Using Particles to Sample and Control More Complex Implicit Surfaces

John C. Hart, Ed Bachta, Wojciech Jarosz, Terry Fleury
University of Illinois
{jch|bachta|wjarosz|tfleury}@uiuc.edu

Abstract

In 1994, Witkin and Heckbert developed a method for interactively modeling implicit surfaces by simultaneously constaining a particle system to lie on an implicit surface and vice-versa. This interface was demonstrated to be effective and easy to use on example models containing a few blobby spheres and cylinders. This system becomes much more difficult to implement and operate on more complex implicit models. The derivatives needed for the particle system behavior can become laborious and error-prone when implemented for more complex models. We have developed, implemented and tested techniques for automatic and numerical differentiation of the implicit surface function. Complex models also require a large number of parameters, and the management and control of these parameters is often not intuitive. We have developed adapters, which are special shape-transformation operators that automatically adjust the underlying parameters to yield the same effect as the transformation. These new techniques allow constrained particle systems to sample and control more complex models than before possible.

1 Introduction

Witkin and Heckbert [14] revolutionized implicit surface modeling by using a particle system to both display and control an implicit surface. Their treatment used a real function $F : \mathbf{R}^3 \times Q \to \mathbf{R}$ over model space $\mathbf{R}^3$ and a continuous parameter space Q. This real function yielded an implicit surface as the solution points x such that $F(\mathbf{x}, \mathbf{q}) = 0$ for a fixed, given vector of parameters $\mathbf{q} \in Q$.

A particle $\mathbf{p}^i$ constrained to the implicit surface of F such that $F(\mathbf{p}^i, \mathbf{q}) = 0$ is called a *floater*. This constraint is enforced by setting its original velocity $\dot{\mathbf{P}}^i$ to a legal velocity $\dot{\mathbf{p}}^i$ by subtracting any illegal components normal to the implicit surface

$$\dot{\mathbf{p}}^i = \dot{\mathbf{P}}^i - \frac{F^i_{\mathbf{x}} \cdot \dot{\mathbf{P}}^i + F_{\mathbf{q}} \cdot \dot{\mathbf{q}} + \phi F^i}{F^i_{\mathbf{x}} \cdot F^i_{\mathbf{x}}} F^i_{\mathbf{x}}. \qquad (1)$$

(Note that here $\dot{\mathbf{P}}$ denotes the desired velocity instead of $\mathbf{P}$ [14].) These illegal components are due to either particle velocities ($F^i_{\mathbf{x}} \cdot \dot{\mathbf{P}}^i$) or parameter velocities ($F_{\mathbf{q}} \cdot \dot{\mathbf{q}}$) that change the resulting value of F. Hence we need the derivative of F with respect to both its embedding $F_{\mathbf{x}}$ and its parameterization $F_{\mathbf{q}}$.

The constrained particle system displayed the implicit surface with a collection of disks centered at the particles oriented according to the surface normal. These oriented disks provide a usable and highly responsive display of the underlying implicit surface, and also yield a quasi-volumetric display of the surface that reveals interior structure in the gaps between disks. The visual edge noise created by the disks can sometimes be distracting, but this can be overcome by connecting the particles into a polygonization using a topological guarantee [12].

Some floater particles can be selected as control particles, which means the implicit surface is constrained to pass through these particles. Control particles can be dragged to new locations, and the implicit surface deforms to accomodate its new position. This deformation occurs by changing the parameters $\mathbf{q}$ of the implicit surface using the parameter velocity

$$\dot{\mathbf{q}} = \dot{\mathbf{Q}} - \sum_j \lambda^j F^j_{\mathbf{q}}. \qquad (2)$$

The index j is the index of the control particle, and $\dot{\mathbf{Q}}$ is the desired unconstrained parameter velocity. The λ^j are Lagrange multipliers found by solving the system

$$\sum_j \left(F^i_{\mathbf{q}} \cdot F^j_{\mathbf{q}}\right) \lambda^j = F^i_{\mathbf{q}} \cdot \dot{\mathbf{Q}} + F^i_{\mathbf{x}} \cdot \dot{\mathbf{p}}^i + \phi F^i. \qquad (3)$$

The desired parameter velocity $\dot{\mathbf{Q}}$ is usually zero, such that (2) and (3) dissipate control particle velocities $\dot{\mathbf{p}}$ into parameter velocities $\dot{\mathbf{q}}$.

Witkin and Heckbert [14] alluded to an object-oriented implicit surface class hierarchy, where new implicit model objects need to implement $F, F_{\mathbf{x}}, F_{\mathbf{q}}$ and a bounding box (to keep the particles from following a non-compact manifold indefinitely). Most implicit modeling systems implement the function, its gradient and a bounding box, so the only

new information needed is the derivative of the implicit surface function with respect to its parameters.

