An Object-Oriented Framework for the Integration of Interactive Animation Techniques

Robert C. Zeleznik, D. Brookshire Conner,
Matthias M. Wloka, Daniel G. Aliaga, Nathan T. Huang, Philip M. Hubbard,
Brian Knep, Henry Kaufman, John F. Hughes and Andries van Dam

Department of Computer Science
Brown University
Providence, RI 02912

ABSTRACT

We present an interactive modeling and animation system that facilitates the integration of a variety of simulation and animation paradigms. This system permits the modeling of diverse objects that change in shape, appearance, and behavior over time. Our system thus extends modeling tools to include animation controls. Changes can be effected by various methods of control, including scripted, gestural, and behavioral specification. The system is an extensible testbed that supports research in the interaction of disparate control methods embodied in controller objects. This paper discusses some of the issues involved in modeling such interactions and the mechanisms implemented to provide solutions to some of these issues.

The system's object-oriented architecture uses delegation hierarchies to let objects change all of their attributes dynamically. Objects include displayable objects, controllers, cameras, lights, renderers, and user interfaces. Techniques used to obtain interactive performance include the use of data-dependency networks, lazy evaluation, and extensive caching to exploit inter- and intra-frame coherency.

CR Categories and Subject Descriptors

I.3.2 Graphics Systems; I.3.4 Graphics Utilities, Application Packages, Graphics Packages; I.3.7 Three-Dimensional Graphics and Realism, Animation; I.6.3 Simulation and Modeling Applications; D.3.3 Language Constructs

Keywords

Real-time animation, object-oriented design, delegation, simulation, user interaction, electronic books, interactive illustrations.

1 Introduction

Over the last two decades, graphics research has concentrated on three main areas, loosely categorized as image synthesis, shape modeling, and behavioral modeling. While image synthesis (rendering) was stressed in the late 70s and early 80s, the emphasis has recently shifted to the modeling of various objects and phenomena — indeed, many researchers believe that graphics today *is* modeling. We wish to expand the definition of "modeling" to include the realms of simulation, animation, rendering, and user interaction.

Since the mid-60s, our research has focused on tools for creating electronic books, specifically hypermedia documents with interactive illustrations [21]. Such illustrations allow readers to interact not just with a canned "movie" but with a stored, parameterized model of a phenomenon they are trying to understand. Interactive illustrations require simulation and animation of the underlying model in an interactive, real-time environment.

Because we want to create interactive illustrations for a wide range of topics, our modeling tools must handle large (and extensible) sets of objects and operations on those objects that change any of their physical attributes over time. We need a rich collection of methods for controlling the time-varying structure and behavior of the objects, especially as they interact under various application-dependent systems of rules. In other words, we cannot use a single, "silver-bullet" modeling or animation technique.

The essence of animation control is the specification of time-varying properties. Traditional graphics packages (such as PHIGS+, Doré, and RenderMan), however, have no explicit notion of time. In these packages, time can be specified only implicitly, as the byproduct of a sequence of editing operations on the database or display-list representation. While today's animation systems do allow time to be explicitly specified, they generally permit only a subset of an object's properties to vary over time. They also tend to be restricted in the objects, properties and behaviors they support. Most limiting, they force the designer to conceptualize the animation process as a traditional pipeline of modeling, animation, and rendering phases. By contrast, we have concentrated on integrating modeling, animation, and rendering into a unified, coherent framework.

2 An Overview

Our system provides a general and extensible set of objects that may have geometric, algorithmic, or interactive (i.e., user-interface-controlled) properties. Geometric objects include quadrics, superquadrics, constructive solid geometry objects (CSGs) and other hierarchical collections of objects, spline patches, objects of revolution, prisms, generalized cylinders or ducts [9] (objects obtained by extruding a varying cross-section along a spline path), and implicit surfaces. Non-geometric objects include cameras, lights, and renderers. Behaviors such as gestural controls, spring constraints, finite-element techniques for cloth simulation [19], dynamics [13], inverse kinematics [4] [14], and constraint solvers [3] are also encapsulated as objects.

Objects can send and receive messages. These messages are persistent — a copy of each message is retained in the receiving object. They provide information on how an object should change itself over time. Objects can also inquire information from each other, information that depends on the nature and content of the messages a particular object has retained. Through messages, objects can be transformed (with scales, rotations, translations, shears,

and reflections), deformed (with bends, twists, tapers, waves [2], and free-form deformations [16]), colored, shaded [17], texture-mapped, dynamically moved (with forces, torques, velocities, and accelerations), and volumetrically carved [8].

