which limits their generality. Additionally it can be difficult to control precisely the generation process to create a specific model. In our approach, we use a surface model, which may be acquired from a wide variety of sources, 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 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-based simulation operations, such as finite element methods and particle systems. 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

In the following sections we describe our language for procedurally authoring a volumetric model. Through a series of examples, we show that this is a natural way to construct and edit models. We have developed the following grammar for our language. The types of operations that can be applied to the model are explained in Section 3.

script : *model operations*
model : `model =` *volume*
volume : `load_volume { file =` *string* `}`
| `volume { distance_field =` *distance_field*
`layers =` *layers* `}`
| `precedence { volume_1 =` *volume*
`volume_2 =` *volume* `}`
distance_field : `surface_mesh { file =` *string* `}`
| `from_volume_surface { volume =` *volume* `}`
| `union { distance_field_1 =` *distance_field*
`distance_field_2 =` *distance_field* `}`

Procedure calls consist of the function name and a list of `name = value` pairs within curly braces. Values can be integers, floating point numbers, strings, or a list of values within curly braces. All procedures in the language are defined with default values for each argument. As a convention in our examples, we use all capital letters to indicate user-defined functions and materials

2.1 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 even the peel of a fruit. Building a physically-realistic model of any of these objects requires definition of the boundaries between materials and the variations within each material. Such a model could be created by an artist, but the process is time-consuming. The data could be obtained through dissection or tomography approaches, but this is also time-consuming. 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.

We will use a simple chocolate candy model to illustrate the power of our language. Our first example illustrates how we can define multiple layers both interior and exterior from the original surface. See Figure 1 a and b.

```
model = 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 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 keyword `nothing` can be used to describe a layer of air with no volumetric properties.

2.2 Signed Distance Field

Signed distance fields are a common volume type which are directly supported in our procedural definitions. They are useful for converting manifolds and meshes into volumes. They elegantly handle changes in topology and do not allow self-intersection of the interfaces. A signed distance field is a continuous scalar function defined over a volume. In most cases, we construct the distance field from a surface mesh using a method described in Section 4.2. Alternatively, we can create the field from an implicit surface or other function. Layers are specified as ranges of distance values.

Often 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, which have varying layer thicknesses, we can define non-Euclidean distance metrics using a modified *interface velocity*. In Figure 1 c and d the distance field interface velocity is set by a random turbulence function resulting in a bumpy appearance. Alternatively, the user may "paint" a pattern of increased velocity on the surface mesh, which will correspond to increased layer thickness in the signed distance field. A diagonal swirl of increased velocity was procedurally applied to the surface in Figure 1 e and f. The velocity can also be computed using visibility, accessibility, etc.

Often the desired distance field is most easily described by combining distance fields using simple operators such as scale, union and intersect (minimum and maximum), and subtract [Ricci 1973; Payne and Toga 1992; Frisken et al. 2000]. To demonstrate distance field composition, we union the candy surface mesh with an almond mesh to produce the model shown in Figure 1 g.

```
model = volume {
  distance_field = union {
    distance_field_1 = surface_mesh {
      file = almond.obj
      scale = 1.25
      rotate = { 0 0 1 -0.5 } }
    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 = 0.1 } } }
```

2.3 Volume Specification

A signed distance field, together with its layer list, is called a *volume specification*. Volume specifications can be combined by the precedence construct to yield another volume specification. In the example below, precedence is used to first create the volume for the almond, and then define the candy shape only within the unused volume (Figure 1 h). Subsequent shapes could be defined to fill the remaining unoccupied volume.

