A Modeling Testbed

Kurt Fleischer
Andrew Witkin
Schlumberger Palo Alto Research
3340 Hillview Ave.
Palo Alto, CA 94304

Abstract

In many geometric modeling systems it is not possible to create new modeling primitives and transformations. Using ideas from object-oriented programming, we generalize the conventional graphics modeling hierarchy to include arbitrary 3D deformations, boolean operations, and shaders (nodes which describe the reflectance properties of a material). The representation used decouples model description from other aspects of the system, making the definition of new types of models possible without detailed knowledge of the sampling, rendering, or user interface implementations. Other benefits of this approach are also discussed, in particular its implications for the user interface.

KEYWORDS: Geometric Modeling, Object Oriented Programming, Solid Modeling, Testbed, User Interface.

I. Introduction

Most geometric modeling systems have a limited set of primitives, and adding new primitives is either difficult or impossible for the average user. The modeling testbed described here is designed to allow extensions and thus encourage experimentation with various types of models. Another major goal of the system is to formulate modeling primitives and transformations such that they can be arbitrarily connected; a primitive defined in the system can be modified by any transformations, in any order. To accomplish these goals, we concentrate on modularity in the design, achieved using an object oriented approach. By defining a standard *protocol* for communicating with the modeling code, we ensure a consistent interface between modeling operations (for arbitrary connectivity), and we isolate the mathematical model of an object from the particular methods used to sample and render it. These goals enable a user to define new modeling operations (eg. a Sphere primitive or a Bend transformation) and dynamically integrate them with existing code without concern for the implementation of other modeling operations, sampling, or rendering.

Information about the reflectance properties of a surface are traditionally bound up in the code implementing a shading algorithm. In an effort to provide the same modularity for surface appearance as for shape, we have developed a technique of specifying shading which enables it to fit nicely into the modeling hierarchy, in nodes called *shaders*.

The design of the system is presented in Section III. Section IV gives examples that show how the system looks to the user, including an example of the introduction of a new geometric transformation. In Section V, more elaborate extensions to the testbed are discussed. The implementation of the user interface is a similar extension, described in Section V-C. Our experiences with the system and some of the consequences of the modular design are given in the conclusion, along with ideas for future work.

Object oriented programming, a useful tool for building modular systems, was used extensively in the development of this system. A cursory knowledge of it is assumed in the presentation, but is not absolutely necessary since the required concepts are introduced in conjunction with concrete design goals and examples. Definitions of italicized terms can be found in a glossary at the end of the paper.

II. Related Work

Geometric modeling and rendering systems are generally large, often consisting of thousands of lines of code and taking years of effort. Such complex programs must be extremely well organized if they are to be at all flexible or extensible. Several testbed systems have been developed which provide the graphics researcher/programmer with these capabilities [6, 18, 30], but few of them provide the ability to create and combine new modeling transformations and primitives at a high level.

New programming paradigms such as object oriented programming can provide more programmer-efficient design and implementation strategies. Systems applying these techniques to geometric modeling and animation are becoming more common [7, 9, 11, 13, 16, 21], and have produced encouraging results.

Nadas and Fournier [17] present a nice overview of testbed systems, and a good discussion of the issues involved in designing a modeling testbed. Although our systems developed independently, there is a striking similarity between our choice of messages and their choice of data tokens (*appels*), which speaks well for the representation. There are still major differences between the systems: the way rendering is incorporated, the user interface, and

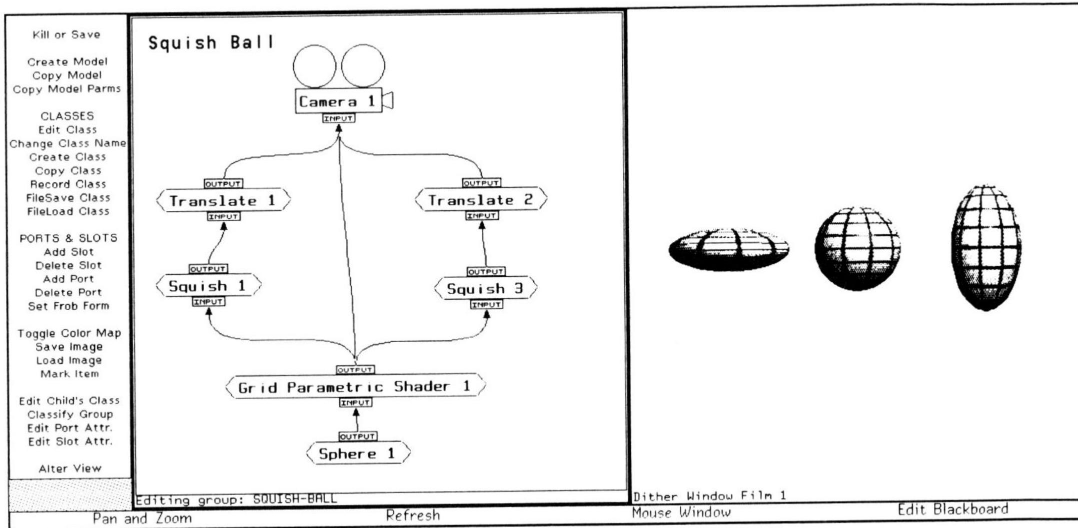


Figure 1: The system as it appears to the user, on a Symbolics 3670.[1] The model represented by the DAG on the left is rendered on the right. Note that the triple branching out of the shader creates three paths to the Sphere, each of which corresponds to a different object (from left to right: squished by .5, undeformed, squished by 1.5).

the implementation environment. These can have consequences on the capabilities of the systems as well. For instance, our message passing scheme and dual shape representation allow calculation of Booleans directly from primitive shape descriptions, without using intermediate representations (such as polygonal or spline approximation). Their system, GRAPE, is written in C, and programmer discipline is relied upon to enforce the modular design. Our system is written in Flavors, an object oriented extension to Zetalisp,[1] which makes the implementation somewhat more straightforward. In an interpreted, object oriented environment we are able to avoid the compilation phase used in GRAPE.