The goal of the work described in this paper is to integrate the particle system into a full-featured implicit surface modeling system. Section 2 reviews some previous implicit surface modeling systems. To our knowledge, no one has yet implemented such an object hierarchy in a full-featured implicit modeling system based on the control particle interface.

We have developed such an object-oriented class hierarchy specifically for inclusion into a particle-based modeling system. Section 3 describes the flexibility of our system that includes a large variety of implicit surface primitives and operators.

Witkin and Heckbert [14] suggest (once F_q is implemented) the application of particle-based interaction to complex hierarchical implicit surface models is straightforward. In the process of implementing such a system we have found several setbacks, specifically in implementing and debugging the function derivatives $F_{\mathbf{x}}$ and $F_{\mathbf{q}}$, and managing large numbers of parameters for manipulating complex composite implicit surface models, as demonstrated by the 30+ parameters shown in Figures 1 and 2.

One of the challenges of implementing a full-featured implicit surface representation for a particle-based modeler is the development of the derivatives needed. The particle-based modeler requires each primitive and operator to have a derivative with respect to its embedding space and its parameters. Section 4 describes several methods available to ease the programming of these derivatives.

Complex hierarchies of implicit operators and primitives quickly grow to become overwhelming. Section 5 describes methods for modeling that use special operators called adapters that reparameterize a complex implicit model. This reparameterization simplifies the modeling process by providing more intuitive parameters to the user.

2 Other Complex Implicit Surface Modeling Systems

This paper develops a modeling system capable of creating complex models of a wide variety of implicit primitives and operators using a particle system for display and control. Several other modeling systems have been previously developed to create complex implicit surface models, but each of these system has been limited to a specific subset of implicit primitives and operations.

Witkin and Heckbert [14] used a prototype implemention of a particle-based implicit surface modeler that proved the concept. Their implementation was never released, and only supported small collections of a few primitives, including blobby spheres and cylinders. Pedersen [8] later released a more robust, flexible and freely available particle system based on (1) of floater particles for displaying implicit surfaces, but it did not include an implementation of (2) and (3) that provided the control particles needed to interactively change the surface parameters. Stander and Hart [12] later described how to use interval analysis and Morse theory to connect surface constrained floater particles into a polygonization. Their implementation was also a proof-of-concept prototype that was never released. It was limited to axis-aligned Gaussian ellipsoids, and the dynamic mesh that connected the particles into a polygonization was very fragile.

The Blob Tree integrated multiple implicit modeling tools into a single scene description hierarchy for implicit surface modeling [15]. The Blob Tree was based on piecewise-polynomial blobby sphere primitives that blend when placed in proximity to each other. The Blob Tree organized objects into groups and groups can be combined with a smooth surface blend or a creased CSG operation. The blend tree also supported other implicit surface operations, for example non-linear deformations.

The Hyperfun system supported collaborative modeling of implicit surfaces [1]. Hyperfun is based on R-functions which generalize blending between primitives to include non-blended CSG operations. Its components include a scene description hierarchy, a declarative language for defining functions, new file formats for communicating implicit surface descriptions, and integration of the results into common graphics systems such as POVRay.

Implicit surfaces are also supported by the VTK toolkit [10, 11]. These implicit surfaces could be used a sources for volumetric "structured-points" data pipelines. These pipelines can be made arbitrarily complex, and have been used to model context geometry to aid in the visualization of acquired volumetric data [9]. Implicit surface sources are integrated into the VTK toolkit by defining the function and its gradient, though the VTK tools typically sample the source into a volume before performing any further processing.

3 A General Implicit Object Model

Many previous implicit surface modeling systems have focused only on a subset of the available implicit surface models. Our goal is to design a general full-featured implicit surface system that could support any implicit surface representation or operation.

The `Implicit` class sits at the root of our implicit surface hierarchy, and contains three key member functions. The member function `proc`($\mathbf{x}$) returns the scalar result $F(\mathbf{x}, \mathbf{q})$. The member function `grad`($\mathbf{x}$) returns the 3-vector result $F_{\mathbf{x}}(\mathbf{x}, \mathbf{q})$. The member function `procq`($\mathbf{x}, \mathbf{dq}$) puts the derivative $F_{\mathbf{q}}(\mathbf{x}, \mathbf{q})$ into the vector $\mathbf{dq}$. Each of these functions assumes $\mathbf{q}$ is constant and held