Messages are functions of time and, since objects retain them, they may be edited. An object's list of messages describes the object's time-varying structure and behavior. By editing this list, through inserting, deleting, adding, or modifying messages, that structure and behavior can be altered. Editing can be performed either by objects or by entities (e.g., a user) external to the set of objects comprising the model.

Our objects have several important characteristics. First, since they can have interactive properties, any object can have a graphical user interface as one of its attributes. Typically, an object supports a user interface for its own specialized information; for instance, a dynamics simulator may permit a user to specify its initial conditions with sliders. Other objects are primarily interactive, such as an object encapsulating a mouse which is queried by other objects for position information, or an object encapsulating a constraints editor.

Second, objects exploit communication, because they contain information that is often essential to other objects. A renderer needs information from other objects in order to make global lighting calculations. Constraint methods also require information from many objects to perform their calculations. Constructive solid geometry objects need information about the boundary representation of their component objects in order to compute their own boundary representations. A camera needs to know the position of another object in order to track it.

Finally, an object in our system is not part of a classical class-instance hierarchy, such as is found in the C++ and Smalltalk programming languages. The constantly changing nature of our models makes a static relationship such as class-instance too restrictive, since, for example, transforming a sphere into a torus would typically require a change in class. Instead, our system is a delegation system [18] [10]. In a class-instance system, objects have two sorts of associations: the association of an instance with its class and the association of a class with its super-class. A delegation system, on the other hand, has only one relation, that between an object and its prototype. An object, termed the *extension*, can be created from another object, its *prototype*; an object in a delegation system interprets a message by using one of its prototype's techniques. Changes to the prototype affect both objects, but changes to the extension affect only the extension. Although it has been suggested [5] [20] that delegation might provide a simpler and more elegant method of solving computer graphics problems, we are not aware of work prior to ours incorporating delegation into animation and modeling systems.

3 The System Architecture

3.1 Control points

At its simplest, a message is a name and a function of time. A message can be edited by changing the particular nature and form of its time-varying function, specified by the series of time-value pairs that we call *control points* (see Figure 1). Thus, editing a message can mean adding or removing control points, associating control points with new times, or changing the value in a control point.

The value in a control point may be scalars, vectors, or arbitrarily complex expressions. For example, a scaling transformation can be given as a single real number for a uniform scale or a list of three real numbers for a non-uniform scale. A control point specifying a CSG tree can be given as a list containing three items: an object name or a list (itself a CSG tree), an identifier specifying a CSG operation, and another object or list. Values can be functions of time: a translation can be given by a vector function of the position of another object in the scene. Function-based control points allow useful behaviors, such as objects following other objects or adjusting their colors to match those of other objects. Many systems support such tracking behavior as a special case for cameras, but our system allows any object to behave in this way.

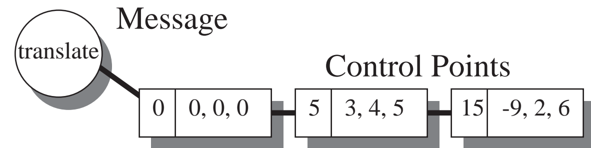

Figure 1: A message is a list of control points; a control point is a time and an associated value.

The CSG tree example above points out that the values of control points can be nested lists. Control-point values are very similar to Lisp s-expressions in this regard. They can contain atomic values, including numbers, strings, vectors, data structures, and object identifiers, or they can be lists of atomic values or other lists. Control point values are thus very flexible, permitting the use of mathematical expressions based on values in the scene. Figure 2 shows some sample control point values.

```
/*atomic values*/ 3.1416 "/u/john/paint.bi"
                  camera  sin(3.14 * 11.7)

/*a list of atoms*/ [1.0, 2.0, 3.0]
/*a list of lists*/ [[1,0,0],[2,[3.3,4.4]]]

/*functions*/
  /*obtain 2nd value of list*/
            select([1.0,sin(t),3.0],2)
  /*changinng position of an object*/
            robotHead.position
```

Figure 2: Examples of control point values (fragments of a scripting language used to describe objects and their messages).

Values at times not explicitly specified can be derived through an interpolation function, typically a weighted sum of control points. When interpolating a series of control points whose values are themselves functions of time, the system cannot just pass direct values to an interpolation method, but must first evaluate each control point at the target time. The series of evaluated values (not functions) thus produced can then be interpolated. As an example, consider a camera tracking two moving objects, smoothly shifting its focus from the first to the second. The value of a message over time can change dramatically, depending on the interpolation method used.