III. The System Structure

There are three levels to the design of this system: a conceptual model (which corresponds to that seen by the high level user), the underlying mathematical representation, and the object oriented implementation. The conceptual model is a generalization of the graphics modeling hierarchy. This directed acyclic graph (DAG) is manipulated by the user to indicate how to transform and combine modeling primitives. In the internal mathematical representation, applying multiple transformations is equivalent to function composition. The mathematical model is implemented as a standard set of messages (a *protocol*) passed among nodes in the DAG. The protocol is a modular interface to the model, accounting for interactions between modeling operations as well as between modeling and sampling or rendering.

[1] Zetalisp and Symbolics 3670 are trademarks of Symbolics, Inc.

A. The Conceptual Model

The DAG we use to describe models is probably familiar to most graphics practitioners [8, 25]. Leaf nodes are primitives, and interior nodes are transformations applied to their children. Each node in the graph, together with its subgraph, represents an object or a collection of objects. Multiple arcs out of a node indicate the duplication (by instantiation) of the object(s) described by that node's subgraph. Multiple arcs into a node indicate a merge of objects (Boolean union). The modeling graph in Figure 1, a screen dump of the testbed in operation, exhibits these properties.

Modeling operations generally have some modifying parameters. A Scale transformation has a scale factor, a Sphere primitive has a radius, etc. These ***static parameters*** [17] affect the calculation of the operation, but do not change during the course of a message evaluation (hence the term static).

The conventional modeling hierarchy is usually limited to polygonal primitives and affine transforms. Several generalizations can greatly extend the power of this modeling scheme. Incorporating Boolean operations into the graph seems a natural step [20]. Instead of restricting transformations to be affine, Barr [1] and Sederberg and Parry [23] have used more general 3D transformations. To model different light reflectance properties we include *Shader* nodes which contain information about the shading model to be applied as well as static parameters indicating the exact characteristics of this object (e.g. diffuse reflection, light sources) [4].

The resulting graphs are able to model a wide range of objects, and are easy to construct and modify. The Figures show graphs as they appear to users of the system. Note that the user is able to interact directly with the model

at the conceptual level. Allowing the user to manipulate these nodes and their connections with a graphical user interface provides a convenient way to manipulate, test, and compare new modeling operations.

Examining the Figures will immediately reveal the existence of a node which is neither a primitive nor a transformation, a *Camera* node. The Camera node is a convenient way to encapsulate information concerning the sampling and rendering operations to be performed on the model represented by the graph. Cameras have static parameters for the associated rendering algorithm (wireframe, zbuffer, antialiased, shadows, etc), as well as viewing information.

In order to evaluate modeling graphs to produce images (i.e. render the models), we need a precise definition of the internal shape representation.

B. Shape Representation

Modeling, in this paper, refers to the process of describing physical objects in mathematical terms; a model for a physical object is a bundle of mathematical functions which describe the shape, reflectance, mass, and other properties of that object. The modeling graphs described above are a means of describing complex objects as combinations of primitive parts which have undergone various transformations.

Our model for shape is dual, containing both parametric and implicit functions which represent the surface and volume (respectively) of an object. This dual formulation enables us to do computations in the most convenient and accurate representation. For instance, sampling a neighborhood of points on a surface can be done quite naturally with a parametric function, whereas Boolean operations are more conveniently specified using the implicit function, or a combination of the two. Because transformations in both parametric and implicit representations can be composed in a modular way, we need only compute the dual for the component primitives and transformations, which is usually straightforward. The disadvantage is that in some cases one of the representations may be difficult or impossible to find from its dual [22]. [2]

In addition to the parametric and implicit functions, we include a normal function. Although this could be determined numerically, keeping it as a separate function enables the implementation to be decided in each object, so that objects with the capability to compute the normals analytically are able to do so.

Surface Position $P_k(u,v) \rightarrow (x,y,z)$
: The piecewise parametric function $P_k: \Re^2 \rightarrow \Re^3$ is defined on the unit patch ($u, v \in [0,1]$). Evaluated at some point (u,v), it returns a 3D position on the surface. Each patch k describes a surface in 3-space, and the ensemble of the P_k patches describe the entire surface of an object.

Surface Normal $N_k(u,v) \rightarrow (nx,ny,nz)$
: Normals are computed using $N_k: \Re^2 \rightarrow \Re^3$, another piecewise parametric function similar in form to P_k.

Implicit $I(x,y,z) \rightarrow s$
: The implicit function $I: \Re^3 \rightarrow \Re^1$ describes the volume enclosed by the P_k patches. This function is positive for points outside the object, negative for points inside the object, and zero for points on the surface. Thus the solution to $I(x,y,z) = 0$ is the same surface as the P_k patches. Note that I may correspond to a volume enclosed by one or more patches.

The preceding functions describe the behavior of a modeling primitive (leaf node in the modeling graph). Modeling transformations (interior nodes) are characterized as follows:

Transform $T(x,y,z) \rightarrow (x',y',z')$
: The transformation function $T: \Re^3 \rightarrow \Re^3$ is a global 3-space deformation. In the case of translate, T is just a vector addition of the argument and the translation vector (a static parameter).

Transform Normal $TN(nx,ny,nz) \rightarrow (nx',ny',nz')$
: The normal transformation function is the inverse Jacobian of the transformation T [1].

Inverse Transform $T^{-1}(x,y,z) \rightarrow (x',y',z')$
: The inverse transformation is another $\Re^3 \rightarrow \Re^3$ function, the inverse of T.

Using function composition, the transformation functions can be composed (recursively) with the primitive functions to create valid parametric, normal, and implicit functions. Given a position function, $P: \Re^2 \rightarrow \Re^3$ and a 3D transformation function $T: \Re^3 \rightarrow \Re^3$ the function $(T \circ P): \Re^2 \rightarrow \Re^3$ exists. So we can apply any valid transformation T to the surface position function and obtain another valid parametric surface position function. A similar result holds for the normal and inverse transformations.