```
model = 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 1 i 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 those layers to also be wrapped around the protruding almond as shown in Figure 1 j. To do this, we use a volume specification to create a distance field. The outermost exterior interface is extracted from the volume and used as the initializing surface for a new distance field. Below is the script that created the model.

```
model = volume {
  distance_field = from_volume_surface {
    volume = precedence {
      volume_1 = volume {
        distance_field = surface_mesh {
          file = almond.obj
          scale = 1.25
          rotate = { 0 0 1 -0.5 } }
        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 } } }
```

In Section 2.2 we discussed how a modified interface velocity of the signed distance field can be used to vary layer thickness. Modified interface velocity is implemented per distance field, and all layers within that field will 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, we could first grow a layer of bumpy frosting from a random velocity field on our candy (as in Figure 1 c), followed by a layer of uniform thickness of white chocolate. This type of specification is common enough to warrant a syntactic sugar construct, which desugars velocities specified per layer into nested volume specifications.

```
model = volume {
  distance_field = surface_mesh {
    file = candy.obj }
  layers = {
    exterior_layer {
      material = PINK_FROSTING
      thickness = 0.2
      velocity = BUMPY }
    exterior_layer {
      material = WHITE_CHOCOLATE
      thickness = 0.1 } } }
```

is equivalent to:

```
model = volume {
  distance_field = from_volume_surface {
    volume = volume {
      distance_field = surface_mesh {
        file = candy.obj
        velocity = BUMPY }
      layers = {
        exterior_layer {
          material = PINK_FROSTING
          thickness = 0.2 } } } }
  layers = {
    exterior_layer {
      material = WHITE_CHOCOLATE
      thickness = 0.1 } } }
```

2.4 Procedural Layer and Material Definitions

A layer need not be composed of a uniform material. 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. The function body is C code which is compiled and linked into the system. (The brick paving used in Section 5 was generated with a similar definition.)