3.2 Messages

An important purpose of messages is to provide communication among objects. Objects such as a user interface or a physically based simulator apply and edit messages to other objects, thus modifying their appearance and behavior. An object can also affect another object by sending it a message containing a reference back to the sender (i.e., the message contains a control point that references another object). Whenever this message is interpreted, the sending object is called back and asked to provide the appropriate information. We use the term *controllers* for objects that modify messages on other objects, either by actively editing or passively being called back.

For example, large scientific visualization projects are often run in batch mode, separating a supercomputer analysis from an inter-

active visualization of the results. A simple controller could read in the results of such batch simulations, obtaining a list of positions (or whatever information is appropriate) and creating a list of messages from it. These messages could then be given to the appropriate objects, telling each how to behave in order to represent the scientific data [6]. Other more sophisticated controllers can apply and then edit a set of messages, adding new messages as they derive new results. Controllers will be discussed in more detail in Section 5.

Our system has a variety of messages to support its many kinds of objects. Highly specialized messages can provide information for an unusual object (for example, the parameters of an implicit equation or the tolerances of a constraint solver). More general changes, such as transformations, deformations, and dynamics (force or torque), provide a diverse class of changes applicable to more common objects.

3.3 Objects

Every object is represented by a retained list of all of the messages it has received. All objects are identical when first created, since no messages have yet been sent, i.e., a new object's list is empty. The message list is then modified in order to give the object interesting behaviors or appearances.

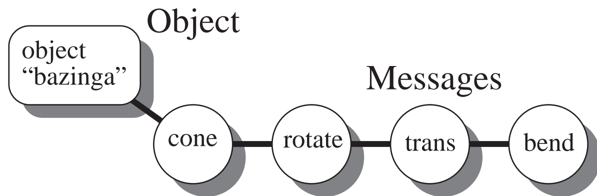

Figure 3: An object is a list of messages.

The interpretation of a message, i.e., its semantic meaning, is determined by the object that receives it. For example, a screen-aligned text object should not behave in the same way as a cube when it receives a message *rotate*; the rotation of the text should be projected onto the plane of the screen. Similarly, requests for information are handled in an object-specific manner: A sphere computes a ray intersection differently from a cube.

To handle variable semantics, an object has a set of methods (i.e., functions) to interpret messages that determine its behavior and appearance. Interpretation of a message can alter some or possibly all of the methods currently in use by an object, and thus can radically change its entire nature. This ability to change methods enables an object to adapt to different situations. For example, a deformed torus no longer performs ray intersections with itself by finding the roots of a quartic equation. Rather than requiring the torus's ray-intersection method to handle all eventualities, the procedure that interprets the deformation method changes its ray intersection method to a more suitable one, such as one that performs ray intersection with a set of polygons.

Many simple objects, such as spheres, cubes, and cylinders, have many methods in common. They handle most transformations identically and differ only in a few shape-specific methods, such as boundary representation, ray intersection, and computation of surface normals and parametric coordinates.

3.4 An example of making objects

A delegation system has straightforward mechanisms for object hierarchy [5] [10]. Recall that an object, the *extension*, can be made from another object, the *prototype*. In our system, an object can receive a message stating that it is to inherit all the messages of another object, thus becoming that object's extension. The extension implicitly inherits the prototype's behavior as well, since its methods are initially defined by the messages in the prototype. Since the prototype-extension relationship in our system is specified with a message, however, it can vary over time, a feature not normally present in a delegation system. For example, the history of automobiles can be modeled as a single object that uses a different model year car as its prototype for each year. This behavior is described very simply by a single time-varying message.

Objects can also be made from several objects. Consider a figure sitting in a chair, shown in figure 7. The chair is a CSG object, built from the parts of many different objects. The figure is made of several extruded duct objects, giving it a smooth, stylized appearance. A duct is itself a hierarchy of several objects, made from several spline path objects: one path describes the spine of the duct while the others describe the duct's cross-section along the length of that path. Paths are themselves composed of point objects used as the control points for the splines.

If the objects composing the CSG object change over time, the CSG object itself will also change. The CSG object asks its components for their boundary representations at a specified time, and the component objects return the boundary representations as functions of time, since the boundaries are specified by messages to the corresponding object. The composition of these functions in a CSG object is necessarily a function of time. Paths and ducts behave similarly: as the points specifying the hull of a path move, the hulls change shape, changing the spline path. As a path changes its orientation and shape, a duct made from this path also changes.