These functions provide a mathematical basis for evaluating objects represented by a modeling graph, and we can use them to produce an image. There are many different techniques for sampling and rendering a mathematical shape model. They range from polygonal approximation, which samples in object space, to ray tracing, which samples in world space. Adaptive sampling based on image criteria, Boolean boundaries, etc, also works well with this model [28]. With each technique the model is first sampled, and then the samples are used to create an image. Here is high level pseudocode for rendering these models using polygonal approximation:

```
for each patch k
  for u,v over patch [0,1]^2
    sample using P_k(u,v) and N_k(u,v)
    save sample in sample array
  for i,j over sample array
    perform viewing transformations on sample points
    create triangles between adjacent samples
    shade triangles
```

[2] In cases where the implicit function has not been defined, the object in question will still be available for use as a parametric surface, but operations such as Booleans will fail. This could be circumvented by using an approximation to the implicit function generated by sampling the parametric surface, but we have not found this necessary.

A sample here is a nugget of information about a point on the surface, containing position and normal. Given this information the shade triangle step can be computed using some standard shading algorithm (more sophisticated shading methods are described in Section V-A).

By defining modeling in a modular way with a consistent functional representation, we reap two immediate benefits. First, the separation from rendering makes changes to the modeling operations much easier. Second, relationships between modeling operations are well defined so we can compose them easily with each other. Note that defining a transformation or primitive is merely a matter of specifying the functions associated with its geometry. A programmer writing code for a new shape model need not be concerned with shading, sampling, or rendering, since they are guaranteed to work with this same mathematical definition.

C. Object Oriented Programming Implementation

Use of the object oriented paradigm should be clear in the preceding exposition of the mathematical model: there is a well defined interface to nodes in the modeling graph, without a specification of how the nodes will implement their pieces of the functions. All that remains is to apply object oriented programming terminology. The position, normal and implicit functions are implemented as messages which are sent to the nodes. Each node is an object in the object oriented programming sense (unfortunate collision of terminology). The ensemble of the three messages is a *protocol*, a standard set of messages implementing a particular facility [24].

IV. Examples of the Benefits of this Scheme

A. An Example of the Modeling Graph

Consider the graph shown in Figure 6. This graph represents a model of two objects. At the top is the Camera node, which contains the code for viewing and rendering. Beneath that are several interior nodes which correspond to familiar transformations such as Translate, as well as deformations (e.g. Bend), and shader nodes (e.g. Layercake Solid Shader). At the bottom, the leaves of the graph correspond to geometric primitives (e.g. Block).

Every path in the graph terminates at the Camera. Each path describes a part of the object, and traversing the path gives the transformations that are successively applied to the primitive to produce the transformed part. All of the paths considered together comprise a complex object (or scene) which are interpreted by the rendering code to create an image.

B. Defining and Integrating New Modeling Operations

One of our original goals was to allow users to define new modeling operations succinctly and integrate them into the rest of the system with ease. This example of the definition of a simple node will help to show both the desirability this goal, and that it has been achieved.

The macro deftransform is used to define a geometric transformation (an interior node in the modeling graph). The user must specify the static parameters of the transformation, position and normal transformation functions (T and TN), and the inverse transformation function (T^{-1}). The macro then expands into code that creates an object with a user interface (for setting the static parameters and connecting into the graph) and provides dummy handlers for all messages of the protocol. Messages such as Color and Shading_info which are not affected by this type of transform are simply forwarded to the children. Analogous macros define primitives, shaders and other modeling operations.

The definition of the new transform Squish given below is an actual definition that is used in the testbed. It consists almost entirely of the equations for the modeling functions, which is as it should be.

```
;;; Squish in z, expand in x and y to preserve volume.
(deftransform squish
  ((sz 1.0))                 ; static parameter - z scale factor
  squish-sample              ; transform function
  inverse-squish-point)      ; inverse xform function

;;; Definition of Squish's transform function
(defun squish-sample (x y z nx ny nz sz)
  (let ((sxy (/ 1.0 sz)))
    (return
      (* sxy x) (* sxy y) (* sz z)       ; return position
      (/ nx sxy) (/ ny sxy) (/ nz sz)    ; return normal
      )))

;;; Definition of Squish's inverse transform function
(defun inverse-squish-point (x y z sz)
  (let ((sxy (/ 1.0 sz)))
    (return (/ (x vect) sxy)
            (/ (y vect) sxy)
            (/ (z vect) sz)))))
```

After defining Squish we can use it just as we would use any other transformation. Figure 1 shows the Squish operation in action. Figures 3 and 6 provide examples of the variety of transformations which can be accommodated. The free form deformations of Sederberg and Parry were implemented in about an hour, using the equations specified in their paper [23].

V. Extending the Protocol

The object-oriented programming concept of *inheritance* enables the easy creation of objects which are almost like other objects with a few incremental changes[24]. We have already used this to create nodes which by default handle

all of the messages in the protocol by passing them on to their children (as in the Squish example). To make a node type that modifies only the shading function, we merely add a *method* (a procedure) to handle the shading message, and the other the messages of the protocol are handled as before (getting passed through to the children). This is what deftransform does, providing dummy methods for the shading messages and adding new methods for the specified functions.

Inheritance is simply the idea that we can add a new message to the protocol (hence a new attribute to our model) without making any changes to the old messages. First we must redefine the protocol to include the new message, then we create methods to handle the message. Then we create nodes which handle the extended protocol by inheriting methods for the new message. The next three sections describe extensions to the protocol to implement shaders, Boolean operations and elements of the user interface.

A. Shaders

Computer graphics is primarily concerned with the appearance of objects, and a part of modeling this is modeling the light reflectance properties of various surfaces. From a more practical point of view, it is often useful to represent scenes where different shading and lighting models are applied to different objects. These concerns motivate the inclusion of shading nodes in the modeling hierarchy. Placing a node such as Phong[19], Texture Map, or Torrance-Cook[3], in the graph indicates the shading algorithm to be performed on objects which are composed with that node.

As with shape representation, the conceptual model constrains but does not completely determine the internal representation for shading. In Cook's *Shade Trees* paper [4], he begins by stating

> *"The traditional approach to shading is to divide the calculations into two stages:*
>
> *1. Determining the values of the appearance parameters.*
>
> *2. Using those values to evaluate the fixed shading equation."*

He then goes on to point out that fixing the shading equation severely limits the generality of the system, and proposes a method which abandons the two stage approach, and avoids this problem. We solve the problem in a different way, by allowing the shading equations to vary based on which object is being shaded. Clearly the first and second stages need to be closely related so that the computed *appearance parameters* are those expected by the shading equation.