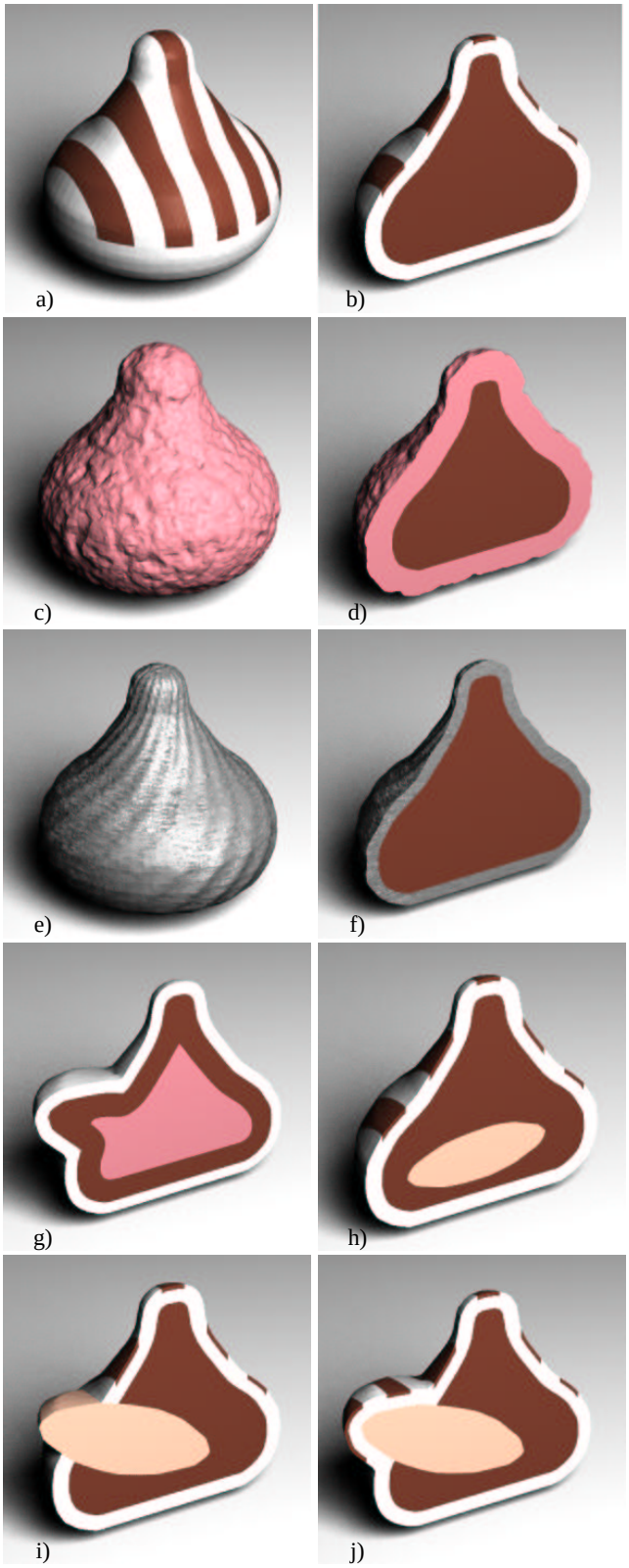

Figure 1: A sampling of the models that can be procedurally created from a simple candy surface mesh. By modifying the interface velocity of the distance field, we can create layers with non-uniform thickness, shown in c, d, e, and f. Specifying the interaction of two meshes allows many other possibilities, a few of which are shown in g, h, i and j.

```
define material STRIPED_CHOCOLATE {
  input = { x y z }
  function = {
    if (y < 0.6) return WHITE_CHOCOLATE;
    if ((x > -1.35 && x < -1.05) ||
        (x > -0.75 && x < -0.45) ||
        (x > -0.15 && x <  0.15) ||
        (x >  0.45 && x <  0.75) ||
        (x >  1.05 && x <  1.35))
      return CHOCOLATE;
    return WHITE_CHOCOLATE; } }
```

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

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

The user can also procedurally define a continuous variation of properties within a single material such as wood grain or concrete particles. This information can be used by the simulation and during rendering.

3 Operations

Many simulation techniques have been researched and 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 Useability through Abstraction

One of the main obstacles the user must overcome to use one of these packages is determining proper values for the numerous parameters needed to make the system run. Different implementations of the same simulation technique may require different sets of parameters. The first goal of our tool language 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 definition begins with a list of parameters and their default values. Then the tool calls one or more of the simulation packages through the interface created in the system. Standard tool parameters include position, orientation, size, and affected materials. Using the sample tools as a guide, the user can create new tools.

We have linked our system to a flexible finite element method (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.

```
define tool HAMMER {
  position = { 0 0 0 }
  orientation = { 1 0 0 }
  magnitude = 1
  size = 1
  affects = everything
  action = fem {
    affects = HAMMER.affects
    force = { HAMMER.magnitude *
              HAMMER.orientation }
    applied_area = gaussian_sphere
      { center = HAMMER.position
        radius = HAMMER.size } } }
```

3.2 Defining Simulation Behavior

The power of a language for tool definition extends beyond copying and modifying existing tools. The language allows us to specify new types of behavior for the simulation. Particle systems have been used in many different applications to create a variety of effects that span the range of physical accuracy. The complexity and accuracy 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.

```
define tool WASH {
  num_particles = 10000
  particle_life = 1
  action = particles {
    affects = everything
    num_particles = WASH.num_particles
    particle_strength = 1
    particle_life = WASH.particle_life
    particle_initialize = vertical_fall
    particle_motion = CLINGING
    particle_action = REMOVE_DIRT } }
```

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

```
define particle_action REMOVE_DIRT {
  input = { p mesh }
  action = parameter_modify {
    parameter = color
    move_value_towards = { 1 1 1 }
    applied_area = gaussian_sphere {
      center = p.position
      radius = 0.1 } } }

define particle_motion CLINGING {
  input = { p mesh }
  function = {
    n = mesh normal at p.position
    if dot(n,gravity) > cos(p.falling_angle)
      drip
    else
      move along mesh in the
      direction of gravity } }
```

In the pseudocode for the motion function above, smaller values for the `falling_angle` result in rain that behaves with greater surface tension.

3.3 Interactive Sculpting

Choosing the appropriate position, orientation and radius for the various tools described above can be tedious for complex models. Our language can be used as an intermediate format for an interactive sculpting program. A simplified version of the volumetric model is sculpted interactively and the actions are 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 our implementation.

4.1 Tetrahedral Mesh

Our internal volume 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. This list of visible interface triangles forms a watertight mesh and is useful 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 *interior interfaces* between different materials. These faces are necessary to accurately render refraction and translucency for non-opaque materials.