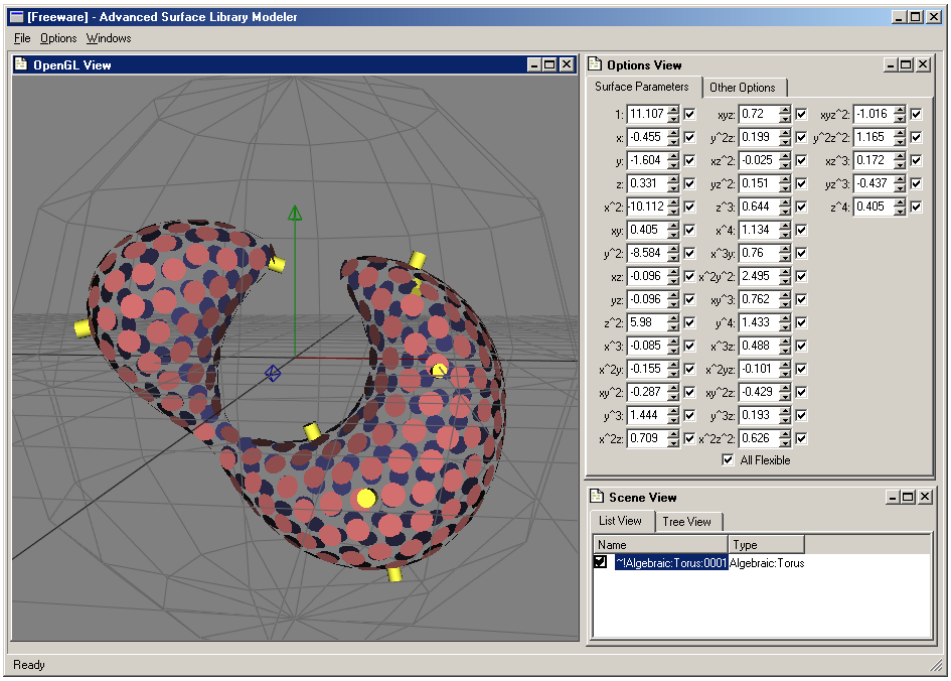

Figure 1. A sample screen of our particle-based modeler directly manipulating a quartic surface. This quartic surface has 35 continuous parameters shown on the right that can be entered individually or selected to be set by the controller particle constraints.

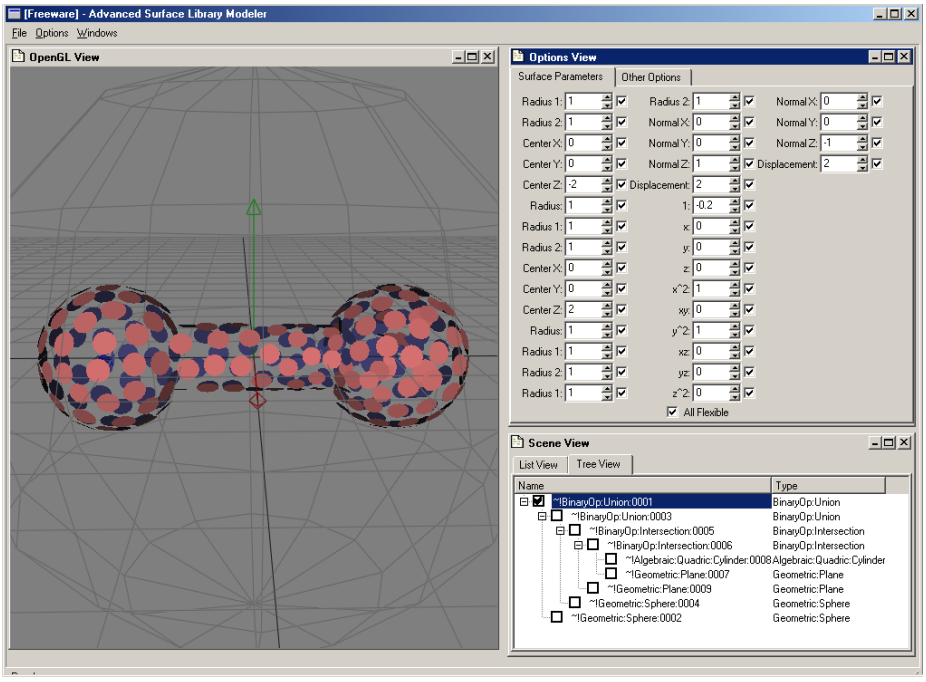

Figure 2. The particle-based modeler directly manipulating a dumbell modeled using R-functions. This composite surface has 34 continuous parameters shown on the right that can be entered individually or selected to be set by the controller particle constraints.

within the `Implicit` as part of its internal state.

We provide member access to the continuous parameters through an interface uniform across all classes derived from `Implicit`. The member function `getq(q)` puts the current parameters in vector $\mathbf{q}$, whereas the member `setq(q)` sets the current parameters to those in vector $\mathbf{q}$. This parameter access allows the particle system to query and adjust the continuous parameters as necessary to apply the floater and control particle constraints. The actual implementation of a specific implicit object need not store its parameters in a vector, and can provide additional specialized access methods to its parameters[1].