The reason for keeping the two stage approach is that the results of the first stage, calculated at sample points, can be interpolated across an approximating surface (e.g. a triangle) and the interpolated values can be used at each pixel to complete the second stage of the shading calculation. For example, in Gouraud shading[10] the first stage computes colors and these are interpolated across the triangle, and the second stage at each pixel merely uses the given color. In Phong shading[19], the first stage returns a normal which is interpolated across the triangle. The second stage, called at each pixel, computes the color using the appropriate algorithm.

Other relevant information such as diffuse reflectance, reflectance spectra, light sources, etc, are contained in the static parameters of the node which is handling the two stages of the shading function.

Composition of shaders is not currently handled as generally as composition of shape models. A shader node either responds to the messages regarding shading, or allows them to pass through to another shader node (this is decided via a static parameter switch). With this protocol it is not possible for more than one shader node to affect the appearance parameters of a single object. Adding appropriate messages to the protocol would enable more flexibility, so that specifying a texture map could be done orthogonally from the specification of the reflectance model.

The actual implementation of these functions uses *closures*[26] instead of message passing to obtain greater efficiency, since Color will be evaluated at least once for every non-background pixel in the final image. The closure contains code to evaluate the shading algorithm as well as pointers to relevant local information from the shading node (such as light sources, diffuse reflectance, texture map, etc). Despite the difference in implementation, the object oriented program structure is retained.

Shading_info($position, normal, u, v$) $\rightarrow ap$

> This function is called at every sample point. It gathers all of the surface-dependent information needed to evaluate the shading function at this u, v point, and returns it as a vector of appearance parameter numbers, (ap). In the case of Gouraud shading[10], it simply returns a color, whereas for Phong shading [19], it returns a vector containing the position and normal of the point.

Color(ap) $\rightarrow (r, g, b)$

> The light reflectance calculations are performed in this function. All of the information needed for the particular shading algorithm is provided in the appearance parameter argument (ap), and is used to calculate a final color. In the case of Gouraud shading, ap is the output color. For Phong shading, ap contains a normal used in conjunction with light source and reflectance properties to compute the final color. The other information (object color, light source, etc.) is available locally since the node handling the message holds these as static parameters.

A rendering scheme such as the polygonal approximation described above can be modified to use this formulation for shading. During triangle display, the vectors containing shading information are interpolated across the triangle. At each pixel, the Color message is sent with the interpolated vector, producing the output color. Other in-

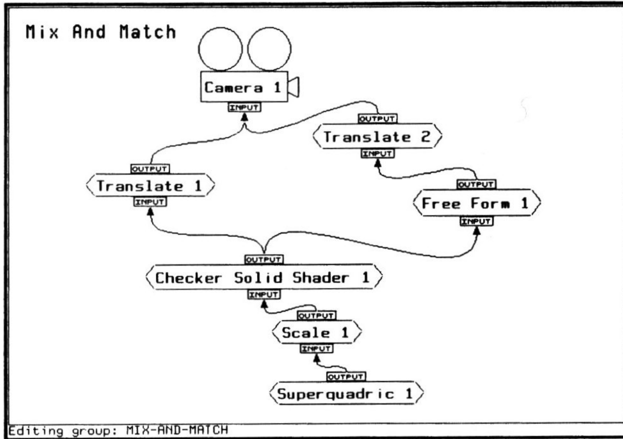


Figure 3: *Mix and Match Models.* Three different modeling techniques are combined in this image: Superquadrics [1], Solid Textures [18], and Free Form deformations (Sederberg & Parry [23]). independently and no special case interface code was written.

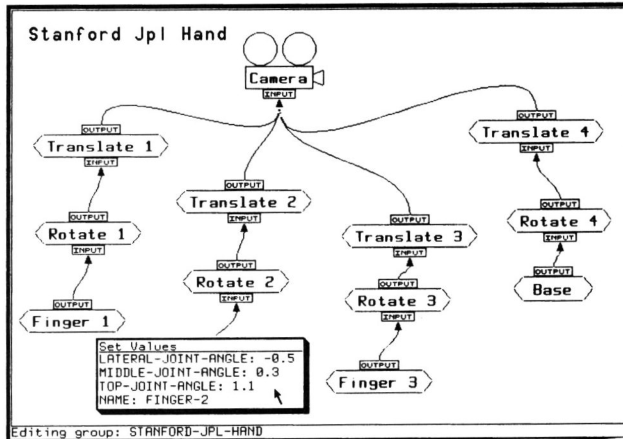


Figure 4: *Stanford JPL Robot Hand.* This complex model exhibits the use of classes for meaningful parts, such as the JPL Finger node, which has been instantiated three times, with different parameter settings.

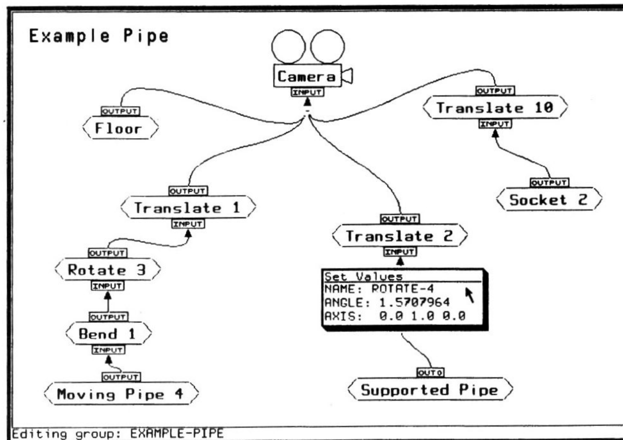


Figure 5: *Energy Constraints.* The parameters determining the shape and location of the bent pipe are determined by solving the constraint that it should lie in the brackets.