We use 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 to arbitrary resolution 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 the finite element method (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 Constructing Tetrahedral Models

We synthesize tetrahedral models from triangle meshes by evaluating the signed distance field (discussed in Section 2.2) on a uniform 3D grid. The system determines a default grid based on the bounding box of the function or surface mesh, which 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 [Sethian 1999]. Level Sets are an elegant way to avoid 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. First, we initialize a band of *known* vertices within ± 2 units of 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. After all faces have been rasterized, the function f_S is defined in the proximity of S.[2] 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.

Once the field 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]. First each *cubic grid cell* is divided into five *tetrahedral cells*, alternating the orientation of the central tetrahedron so that diagonals match on

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

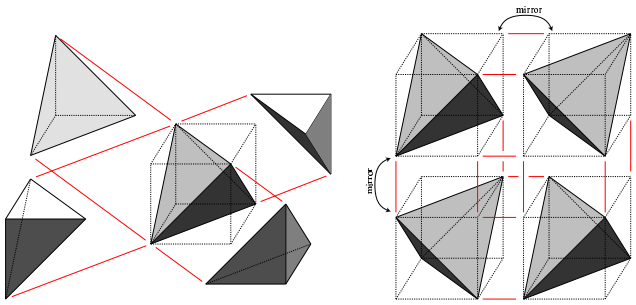

Figure 2: Illustration of how a cube is decomposed into five tetrahedra. This decomposition must be alternated so that the diagonals of neighboring cells align.

neighboring cubic cells (Figure 2). 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.

Each tetrahedral cell is then divided into tetrahedra of the appropriate materials using the set of cases enumerated by 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 an edge of the tetrahedron at an interface crossing, which splits all tetrahedral cells sharing that edge, and recurse. A simple ordering of edge splits based on vertex and interface identifiers guarantees a proper tetrahedral mesh with no T-junctions. The algorithm places no constraints on the thickness of layers or the number of interface crossings allowed per tetrahedral cell.

Additional subdivision is performed as necessary to correctly assign materials for layers with procedural definitions (Section 2.4).

4.3 Simplification of Models for Simulation

The tetrahedralization method described in Section 4.2 is simpler and more robust than other methods; however, it produces a large number of tetrahedra. Additionally, the axis-aligned technique produces *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 approach dramatically reduces the initial number of tetrahedra produced, as illustrated in Figure 3. Similarly to Frisken et al. [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 be 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.

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

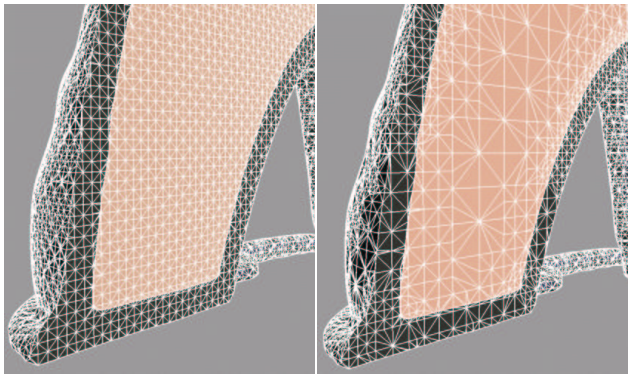

Figure 3: The mesh on the left was created from a uniform distance field and has ∼565,000 tetrahedra. The mesh on the right was created from the same distance field after adaptive refinement resulting in ∼388,000 tetrahedra. The meshes have similar interface quality. Simplification can be used to further reduce the size of the model.

strategy described by Freitag et al. [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 ∼0.54 steradians) and have volume close to the ideal volume (total model volume / desired tetrahedral count).

$$\text{Quality}(t) = 0.7 * \text{A}(t) + 0.3 * \text{V}(t)$$

$$\text{A}(t) = \min\left(10, 20 * \sqrt{\text{min solid angle}(t)}\right)$$

$$\text{V}(t) = \min\left(10, 10 * \sqrt{\frac{\text{volume}(t)}{\text{ideal volume}}}\right)$$

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

for q = 0 to 10
  T = { all tetrahedra t | Quality(t) $\leq$ q }
  foreach t in T
    try these actions:
      - $3 \rightarrow 2$, $2 \rightarrow 3$, and $2 \rightarrow 2$ 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 stopping criteria (such as a desired number of tetrahedra) has not been met, we reduce the interface requirements and repeat.