Suppose we want the figure to watch a fly a fly flitting about the scene. We can specify the motion of the fly with a path object, making the fly move along a spline path, and the fly can ask the path for the position of points further along and for tangent information at those points. Thus the fly can be oriented along the path, as if it were flying through the air. Likewise, the figure's eyes can ask the fly for its position and use that information to track it, and the points and paths comprising the figure's ducts can also ask the fly for its position and change their orientation accordingly.

4 Interpreting Messages

4.1 The Simple Case

An object computes the answer to an inquiry by interpreting each of its messages in sequence. As we said earlier, a message can change the methods used to interpret subsequent messages; therefore, the particular order of messages is important. The object's message list itself provides this ordering. This linear traversal is satisfactory until we begin using references to other objects and making multiple inquiries of an object. Under these circumstances, work will be repeated unnecessarily, and it becomes useful to exploit coherence, as discussed in Section 4.2.

Recall that some messages, like the deformations mentioned in Section 2, will, when interpreted, change the methods the object uses to interpret messages further along in the list. Note also that objects change their methods in different situations. For example, applying a deformation to a spline patch might not cause it to change its methods if the inaccuracy of applying the deformation to the control hull (and not the patch itself) is acceptable [7].

When objects depend on other objects, traversal becomes recursive. Consider the figure watching a fly discussed in Section 3.4. To determine the orientation of an eye, the position of the fly must be determined. When, in interpreting the messages of the eye, the message containing the reference to the fly (i.e., asking the fly for its position) is reached, a recursive traversal of the fly begins. The fly's position is determined by interpreting the fly's list of messages, and then interpretation of the eye's list continues.

The interpretation mechanism implicitly utilizes lazy evaluation: no calculations are performed until an object is actively asked for information. For example, time would be wasted in computing

polygonal boundary representations of CSG objects if all inquiries concerned the tree hierarchy of the object. We use lazy evaluation and the caching scheme described next to simultaneously avoid unnecessary computation and exploit inter- and intra-frame coherency.

4.2 Caching

Messages can be used to store arbitrary information. Objects can send messages to themselves in order to cache useful but computationally expensive data. Since such data is a function of time and messages are also functions of time, messages are an appropriate mechanism for data caching.

The first time an inquiry is made, the object computes the value for the time of inquiry and the time interval over which that value holds. If a second inquiry is made within the valid interval, the object simply returns the previously computed value (see Figure 4). Note that some messages contain data relevant to the cache. If such a message is modified, the cache is marked as invalid (see Figure 5). Multiple edits to these messages simply flag the cache as invalid multiple times, a very cheap operation. Thus, several edits can be "batched" into one, and interactive updates become much faster, since the data is not recomputed until actually requested.

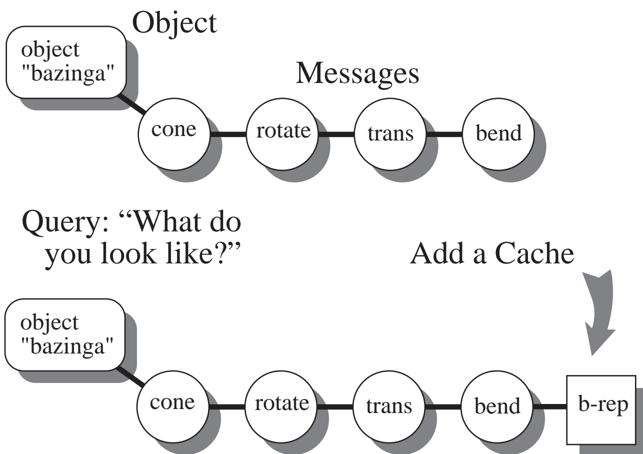

Figure 4: An inquiry adds a cache to the end of a list of messages. Here, the cache is of a boundary representation (b-rep) of the object.

Editing and subsequently invalidating a cache is a selective process: editing a translation invalidates only a cache of a transformation matrix, not a cache of a polygonal boundary representation. Objects understand how different messages affect each other. In particular, they know which messages invalidate which caches. Further, since each cache stores the interval over which it is valid, invalidation may merely change the size and shape of that interval (perhaps splitting it into multiple intervals) instead of completely invalidating it. If the edit changes the value of a message halfway through the time span of the cache's interval, the interval will be halved. More detailed manipulations of intervals are also supported, such as scaling and Boolean operations.