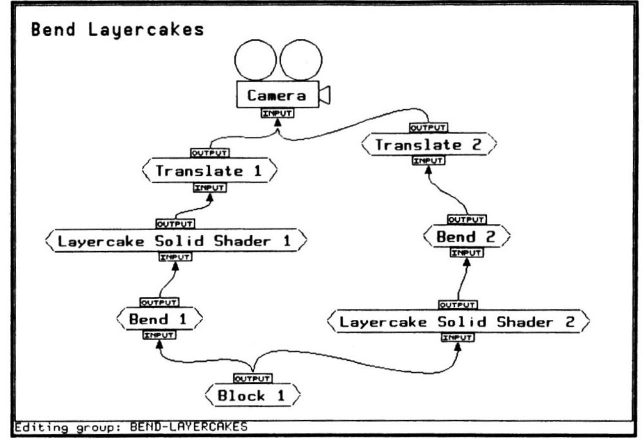


Figure 6: *Bend Layercakes.* The Bend operation applied to a Block before it gets the Layered Solid Texture has the effect of cutting a bent block from the layercake. By rearranging transformations, we can apply the layered texture first and then bend the block and the texture.

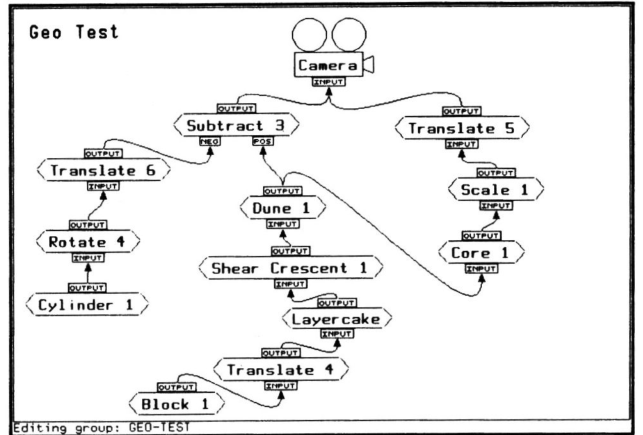


Figure 7: *Geology.* Starting with a solid texture to describe sedimentary strata, we deform it using Dune and Shear Crescent transformations to obtain a geologic model. Boolean operations are used to extract a core sample.

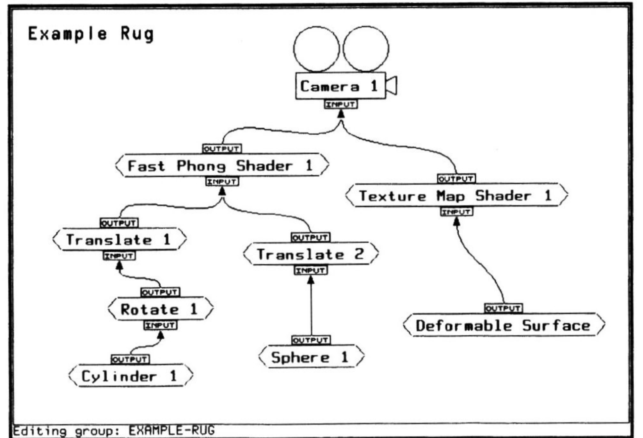
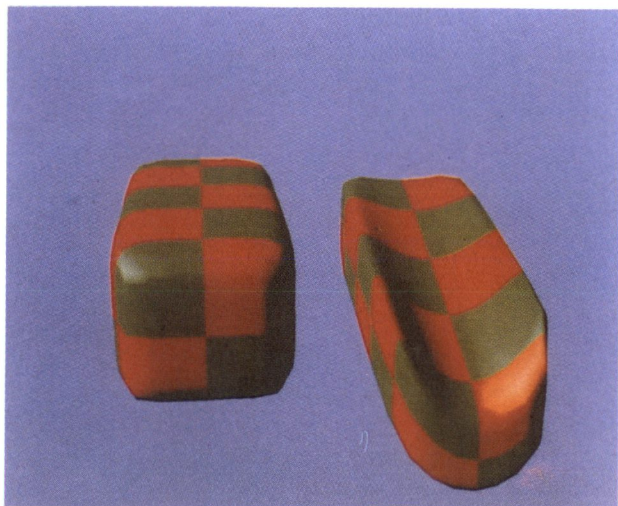
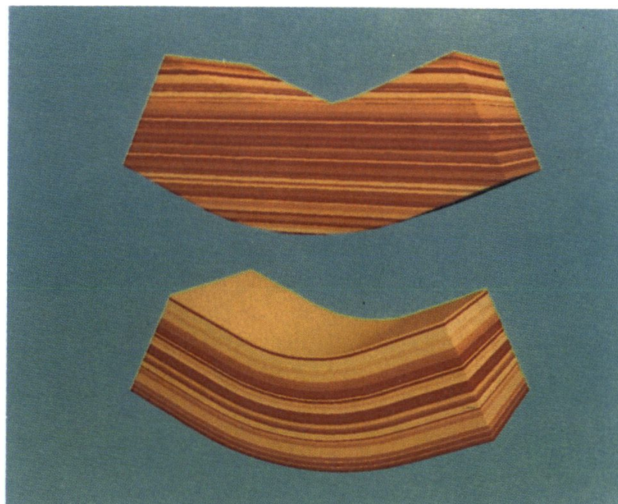
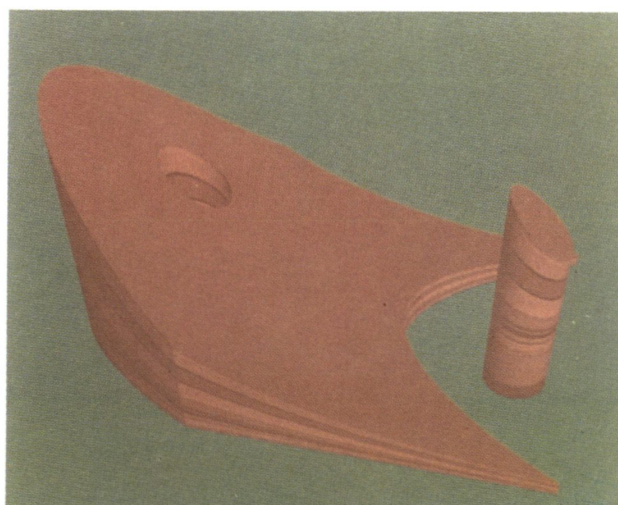


Figure 8: *Deformable Surface.* A simulated deformable surface is falling upon a stationary sphere and cylinder. The rendering of the surface with a texture map is provided by the system, as well as the definitions of the sphere and cylinder with which it collides.