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.

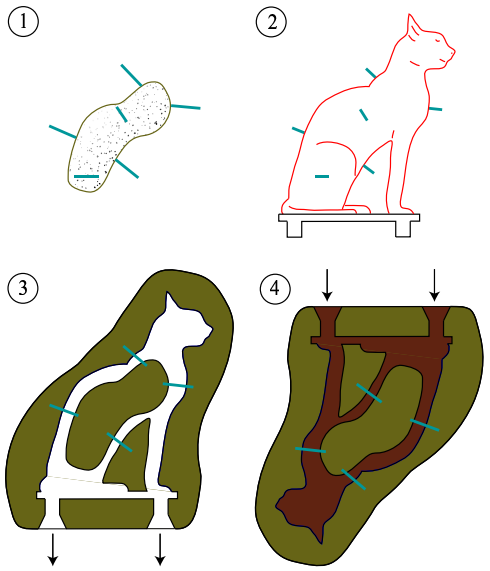

Figure 4: The lost wax casting process is used to create a bronze statue.

5.1 Lost Wax Casting

The *lost wax casting process* is a common technique for creating bronze statues (see Figure 4). 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 clay layers do not contain such detail. In the physical process, the artist applies a thicker layer of clay to the areas that are less accessible. To model this process, we use the convex hull of the original surface as a second mesh.

```
model = 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 } } } }
```

We used the hammer tool to crack off the outer layer, by specifying that only fired clay tetrahedra are affected. The tool is used repeatedly on different portions of the model. We designed a polish tool to clean and shine the statue. The tool calls two volumetric packages. First, the tool performs a Constructive Solid Geometry (CSG) subtraction operation to remove clay left on or around the model. Subtraction is implemented in our system by subdivision