Profiling indicates that the improvement in performance with caching more than justifies its expense. By monitoring memory usage, we have seen that animations using extensive caching use approximately thirty percent more memory but achieve as much as a tenfold speedup. In these animations, the caching mechanism caches essentially *all* inquired data, for any time of inquiry, thereby minimizing the need to recalculate coherent data. Note that inter-frame coherency is automatically exploited, since caches are valid over intervals of time.

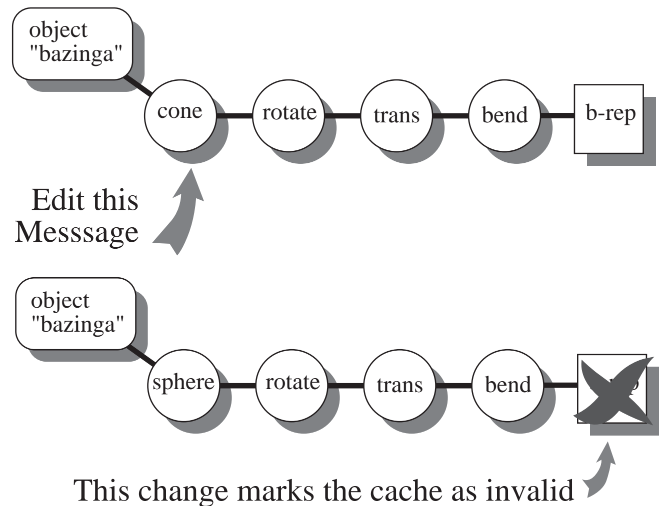

Figure 5: Editing a message before a cache can invalidate the cache.

4.3 Message list traversal with caching

To see how the system works with caching, let's consider a simple example: rendering all objects in a scene with a z-buffer. The renderer asks each object in the frame for its polygonal boundary representation. When inquired for the first time, each object computes its polygonal representation and caches it, marking it with the interval over which it is valid, and then gives the data to the renderer. The renderer then z-buffers the polygons, producing a frame. When the renderer asks an object for its polygons in the next frame, the object merely returns the previously computed boundary representation, if it is valid for the requested frame.

Caching helps expedite inquiry within a frame as well. If we ask the same object more than once for the same data, inquiries after the first will perform much faster by just returning the cache. Consider a car with four tires, each alike (up to a linear transformation). The tire could be an expensive-to-compute spline surface, yet this surface could be computed only once, not four times.

As another example, consider updating the figure in the fly animation of Section 3.4. Suppose one of the points used in a path is translated. This translation invalidates the point's CTM cache (current transformation matrix). Since caches are generated when an object is asked for information, the identity of the inquiring object (here, the path) is also stored in the cache. Thus, when the cache is invalidated, the path is informed and invalidates its own caches. These caches include references to the duct made from the path, so the duct's cache of its boundary representation is also marked as invalid. When the duct needs to provide its polygonal representation again, it will see its invalid cache and retraverse its list of messages, asking the path for the spline equations. The spline will notice its own invalid cache and recalculate the spline equations, asking the moved point for its new position.

5 Controllers

As mentioned in Section 2, behavior can be encapsulated in objects we call controllers. A controller affects other objects by sending them messages that refer back to the controller. Consider an inverse kinematics controller that must make an articulated arm reach for a goal (see Figure 8). Each joint is an object with a message that specifies its orientation through a reference to the inverse kinematics controller. This dependency allows the controller to indicate the amount of translation and rotation produced by any joint at any

given time. Interactive techniques can be considered controllers as well, for example, when a user specifies the initial conditions of a simulation (see Figure 9). In this case, the simulated objects reference a user interface object.

The use of dependencies in this situation is similar to that described in Section 4.1. Thus, when the position of an object in the linkage is needed at a particular time t, the serial interpretation of the object's messages begins. Upon reaching the translation or rotation message referencing the controller, the object asks the controller for the correct value, and the controller then supplies the necessary translation or rotation for the given time t.

The messages sent to a controlled object by a controller are intentionally as abstract as possible. The responsibility of determining how these messages affect a controlled object is left to the object itself. Essentially, a controller determines *what* to do and a controlled object determines *how* to do it. Consider, for example, a rigid-body dynamics simulation involving collision detection and response. It would be possible to have one controller handle all aspects of the simulation, exerting control by sending only translation and rotation messages to the controlled objects. However, we use instead a collision-response controller that sends a "collision" message to each controlled object: the object itself interprets the details of the collision message in terms of velocity (for momentum transfers) or acceleration (for continuous contact).