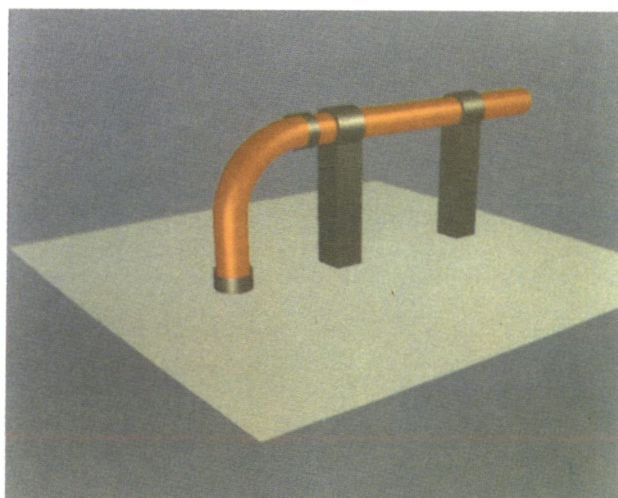


Figure 9: Images for the models of Figures 3 through 8.

formation needed during the shading calculation which is constant over the surface is incorporated directly into the object receiving the Color message. For ray tracing no interpolation is necessary, and the messages can be composed and called at each sample point (ray), where the color is Color(Shading_info($position, normal, u, v$)).

This characterization of shading is quite general, and has allowed us to implement a wide variety of shading algorithms, including Gouraud, Phong, texture and bump maps [12], solid textures [18], Torrance-Cook [3], and others. It enables shading to be separated from sampling and rendering processes, so that new shading models can be tested without changes to the underlying system. All of the code a user writes to define a new shading model involves shading; the sampling, rendering, and shape modeling code need not be changed or even consulted.

B. Boolean Operations

Boolean set operations (subtract, intersect, union) on volumes are a useful modeling tool which can be added to the system. Booleans operate on solid models, hence the implicit form of the shape model is used for their implementation (this formulation is due to Von Herzen [28]).

The parametric surface position function of a Boolean operation must be some combination of pieces of the surfaces of the source objects for the operation. No new surfaces will be added, but some previously visible surfaces may be removed, or some previously hidden (embedded) surfaces may be revealed. Since the surfaces are unchanged, a single visibility message will suffice for indicating which points on the surfaces are visible on the resulting object:

Visibility $V_k(u,v) \to r$

V_k: $\Re^2 \to \Re^1$, like P_k earlier, is a piecewise (for each patch k) parametric function defined on the unit patch ($u, v \in [0,1]$) which describes the visibility of each point on the surface. V_k is negative where the surface is visible, positive where it is not visible, and crosses zero at the boundary.

The visibility function could just be a binary function, but in practice that is unwise since it may be used to locate a Boolean boundary with standard root finding methods. A visibility number calculated from the implicit function (already in the protocol), is appropriate for root finding methods.

Now we can implement Boolean operations, for example *subtract*. To respond properly to the entire protocol, a subtract node must handle several messages:

Visibility $V_k(u,v) \to r$

First calculate the position $P_k(u,v)$, and figure whether the patch k is generated by a positive or negative child. If k is a positive surface, then a point is visible only if it is outside the object defined by the negative children. Alternatively, if k is a negative surface, points are visible only if they fall within the object defined by the positive children. We use the implicit functions of the children to determine when a point is inside the object.

Surface Normal $N_k(u,v) \to (nx, ny, nz)$

This function negates normals from negative children, and leaves normals from positive children unchanged.

Implicit $I(point) \to s$

The functionality that we want is that a point inside this composite object will have to be inside the positive object and outside the negative object. A function to do this (using the implicit functions of the positive and negative children) is $max(I_{Positive}(point), -(I_{Negative}(point)))$.

Shading_info and Color

When a subtract is used to cut open an object, the color of the interior can be determined either by the shaders in the negative object or the positive object. Thus we can express things like "the inside of this red object is green," or we can explore the internal structure of an object by using the same solid texture for all surfaces (Figure 7). A switch (as a static parameter) is provided for this purpose.

The Boolean operations Intersect and Subtract are used to create the core sample shown in Figure 7.

C. The User Interface

Another extension to the protocol provides handles for the user interface. The particular user interface we have chosen displays the modeling graph on the screen as labeled boxes connected by arrows, and enables the user to change the connectivity of the graph interactively. Parameters such as rotation angle and cylinder radius are available for inspection and modification using pop-up menus (see Figures 5 and 7).

As with the protocols to describe shape and material properties, the messages in the user interface protocol can be handled by each node in a different way. However, since all nodes are essentially identical from a graph editing point of view (each is a node with directed ports and static parameters), we provide a group of methods for nodes to handle the user interface protocol. Changes to these methods can be made for specific node types, as in the case of a special node like Camera, but for most nodes the default is sufficient.

The user interface is a directed graph editor, and it expects to deal with nodes according to a simple protocol. It expects the nodes to have a list of slots with values and properties. The properties determine whether the slot is a connection or a static parameter, the name of the slot, and details about valid data types for that slot. The user interface also offers the nodes the opportunity to handle mouse clicks in special ways, in case there is a special purpose interface for setting the parameters.

When a new node is defined and presented to the user interface, the slot information is used to create a box on the screen with appropriate mouse-sensitive ports for connec-

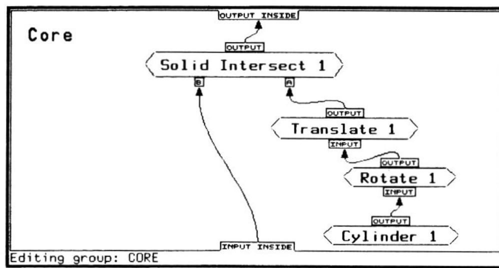


Figure 2: *Core: A Class Definition.* The Core transformation (to take a core sample) was defined using existing primitives and transformations. A message sent to the Core from outside is handled by the nodes inside the group. Messages sent from objects inside the group are passed to the appropriately connected nodes outside.

tions, and a pop-up menu for selecting static parameters (Figures 4, 5).