We have also implemented implicit surface operators (e.g. offset, scale, union, etc.). These operators affect the inputs and/or the outputs of one or more operand implicit objects. The `getq` and `setq` functions for the operators place the operator's parameters at the beginning of the vector, and follow them with the parameters of its operands. Hence, one can access all of the parameters of a hierarchically-defined implicit object through the `getq`/`setq` interface of the root operator object in the hierarchy.

4 Differentiation

One of the main obstacles in our implementation of a variety of implicit surface models has been in the derivation, implementation and debugging of derivatives of the functions.

4.1 Operator-Level Automatic Differentiation

The derivatives of operators, namely `grad` and `procq`, are implemented using use the chain rule, eventually calling `grad` and `procq` of the operands. The operations can be considered an arithmetic on implicit objects, which makes the chain-rule `grad` and `procq` implementations a partial form of automatic differentiation,

One could define arithmetic operations as Implicit operators. For example, the operator `Times` implements the function FG as

```
Times::proc(x) {
  return F->proc(x) * G->proc(x);
}
Times::grad(x) {
  return F->grad(x)*G->proc(x) +
         F->proc(x)*G->grad(x);
}
```

[1]For example, we have derived an `Algebraic` class from Implicit that provides access to the $d^3/6 + d^2 + 11d/6 + 1$ coefficients of a degree d trivariate polynomial. The coefficients are returned sorted by degree, then by x, y and z exponents. However, access to these coefficients is much easier by providing the exponents of x, y and z than providing a single coefficient index.

Given a sufficient set of these arithmetic operations, one could implement any implicit model. Implicit models constructed by composing these operators would have their differential functions automatically defined, since these objects compute their own derivatives. However, such an implementation would be cumbersome and inefficient.

Rather than implement high-level shape operators using automatically differentiated low-level arithmetic operators, we chose instead to automatically differentiate the high-level shape operators. A good example is our implementation of an abstract `Blend` class. Our blend class assumes the blend combines the isocontours of its operand implicit objects F and G [6]. The blend is thus defined by a real bivariate function $h(f, g)$ of the values f, g returned by the operands. The blend surface is thus implemented as

```
Blend::proc(x) {
  return h(F->proc(x),G->proc(x));
}
```

Specific blends are derived from the `Blend` class, and define the pure virtual function h. The blobby model can blend arbitrary implicit surfaces using the function

$$h(f, g) = T - e^{-f-a} - e^{-g-b} \tag{4}$$

where T, a and b control the "blobbiness" of the blend [4]. The superelliptical blend is given in this form as

$$h(f, g) = \frac{(f-a)^d}{a^d} + \frac{(g-b)^d}{b^d} - 1 \tag{5}$$

where a and b are continuous parameters controlling the extent of the blend and the discrete parameter d is the degree of the blending function [6]. R-functions provide a continuous neighborhood for CSG operations, and are given by

$$h(f, g) = (f + g + s\sqrt{f^2 + g^2})(f^2 + g^2)^{d/2} \tag{6}$$

where the discrete parameters $s = \pm 1$ differentiates between union and intersection, and d again provides a degree of smoothness [7].

Using the chain rule, we define

```
Blend::grad(x) {
  return hf(F->proc(x),G->proc(x))*F->grad(x) +
         hg(F->proc(x),G->proc(x))*G->grad(x);
}
```

using the partial derivative methods `hf` and `hg`. Thus, classes derived from `Blend` need not implement `proc`, `grad` and `procq`, but must implement the simpler method `h` and its partial derivatives `hf` and `hg`.

4.2 Code-Level Automatic Differentiation

Given a `proc` implementation, another technique for automatically generating the derivatives `grad` and `procq` is

to apply the automatic differentiation method to the `proc` procedure source code. Computational differentiation is an automatic differentiation applied to algorithms by declaring some variables as dependent and others as independent, and synthesizing the source code necessary to yield the derivatives of the dependent variables with respect to the independent variables. For example, recent tools exist (ADOL-C,ADIC) that differentiate C language source code [5, 3]. Performing automatic differentiation at compile time yields faster derivatives than automatically differentating at run time.