This object-oriented approach to control has several advantages. First, it reduces the complexity of controller implementation. A controller need not keep track of how it is actually changing the specific attributes of a controlled object, and this controller thus can store less information than might be needed by another controller affecting the same object, reducing the need for communication between controllers. Second, our approach increases the efficiency of communication between controllers and objects. A single abstract message that directs changes to many attributes of an object requires less system overhead than many messages each of which concerns only a single attribute. Third, our approach allows different types of objects to respond differently to the same abstract command, so that a flexible object like cloth, for instance, can interpret a collision message differently from a rigid object.

6 Controller Interaction

Allowing heterogeneous controllers to coexist and communicate in the same environment has been a research goal in computer graphics for several years [1] [12]. Such interaction between controllers should allow many powerful behavioral control techniques to affect a common set of objects in a meaningful way. The ideal system should be flexible, extensible and efficient.

6.1 Problems with interaction

A number of difficult problems must be solved to achieve the goal of heterogenous controller interaction. The first involves identifying the aspects of interacting controllers that hinder successful cooperation.

Consider the situation depicted in Figure 6. The rod with endpoints a and b has length l. The distance between the two walls A and B is also l, so it should be possible to make the rod span the walls. Assume that we have a controller that can move a to A and a controller that can move b to B, and assume also that we constrain the rod to remain rigid. The simplest way to make the rod span the walls is to invoke the two controllers independently so that each handles the task of moving one endpoint to the appropriate wall. This solution does not necessarily work, however. Each controller might, for instance, decide that the simplest way to move an endpoint to the wall is to translate the rigid rod. Neither controller will realize that the rod must be rotated to allow both endpoints to touch the walls, so spanning will not be achieved. This problem cannot be solved until the controllers consider both endpoints. Diagnosing

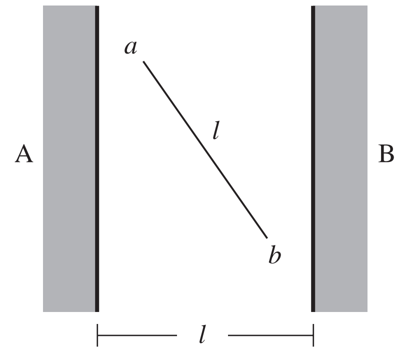

Figure 6: An example of incompatible controllers.

and correcting this lack of cooperation currently requires human intervention, and automation of the process does not seem feasible without severely restricting the possible controller types (e.g., to constraint solvers).

Another issue in controller interaction is data incompatibility. This arises, for example, when a kinematic controller and a dynamic controller both affect the same object. The dynamic controller works with velocity and acceleration data that the kinematic controller does not understand. When the kinematic controller changes the position of the object over time, however, the velocity of the object *appears* to change. If the dynamic controller is not made aware of this apparent change in velocity, its computations may produce visual inconsistencies.

Controllers that work iteratively present a third problem. The iterations of different controllers may proceed at different rates, and aliasing may result. This problem can appear when several controllers perform numerical integration with different time steps.

Our system provides several mechanisms that facilitate controller interaction, while trying to handle some of these problems. These mechanisms allow us to conduct further research into interaction policies.

6.2 Some solutions

Certain attributes of an object are dependent on its other attributes. The object's position, for instance, is related to its velocity and acceleration. Our system provides a mechanism to maintain the relationships among the attributes of an object. When a controller inquires an attribute of a controlled object, the system keeps that attribute consistent with any related attributes, even if those attributes have been changed by other controllers. Consider, for example, a controller that handles momentum transfers in response to collisions. This controller adds a collision-response message to each object it controls; the velocity resulting from the message is nonzero only if the object has just penetrated another object. If some other controller needs to know the position of an object whose momentum was changed, the controlled object will convert the change in momentum to a change in velocity; the collision-response controller need not be concerned with this issue.

Caching and its relation to controllers are especially important for attributes related by differential equations (parameterized by time). By caching the acceleration and velocity of an object at one instant in time, the system can use Euler's method [15] to obtain the velocity and position of that object at the next instant. We are currently investigating how to use the Runge-Kutta method of integration. This method is more effective overall than Euler's method, but because of our interobject dependencies, computing the necessary intermediate values may require global information; thus it cannot be implemented as easily in our system, which distributes

this global information and its interpretation among objects.