1. Hierarchy of Nodes

When creating a model for an object that has many parts, the modeling graph gets unwieldy very quickly, filling up the screen with little boxes. We can hide some of the complexity by collapsing a subgraph into a single node (another hierarchy). This can be done without changing the protocol. A *group* is defined to handle the protocol by passing all messages from outside through to the nodes on the inside, and vice versa (Figure 2). The user interface can then display this group node instead of the subgraph of nodes inside. Since each level of group will pass the messages through properly, this works recursively to create a hierarchy of groups within groups.

2. Physical Object Classes & Instances

The definition of groups can be extended to make object classes (templates) by saving the structure of the group. Instances of the class can be used throughout the system just as regular nodes, since they obey the same protocol. For example, the class *JPL Finger* is instantiated three times in the final *Stanford JPL Hand* (Figure 4). An instance is not the same as a copy, since changing the class will cause all of the instances to be updated to maintain consistency with the class definition.

Note that a class or group need not be a primitive (leaf node) like *JPL Finger*; it could be a transformation such as Core (Figures 2, 7), involving several inputs and/or outputs.

3. Parameterized Instances

Instances are more useful if they can be parameterized. In our previous example, *JPL Finger* has parameters lower-angle, middle-angle, and upper-angle, which specify the angles to bend the respective joints in the finger (Figure 4). In this way the three fingers can be bent independently. In this case, the parameter lower-angle to the group merely sets the rotation angle parameter in an internal Rotate node to the given value. A more general version of this concept allows the attachment of an arbitrary function to a node's parameter. This can be a function of any combination of the group's parameters, so the node's parameter function gets re-evaluated whenever any of the group's parameters is changed. Using this mechanism we can do simple downward-only constraints such as keeping the aspect ratio of a block constant.

VI. Experience with the System

In addition to enabling the definition of new geometric models, we have found the modularity of the system lends itself to the development of other modeling techniques as well. Included here are examples of how the modeling testbed aided in the development of two very different modeling schemes.

A. Energy Constraints

The model represented by an entire graph depends on the union of all of the static parameters in all of the nodes. We can define constraints in world space which depend on the functions implemented by the protocol, and do optimization to find the parameter settings which solve the constraint. For instance, given a model of a bent pipe, we can modify the parameters of the Bend (and other components of the model such as length of pipe, position, etc.) until the end of the pipe touches a given point in space. The constraint in this case is the distance between the end of the pipe and the point, and a requirement that the normals be opposing (Figure ??). This formulation of constraints and their solution was partially inspired by the structure of this testbed. It is sometimes a tedious process to set the static parameters appropriately for the desired scene. Since it is easy to get functions describing physical objects and their parameters from the system, optimizing those functions based on constraint goals seemed a natural step. A detailed description of this work can be found in [31].

B. Deformable Models

By numerically simulating the behavior of elastically deformable models we can obtain a variety of materials such as cloth, rubber, paper, and flexible metal, as described in [27]. In our work on deformable models, the modeling testbed provided initial shapes for deformable objects, objects with which to interact, and a variety of rendering techniques (Figure ??). The functional definitions of the geometric models made it easy to use existing geometric models in conjunction with the new deformable models. After the simulations were complete, it was also simple to create a geometric model containing the shape of the deformable model at an instant in time, enabling us to use existing rendering tools.

VII. Future Work

We envision many potential improvements and extensions in the user interface and rendering/sampling methods. Developments to increase the speed are also being considered.

The user interface for setting static parameters is currently tailored to specific parameters by specifying the data type (number, string, etc), and the user is not allowed to put incorrect data types into slots. This could be extended by attaching specific interaction methods to particular static parameters, allowing users to set numbers using the mouse or some other input device. With sufficient improvements in speed, it would be possible for the user to interactively modify any numeric parameter in the graph and watch the object change accordingly (in either shape or shading). We currently accomplish this functionality by rendering a sequence of images with various parameter settings into a short movie, and viewing the sequence using bitblt operations.

The current rendering methods in the system include wireframe, zbuffer polygons, and antialiased polygons. Although these are largely sufficient for our needs, ray tracing the implicit shape representations [2, 15] would provide some new capabilities. Hierarchical bounding boxes, which exist in the current system, would aid the efficiency of a ray tracing implementation. A micropolygon rendering technique [5] fits into this scheme quite nicely since there is always a parametric definition for dicing the objects.

Some objects have a computationally preferable method of being sampled, such as subdividable patches. This information should be taken into account by defining a protocol for describing sampling algorithms. A simple version of this protocol has been successfully implemented to avoid redundant sampling of polyhedral surfaces.

In working with the system, a user often makes a change to one static parameter and then renders the model to examine the modification. Caching unchanged samples could speed up this process substantially. With dependency information, it is possible to determine which parts of the model are affected by changing certain static parameters. Such an extension might enable us to dynamically vary a static parameter and render the modified model in near-real time, as suggested above. Another potential speed up is to perform graph optimization, compiling parts of the graph into faster representations where possible. For instance, a sequence of affine transformations can be compressed into a single matrix multiply. In the extreme, the sequence of functions represented by a path in the graph could be collected and compiled as one huge function, saving the overhead associated with message or function calling.

VIII. Conclusion

Versions of the modeling testbed described have been in use at Schlumberger Palo Alto Research and Schlumberger-Doll Research for over three years, and the goals of modularity with respect to rendering and between modeling operations seem to have been achieved. The testbed has been used to model robots, geologic formations, mechanical linkages, and a variety of deformable objects. Use of the object-oriented programming methodology has given the system enough flexibility and extensibility to change with our needs.

Acknowledgements

Many aspects of the basic functional representation for shape came from discussions with Alan Barr and Brian Von Herzen of Caltech. Thanks also to Stefan Luthi of Schlumberger-Doll Research and Robert Howe of Stanford University for their work with the geology and robot hand models.