4.3 Numerical Differentiation

We can also use numerical techniques to evaluate the derivatives. Forward differencing of the spatial derivative is implemented as

$$F_{\mathbf{x}}(\mathbf{x},\mathbf{q}) = \begin{pmatrix} F(\mathbf{x}+\epsilon\mathbf{e}_0,\mathbf{q}) - F(\mathbf{x},\mathbf{q}), \\ F(\mathbf{x}+\epsilon\mathbf{e}_1,\mathbf{q}) - F(\mathbf{x},\mathbf{q}), \\ F(\mathbf{x}+\epsilon\mathbf{e}_2,\mathbf{q}) - F(\mathbf{x},\mathbf{q}) \end{pmatrix} \quad (7)$$

where $\mathbf{e}_i$ is a unit vector in the ith dimension direction. Using this notation, the parameter derivative can be similarly derived

$$F_{\mathbf{q}}(\mathbf{x},\mathbf{q}) = (\ldots, F(\mathbf{x},\mathbf{q}+\epsilon\mathbf{e}_i) - F(\mathbf{x},\mathbf{q}), \ldots)\,. \quad (8)$$

We have found that the constrained particle system remains stable even when numerical versions of $F_{\mathbf{x}}$ and $F_{\mathbf{q}}$ are used. The symbolic $F_{\mathbf{x}}$ runs about four times as fast as the numerical $F_{\mathbf{x}}$ because the forward differencing implementation calls the implicit surface function four times. Similarly, the symbolic $F_{\mathbf{q}}$ implementation is $|\mathbf{q}|$-times faster than its numerical version.

The virtual methods `grad` and `procq` of our `Implicit` object default to the forward differences approximations. Hence, we can add a new implicit surface model into our library as a black-box by implementing the method `proc`. This task is a less daunting than deriving $F_{\mathbf{q}}$ by hand, as has been previously suggested [14].

5 Parameterization

A second problem with using particles to manipulate complex implicit surface models is the management of the parameters $\mathbf{q}$ of the implicit surface model. This problem can be decomposed into two specific issues. The first issue is the conceptual disconnection between an object's intuitive parameters (such as location and orientation) and its actual parameters (such as the coefficients of an algebraic). The second issue is that there are often an overwelmingly large number of free parameters, even for a moderately complex implicit surface model.

One problem we have found is that it is difficult to translate the ellipsoid

$$\begin{aligned} F(x,y,z,q_0,\ldots,q_9) \quad = \quad & q_4x^2 + q_5xy + q_6y^2 + \\ & q_7xz + q_8xy + q_9z^2 + \\ & q_1x + q_2y + q_3z + q_0. \end{aligned} \quad (9)$$

using the control particles. We have the ability to select which parameters the control particles affect, but identifying which of the ten quadric coefficients control translation is not intuitive.

In order to translate the ellipsoid by $\mathbf{o} = (o_x, o_y, o_z)$, we need to apply the domain transformation $F(x-o_x, y-o_y, z-o_z, \mathbf{q})$. Evaluating (9) and collecting terms shows that translation does not affect the parameters q_4 through q_9, but affects q_0 through q_3 as

$$\begin{aligned} q_0 \quad \leftarrow \quad & q_0 + q_4o_x^2 + q_5o_xo_y + q_6o_y^2 + \\ & q_7o_xo_z + q_8o_yo_z + q_9o_z^2 + \\ & q_1o_x + q_2o_y + q_3o_z, \end{aligned} \quad (10)$$

$$q_1 \quad \leftarrow \quad q_1 - 2q_4o_x - q_5o_y - q_7o_z, \quad (11)$$

$$q_2 \quad \leftarrow \quad q_2 - q_5o_x - 2q_6o_y - q_8o_z, \quad (12)$$

$$q_3 \quad \leftarrow \quad q_3 - q_7o_x - q_8o_y - 2q_9o_z. \quad (13)$$

Enabling only these four coefficients to be changed by the control particles actually causes the entire ellipsoid to deform instead of translate because (2) and (3) dissipate particle velocity evenly among the parameter velocities. The velocities of the ellipsoid parameters due to translation are in fact $\dot{\mathbf{q}} = (q_1o_x + q_2o_y + q_3o_z + q_4o_x^2 + q_5o_xo_y + q_6o_y^2 + q_7o_xo_z + q_8o_yo_z + q_9o_z^2, -2q_4o_x - q_5o_y - q_7o_z, -q_5o_x - 2q_6o_y - q_8o_z, -q_7o_x - q_8o_y - 2q_9o_z, 0, 0, 0, 0, 0, 0)$.

5.1 Adapters

We solve this problem with the construction of a special kind of operator called an adapter. Whereas operators are designed to remain in the model, adapters are temporary and are used to control the parameters of a model during interactive editing.

An operator creates its parameter vector from its parameters and the parameters of its operands. This assumes that the operator's parameters are independent of the operands' parameters (e.g. the radius of an offsetting operation). The parameters of an adapter are assumed to be related to a subset of the parameters of its operands. The parameter vector of the adapter contains only the parameters of the adapter, and ignores the parameters of the adapter's operands.

For example, the adapter `Mover` is defined by

```
Mover::proc(x) { return F->proc(x - o) }
```

where F is the operand of `Mover` and $\mathbf{o}$ is an offset vector. The parameters specific to `Mover` consist only of the

offset vector. If `Mover` was an ordinary operand, then its parameter vector $\mathbf{q}$ would be $\mathbf{o}$ catenated with whatever parameters operand object F may have. Assuming F has been sufficiently parameterized, the additional components to $\mathbf{q}$ offered by the offset vector $\mathbf{o}$ would be redundant.