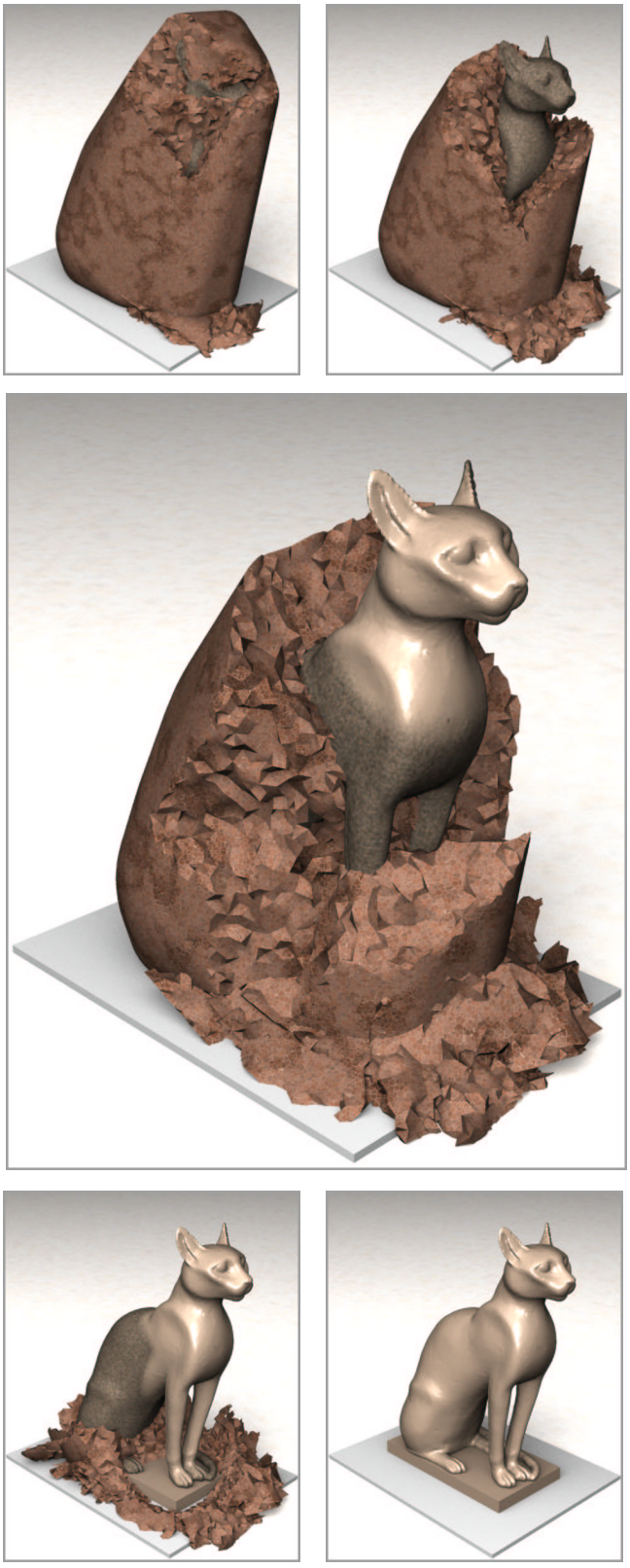

Figure 5: A sequence of images from our bronze statue simulation. The outer layer of fired clay is cracked off using a hammer tool. A polish tool is used to clean and shine the model.

Figure 6: The roots for our tree example are created from a 2D image.

and tetrahedron removal. Then a parameter modify action locally increases the shininess of the model. Parameter modify is implemented by collecting all tetrahedra within the applied area, and then modifying a value as indicated.

```
define tool POLISH {
  position = { 0 0 0 }
  size = 1
  action = csg {
    affects = FIRED_CLAY
    action = subtract
    applied_area = sphere
      { center = POLISH.position
        radius = POLISH.size } }
  action = parameter_modify {
    affects = BRONZE
    applied_area = gaussian_sphere
      { center = POLISH.position
        radius = POLISH.size }
    parameter = shininess
    move_value_towards = 100 } }
```

We interactively sculpted a model of ∼100,000 tetrahedra, and replayed the operations on a ∼300,000 tetrahedra model. A sequence from this simulation is shown in Figure 5.

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 brick. Here is the script we used to produce our initial model.

```
model = 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 (see Figure 6). The tree trunk is represented with an implicit function for a cylinder plus turbulence. The tree roots are procedurally created from a simple 2D sketch. The simplified model has ∼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 [Müller et al. 2001]. The results are shown in Figure 7. The bricks maintain their rectilinear shape because the brick material has a large value for the Young's Modulus elasticity parameter. The dirt between and beneath the bricks deforms easily because it has a small value for this parameter. 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 the 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. Our model was created from a scanned mesh with one layer of stone.

We use several tools built on our particle system that use procedures for particle motion and action. First, we apply an even layer of dirt to the model and then use the wash tool to remove dirt according to rain flow. The FEM hammer tool is used to crack 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 create the model, which after simplification con-

tained ∼500,000 tetrahedra.

```
DIRT {
  color = { 0.5 0.5 0.5 } }
WASH {
  num_particles = 200000
  particle_life = 1.0 }
HAMMER {
  position = { -0.78 1.22 0.77 }
  orientation = { -0.23 -0.47 0.85 } }
HAMMER {
  position = { -2.53 1.03 1.06 }
  orientation = { 0.56 -0.19 -0.80 } }
ERODE {
  num_particles = 2000
  particle_life = 0.01 }
LICHEN {
  num_particles = 40000
  particle_life = 0.1 }
```

6 Discussion and Future Work

We have presented a procedural framework for specifying volumetric 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 implicit functions in three dimensions, such as 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-scale models, with a few hundred tetrahedra, for use in real-time animation research, as well as large-scale models with millions of tetrahedra for off-line weathering and erosion simulations. In fact, consistent models at either scale 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 seamlessly. We would also like to add better procedural support for volume generation, perhaps enabling the modeling of biological growth.

Overall, we believe that a procedural interface between modeling and simulation is an important missing tool in 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 Acknowledgements

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.

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.

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.

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

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.

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.

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.

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.

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.