Caching is also useful for the numerical differentiation of attributes, helpful in solving the data incompatibility between kinematics and dynamics mentioned at the start of this section. A velocity corresponding to a kinematic change can be approximated by dividing the most recent displacement created by the kinematics controller by the elapsed time (in the animation's time units) from the previous displacement.

A variety of interesting controller interactions can be created simply by using the mechanisms for maintaining related attributes. When multiple controllers send messages to the same object, the relative ordering of the messages determines how the effects of the controllers combine to determine the object's overall behavior: messages earlier in the list affect the object first, because of the order of traversal. We call this type of interaction *strict priority ordering*. By allowing different orderings of the messages, the system can provide different effects, since messages are in general non-commutative (e.g., a translate followed by a rotate is not the same as a rotate followed by a translate).

A more general mechanism for controller interaction is supported by allowing multiple controllers to affect one or more objects indirectly through an intermediary controller. The multiple controllers are referenced by messages to the intermediary controller, not to the controlled object, and the controlled objects reference only the intermediary controller. The job of the intermediary controller is to combine the effects of the multiple controllers into a meaningful result and convey this result to the controlled objects. Such intermediary controllers can be used whenever strict priority ordering is not sufficient, e.g., when the behavior of an object should be the weighted average of the effects of two controllers. A toolbox of standard intermediary controllers that perform useful functions (such as the weighted average) could be added to the system.

7 An Example — 3D Pong

To see how this system works in practice, let's look at a sample interactive environment: a 3D pong game in which a sphere represents the ball, and two cylinders, appropriately scaled and translated, represent the paddles. We can play our game inside a court that is an object of revolution lying on its side, using scaled cylinders capping the ends of the revolve object for the back walls, the ones that the ball should not hit.

Two pairs of dial objects control the paddles, so that two users can manipulate the paddles next to the appropriate walls. A collision-detection object checks for intersections between the ball, the revolve object, the back walls, and the paddles. Finally, collision-response objects tell different objects what to do. One is responsible for collisions between the ball and anything except the back walls: when the ball hits the object of revolution, this object produces physically based collision-response messages. The ball then moves accordingly, while the revolve object uses a null interpretation of the response message and thus is unaffected by collisions. When the ball collides with a paddle, it uses the same collision-response interpretation as before, but the paddle would use its own, a different one, perhaps one that makes the paddle visually light up. A second collision-response object will be responsible for collisions between the ball and the back walls. When the ball hits one of the back walls, the ball will receive a different kind of collision response message, since it will not bounce back. The wall's collision response interpretation will add a point to the current score.

Many interesting features could be added to this game. For example, the walls of the court could change as the game progressed (possibly in response to collisions with the ball). The ball could move faster when hit by a faster paddle, despite the fact that the paddles are under kinematic control and have no intrinsic notion of velocity. Users could control their paddles in different ways, for example, using polar coordinates or Euclidean coordinates, simply by changing the messages between the dials and the paddles. Moving obstacles could be placed between the two players, perhaps obstacles that follow the ball, or follow a pre-scripted plan of motion. Since objects can be asked to display themselves at any time, instant replay of a game works automatically, allowing users to see what they just did, and change it. Finally, these features can be added interactively by the game players.

8 Summary

We have designed and implemented an interactive graphics system [11] with an unusually close integration of modeling and animation. All modeling is done through time-varying messages and thus all modeling tools can be used for animation. The system is object-oriented and provides a time-varying delegation hierarchy for maximum flexibility.

Behavioral control is supported by giving controllers the responsibility for calculating what controlled objects are to do, while letting each object interpret the abstract instructions according to its own methods. Multiple controllers can operate independently by instructing their objects in priority order. Alternatively, intermediary controllers can be written to integrate the behavior of control mechanisms. The system currently contains a large class of geometric primitives and a growing collection of user-interface objects and controllers.

A number of efficiency mechanisms contribute to interactive performance. In particular, lazy evaluation and caching exploit all inherent inter- and intra-frame coherence. By distributing the database, we expect to further improve the performance of the system. It should be possible for each object to evaluate its messages in parallel, but we will have to consider scheduling problems when objects depend on each other.

Our system is meant to go the next step beyond the scope of traditional graphics systems such as PHIGS+ or Doré. Such systems enforce a rigidly divided modeling/animation/rendering pipeline. We believe our system provides some indications of where the next generation of graphics systems software is headed: towards an environment with both time and behavior as first-class notions, and not just shape description and rendering.