Since `Mover` is an adapter, we mask all of its operand's parameters, such that `Mover::procq()` returns only $\mathbf{o}$. This restricted parameter vector allows the constrained particle system to move an object using a single control particle. But in order for the object to remain in its new location, the adapter must remain attached. One can imagine a model becoming quite complex with adapters every time a subset of the model needs to be positioned.

We can remove an adapter if its parameters are set to its identity configuration, (zeroed in the case of `Mover`). Hence we need a way of transferring changes in parameters in the adapter to parameters in the adapter's operand.

An adapter implements some deformation function $D : \mathbf{R}^3 \rightarrow \mathbf{R}^3$. We can apply the deformation D to the implicit surface of $F(\mathbf{x}, \mathbf{q}_0)$ as a domain transformation, yielding the implicit surface of $F(D^{-1}(\mathbf{x}), \mathbf{q}_0)$. We need to find a new set of parameters $\mathbf{q}_1$ such that

$$F(D^{-1}(\mathbf{x}), \mathbf{q}_0) = F(\mathbf{x}, \mathbf{q}_1) \;\; \forall \mathbf{x} \in \mathbf{R}^3. \qquad (14)$$

The adapter function D has its own set of parameters $\mathbf{q}_D$. (In the `Mover` example, $\mathbf{q}_D = \mathbf{o}$). Let the parameters of F be denoted $\mathbf{q}_F$. We can use the Jacobian $d\mathbf{q}_F/d\mathbf{q}_D$ to find how changes in $\mathbf{q}_D$ affect $\mathbf{q}_F$. But this Jacobian would need to be derived and implemented to interface between every adapter and every `Implicit` primitive and operator in the modeling system.

Equations (2) and (3) provide a more general solution. We construct a collection of control particles and apply the deformation D to them, which causes the effect of the distortion to be applied to the original parameters of F.

We create a special array of particles $\mathbf{p}^i$. Even though each particle constrains the surface to pass through a three-dimensional point, the constraint restricts only one degree of freedom, since the particle may freely move across the two degrees of freedom along the surface. Hence, the number of particles n in the array should be $|\mathbf{q}_F|$. We assume that $|\mathbf{q}_D| << |\mathbf{q}_F|$ and F is flexible enough to find the solution of a much less flexible deformation D.

We then solve a variation of (3) specifically for processing the effect of D into the parameters of F,

$$\sum_j \left(F^i_{\mathbf{q}} \cdot F^j_{\mathbf{q}}\right) \lambda^j = F^i_{\mathbf{x}} \cdot (D(\mathbf{p}^i) - \mathbf{p}^i) + \phi(F^i - F(\mathbf{p}^i_0, \mathbf{q}_0)). \qquad (15)$$

The resulting Lagrangian multipliers λ^j provide the solution as

$$\dot{\mathbf{q}}_F = -\sum_j \lambda^j F^j_{\mathbf{q}}. \qquad (16)$$

We have assumed no desired parameter velocity ($\dot{\mathbf{Q}} = 0$).

Equation 15 has a slightly different feedback term that allows $F(\mathbf{p}^i)$ to be fixed to an arbitrary value, instead of just zero as was the case in (3). This feedback term makes sure the values at the particles do not drift away from their original values, which are found by evaluating F at the pre-deformation particle locations $\mathbf{p}^i_0$. Hence we can place particles anywhere in space to capture the field of F instead of just its implicit surface.

We sprinkle these particles randomly in space instead of across the surface. The implicit surface of aF is the same as that of F for any $a \neq 0$. Using particles on the surface constrained to $F = 0$ could yield an aF result with $a \neq 1$[2].

5.2 Implementation

We implemented (15) and (16) in the `setq` method of the adapter. When a surface control particle is dragged, (2) and (3) determine new parameters for the adapter's deformation through Euler integration of $\dot{\mathbf{q}}$. These new adapter parameters are then set by the particle system calling `setq`.

The `setq` of the adapter performs the following algorithm.

1. Set the state of its deformation D to use the parameters from the parameter vector $\mathbf{q}$ passed to it.
2. Use the deformation D to evaluate (15) and (16) to find the resulting parameter velocity $\dot{\mathbf{q}}_F$ of its operand F.
3. Perform an Euler step on the velocity $\dot{\mathbf{q}}_F$ by adding a fraction of it to the operand's parameter vector returned by `F->getq`, and store the result back in the operand's parameter vector via `F->setq`.

Since the parameters have been passed from the adapter to its operand, the adapter then returns its parameters to their original state. Hence, when an adapter's control particle is moved on an implicit surface, the adapter's parameters remain fixed and its operand's parameters change instead. This is the primary difference between an adapter and an operator in our modeling system.