Glossary

appearance parameters Any value that is used in a shading calculation (Cook[4]).

appel Appearance element. This is a term which Nadas and Fournier[17] use to describe the data tokens in their networks. It is a compound structure, consisting of several elements such as normal, color, transparency, etc. We include it because of the similarity to our message choices of position, normal, visibility, color, etc.

camera A special node which contains information about how to render the objects represented in the modeling graph below it. Its static parameters include viewing parameters, background color, sampling information, etc. (see Figures).

closure A function which contains another function and a set of (local or global) variable bindings[26].

group A subgraph of modeling operations which has been collapsed into a single virtual node.

inheritance This is an object oriented programming concept which enables the easy creation of objects which are almost like other objects, with a few incremental changes [24].

message A form of indirect procedure call which invokes a procedure associated with a particular object's class, using the data contained in the object receiving the message [24].

method The code used to handle messages [24].

modeling graph A graph whose nodes are modeling operations (e.g. translate) and primitives (e.g. sphere), and whose connections indicate the order of composition of those operations to describe a physical object.

modeling operation A primitive or transformation.

primitive (or modeling primitive) A leaf node in a modeling graph. A primitive describes the shape of some basic object (e.g. cylinder, superquadric) which has no component parts.

protocol A set of messages defining a uniform interface to objects that provide a certain facility [24].

prototype A class describing a physical object or transformation (in this paper only).

shader A transformation which affects the shading properties of an object. It may change both the shading algorithm used as well as the static parameters.

static parameters High level parameters to a modeling operation, which do not change during the course of evaluating a model. This is a term from Nadas and Fournier [17].

transformation (or modeling transformation) An interior node in a modeling graph, describing a transformation to be performed on a primitive.

References

[1] Barr, Alan, *Global and Local Deformations of Solid Primitives,* Proc. Siggraph 84, pp. 21–32.

[2] Barr, Alan, *Ray Tracing Deformed Surfaces,* Proc. Siggraph 86, pp. 287–296.

[3] Cook, Robert L., and Kenneth E. Torrance, *A Reflection Model for Computer Graphics,* ACM Transactions on Graphics, Volume 1 Number 1, January 1982.

[4] Cook, Robert, L., *Shade Trees,* Proc. Siggraph 84, pp. 223–231.

[5] Cook, Robert L., and Loren Carpenter, and Edwin Catmull, *The Reyes Image Rendering Architecture,* Proc. Siggraph 87 pp. 95–102.

[6] Duff, Tom, *Compositing 3-D Rendered Images,* Proc. Siggraph 85, pp. 41–44.

[7] Fleischer, Kurt and Andrew Witkin, *Implementation of a Modeling Testbed,* Siggraph 87 Course Notes, Vol 14.

[8] Foley, James D., and Andries van Dam, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley (1982).

[9] Getto, Phillip, *The Clockworks: An Object Oriented Computer Animation System,* Siggraph 87 Course Notes, Volume 14.

[10] Gouraud, H., *Continuous Shading of Curved Surfaces,* IEEE Transactions on Computers, June 1971.

[11] Grant, Eric, Phil Amburn, and Turner Whitted, *Exploiting Classes in Modeling and Display Software,* Proc. Graphics Interface 1986, reprinted in IEEE CG&A Vol. 6 No. 11, November 1986.

[12] Heckbert, Paul, *Survey of Texture Mapping,* IEEE CG&A Vol. 6 No. 11, November 1986.

[13] Hedelman, Harold, *A Data Flow Approach to Procedural Modeling,* IEEE CG&A Vol. 4 No. 1, Jan 84.

[14] Howe, Robert, *Geometric and Physical Modeling of Robot Hands,* Stanford CS529 report, Spring 1985.

[15] Joy, Kenneth I. and Murthy N. Bhetanabhotla, *Ray Tracing Parametric Surface Patches Utilizing Numerical Techniques and Ray Coherence,* Proc. Siggraph 86, pp. 279–285.

[16] Lorenson, W., M. Barry, D. McLachlan, and B. Yamrom, *An Object-Oriented Graphics Animation System,* General Electric TIS Report #86CRD067, June, 1986, reprinted in Siggraph 87 Course Notes, Volume 14.

[17] Nadas, Tom, and Alain Fournier, *GRAPE: An Environment to Build Display Processes,* Proc. Siggraph 87, pp. 75–84.

[18] Perlin, Ken, *An Image Synthesizer,* Proc. Siggraph 85, pp. 287–296.

[19] Phong, Bui-Tuong, *Illumination for Computer-Generated Pictures,* CACM June 1975.

[20] Requicha, A. A. G., and H. B. Voelcker, *Solid Modeling: Current Status and Research Directions,* IEEE CG&A, Vol. 3, No. 7, October 1983.

[21] Sabella, Paolo, *Geometric Knowledge Bases,* Siggraph 87 Course Notes, Volume 14.

[22] Sederberg, Thomas W., *Implicit and Parametric Curves and Surfaces for Computer Aided Geometric Design,* PhD Thesis, Purdue University, West Lafayette, IN (1983).

[23] Sederberg, Thomas W. and Scott R. Parry, *Free-Form Deformation of Solid Geometric Models,* Proc. Siggraph 86, pp. 151–160.

[24] Stefik, Mark, and Bobrow, Daniel G., *Object Oriented Programming: Themes and Variations,* The AI Magazine, 1985.

[25] Sutherland, I. E., *SKETCHPAD: A Man-Machine Graphical Communication System,* SJCC 1963, Spartan Books, Baltimore, MD, pp. 329.

[26] Symbolics, Inc. *Symbolics Release 7 Documentation,* Volume 2A, pp. 251-271.

[27] Terzopoulos, Demetri, John Platt, Alan Barr, and Kurt Fleischer *Elastically Deformable Models,* Proc. Siggraph 87, pp. 205–214.

[28] Von Herzen, Brian, *Sampling Deformed, Intersecting Surfaces with Quadtrees,* Caltech Masters Thesis, 1985.

[29] Von Herzen, Brian, *Accurate Triangulations of Deformed, Intersecting Surfaces,* Proc. Siggraph 87.

[30] Whitted, Turner, and David Weimer, *A Software Test-Bed for the Development of 3-D Raster Graphics Systems,* ACM Transactions on Graphics, Vol. 1 No. 1, January 1982, pp. 43–58.

[31] Witkin, Andrew, Kurt Fleischer and Alan Barr, *Energy Constraints on Parameterized Models,* Proc. Siggraph 87, pp. 225–232.