9 Acknowledgements

We cannot begin to thank all the people that have made a system of this complexity possible. We would like to thank the many people who have commented on this paper, especially the reviewers. We would also like to thank Paul Strauss and Michael Natkin, architects of an earlier version of the system that provided much insight into the problem of a general animation system. In addition, the entire Brown Graphics Group, especially the artists, have provided much valuable criticism of the system's capabilities.

References

[1] Phil Amburn, Eric Grant, and Turner Whitted. Managing geometric complexity with enhanced procedural models. In *Proceedings of the ACM SIGGRAPH, Computer Graphics*, volume 20(4), pages 189–195, August 1986.

[2] Alan H. Barr. Global and local deformations of solid primitives. In *Proceedings of the ACM SIGGRAPH, Computer Graphics*, volume 18(3), pages 21–30, July 1984.

[3] Ronen Barzel and Alan H. Barr. A modeling system based on dynamic constraints. In *Proceedings of the ACM SIGGRAPH, Computer Graphics*, volume 22(4), pages 179–188, August 1988.

[4] Lisa K. Borden. Articulated objects in BAGS. Master's thesis, Brown University, May 1990.

[5] A. H. Borning. Classes versus prototypes in object-oriented languages. In *IEEE/ACM Fall Joint Computer Conference*, pages 36–40, 1986.

[6] Ingfei Chen and David Busath. Animating a cellular transport mechanism. *Pixel Magazine*, 1(2), 1990.

[7] Gerald Farin. *Curves and Surfaces for Computer-Aided Geometric Design*. Academic Press, second edition, 1990.

[8] Tinsley A. Galyean. Sculpt: Interactive volumetric modeling. Master's thesis, Brown University, May 1990.

[9] Andrew Glassner, editor. *Graphics Gems*. Academic Press, 1990.

[10] Brent Halperin and Van Nguyen. A model for object-based inheritance. In Peter Wegner and Bruce Shriver, editors, *Research Directions in Object-Oriented Programming*. The MIT Press, 1987.

[11] Philip M. Hubbard, Matthias M. Wloka, Robert C. Zeleznik, Daniel G. Aliaga, and Nathan Huang. UGA: A unified graphics architecture. Technical Report CS-91-30, Brown University, 1991.

[12] Devendra Kalra. *A Unified Framework for Constraint-Based Modeling*. PhD thesis, California Institute of Technology, 1990.

[13] Matthew Moore and Jane Wilhelms. Collision detection and response for computer animation. In *Proceedings of the ACM SIGGRAPH, Computer Graphics*, volume 22(4), pages 289–298, August 1988.

[14] Cary B. Phillips, Jianmin Zhao, and Norman I. Badler. Interactive real-time articulated figure manipulation using multiple kinematic constraints. In *Proceedings of the Symposium on Interactive 3D Graphics*, pages 245–250, 1990.

[15] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, 1988.

[16] T. W. Sederberg and S. R. Parry. Free-form deformation of solid geometric models. In *Proceedings of the ACM SIGGRAPH, Computer Graphics*, volume 20(4), pages 151–160, August 1986.

[17] Paul S. Strauss. A realistic lighting model for computer animators. *IEEE Computer Graphics and Applications*, 10(6), November 1990.

[18] Peter Wegner. The object-oriented classification paradigm. In Peter Wegner and Bruce Shriver, editors, *Research Directions in Object-Oriented Programming*. The MIT Press, 1987.

[19] Jerry Weil. A simplified approach to animating cloth objects. Unpublished report written for Optomystic, 1988.

[20] Peter Wisskirchen. *Object-Oriented Graphics*. Springer-Verlag, 1990.

[21] N. Yankelovich, N. Meyrowitz, and Andries van Dam. Reading and writing the electronic book. *IEEE Computer*, 18(10), October 1985.

Figure 7: A figure assembled from several spline paths. One defines a path for extrusion. Others define the shape of the extrusion at various points along the path of extrusion. Because of the rich dependencies of our system, such an object will change as the paths used to make it change.

Figure 8: Inverse kinematics controllers (affecting the robots) interacting with a finite-element simulation (affecting the cloth) in a radiosity-rendered room. All are part of the same database, and work together. For example, note that the cloth is being dragged off of the table.

Figure 9: Newton's cradle, a physically simulated toy. The user can specify initial conditions, or interact with the simulation as it procedes.