5.3 Results

Figure 3 demonstrates this process on the `Mover` adapter applied to an ellipsoid that was originally placed at the origin. We have placed the translation adapter on the object and dragged the resulting composite object with a single particle. The paremeters of the translation are automatically propagated to the parameters of the underlying implicit ellipsoid primitive.

[2]This is also an issue when constructing implicit surfaces using radial basis functions. One or more constraint points are placed inside or outside the desired surface to indicate a desired interior or a desired local surface orientation [13].

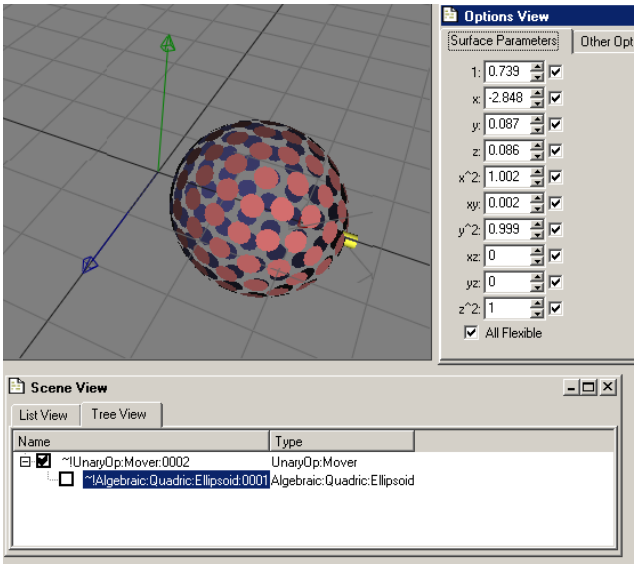

Figure 3. The effects of moving an ellipsoid by the single yellow particle on the coefficients of the quadric representation.

Careful examination of the parameters in Figure 3 reveals some numerical noise leaking into q_4, q_5 and q_6. This is most likely due to numerical error from the Euler integration. These inaccuracies may also contain some discretization error from the finite stochastic point-based sampling of the effects of the distortion.

This numerical noise causes the `Mover` operand to deform the ellipsoid as it translates. The feedback term is designed to reduce this distortion, but it is difficult to use this feedback term during the Euler integration because the randomly-positioned field particles do not actually move into their appropriate intermediate location as the parameter vector moves closer to the desired parameter vector. As a result, our implementation worked best when we took a large "predictor" step without feedback, followed by several "corrector" steps containing only the feedback term.

6 Conclusion

We have developed a implicit surface representation that is flexible enough to include a wide variety of different implicit modeling primitives and operations. We have built our system around a particle-based display and control system, and have explored several method for easily programming the derivatives needed for the particle dynamics and constraints.

We have also developed adapters which deform an implicit surface but embed the deformation in the parameter space of the implicit surface. These adapters provide the user with more intutive modeling parameters during interactive manipulation, but do not increase the complexity of the model.

6.1 The Implicit Modeling System

The implicit surface representation described in this paper is a subset of our actual system. Our full system serves as a base for our surface modeling research, and was designed to include a wide variety of different surface models. Figure 4 (generated automatically by *Doxygen* and *dot*) shows our present class hierarchy, demonstrating the range of implicit surface representations this class supports. Implicit surfaces are only one of the base representation classes supported by the system.

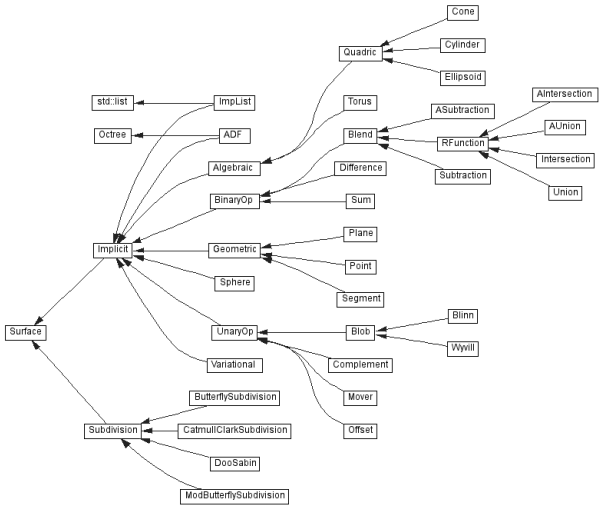

Figure 4. A current snapshot of our `Implicit` class hierarchy.

In the full system, our implicit objects include a second-derivative Hessian matrix for interrogating the curvature of the implicit surface. We plan to use the curvature to vary the size of particles across the surface, such that areas of high curvature receive many smaller particles. The `Implicit` class also includes interval extensions of many of the methods, to provide guarantees on properties over regions of space. We also have developed a toolkit for interval-based root finding and plan to implement topological polygonization guarantees on a larger selection of implicit surfaces than has previously before been accomplished [12].

Our goal for this system is to provide a publically-available open-source common environment for implicit surface research. The environment is available online at:

`http://graphics.cs.uiuc.edu/projects/surface`

The core of this representation was a group project of a class on advanced surface modeling taught in the Fall Semester 2000 at the University of Illinois. Each of the students was assigned a component of the library to implement as a project for the class. This format for a class project had several advantages. The student projects were not discarded at the end of class, which gives the students a stronger sense of accomplishment. The student projects were also distinct, which encouraged cooperation and teamwork instead of competition among the students. The interdependencies among the components of this library meant that it was in a student's best interest to help any other student that might be falling behind. This exercise provided production programming experience in an environment similar to the one many will find in their first jobs.

6.2 Future Work

The application of program differentiation tools on a given code segment is itself a complex task. It would be useful to develop a simpler subset of existing program differentation tools specifically for automatically translating `proc` methods into `grad` and `procq` methods.

The ability to "flatten" chains of multiple operators into a single operators is often employed in other hierarchical models in computer graphics. Depending on the success of implementations on implicit surfaces, it may be interesting to reparameterize the interfaces of other shape representations.

We have implemented the `Mover` adapter to verify the derivations in Section 5. We are in the process of implementing other adapters to perform deformations such as scale, rotation, taper, twist and bend. Barr [2] introduced the latter non-affine deformations, and it will be interesting to see how well some implicits, such as high-degree algebraics, can simulate the deformation effects within their parameterizatons.

Implicit surfaces are still *slippery* [14]. We have found that it is much easier to "pull" a convex surface than to "push" it. The *slipperiness* of the surface appears related to the flow gradient of the surface in the direction of the user-exerted force on a control particle. Constraining a control particle to a given position on the surface relative to nearby features could reduce the slippery feel of this method of modeling.

6.3 Acknowledgments

The core of our implicit surface library was coded by CS497JCH students Ed Bachta, Lennie Brown, Nate Carr, Jeff Decker, Bill Nagel and Steve Zelinka. Bill Lorensen, Will Schroeder and Ross Whitaker provided valuable insights into object oriented libraries from their experience with the vtk project. This research is supported in part by the NSF grant CCR-0196226 and the University of Illinois Department of Computer Science.

References

[1] V. Adzhiev, R. Cartwright, E. Fausett, A. Ossipov, A. Pasko, and V. Savchenko. Hyperfun project: a framework for collaborative multidimensional f-rep modeling. *Proc. Implicit Surfaces '99*, pages 59–69, Sept. 1999.

[2] A. H. Barr. Global and local deformations of solid primitives. *Computer Graphics*, 18(3):21–30, July 1984.

[3] C. H. Bischof, L. Roh, and A. J. Mauer-Oats. ADIC: an extensible automatic differentiation tool for ANSI-C. *Software: Practice and Experience*, 27(12):1427–1456, 1997.

[4] J. F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, July 1982.

[5] A. Griewank, D. Juedes, H. Mitev, J. Utke, O. Vogel, and A. Walther. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software*, 22(2):131–167, June 1996.

[6] C. Hoffman and J. Hopcroft. Automatic surface generation in computer aided design. *Visual Computer*, 1:92–100, 1985.

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

[8] H. K. Pedersen. imp. Source code available via implicit.eecs.wsu.edu, 1997.

[9] W. Schroeder, W. Lorensen, and S. Linthicum. Implicit modeling of swept surfaces and volumes. *Proc. Visualization '94*, pages 40–45, Oct. 1994.

[10] W. Schroeder, K. Martin, and W. Lorensen. *The Visualization Toolkit: An Object-Oriented Appriach to 3D Graphics*. Prentice Hall, Dec. 1997.

[11] W. J. Schroeder, K. M. Martin, and W. E. Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. *IEEE Visualization '96*, pages 93–100, Oct. 1996.

[12] B. T. Stander and J. C. Hart. Guaranteeing the topology of an implicit surface polygonization for interactive modeling. In *Computer Graphics (Annual Conference Series)*, pages 279–286, Aug. 1997.

[13] G. Turk and J. O'Brien. Shape transformation using variational implicit functions. *Computer Graphics (Proc. SIGGRAPH 99)*, pages 335–342, Aug. 1999.

[14] A. P. Witkin and P. S. Heckbert. Using particles to sample and control implicit surfaces. In *Computer Graphics (Annual Conference Series)*, pages 269–277, July 1994.

[15] B. Wyvill, E. Galin, and A. Guy. Extending the CSG tree: Warping, blending and boolean operations in an implicit surface modeling system. *Computer Graphics Forum*, 18(2):149–158, June 1999.