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

Robert C. Zeleznik
D. Brookshire Conner
Andries van Dam

Department of Computer Science
Brown University
Providence, Rhode Island 02912

Technical Report No. CS-91-10
January 1991

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

Robert C. Zeleznik D. Brookshire Conner Andries van Dam

January 22, 1991

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. Because all properties are functions of time, and because we integrate modeling and animation, we refer to our system as a "4D modeler." Changes in properties 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. The paper discusses some of the issues in modeling such interactions, and mechanisms implemented to provide solutions.

An object-oriented architecture uses delegation hierarchies to exploit multiple inheritance and to permit objects to change all of their attributes dynamically, including object type. Objects include displayable objects, controllers, cameras, lights, renderers, and user interfaces. Further, the delegation system permits network distribution of objects and their associated computation. Additional ways to obtain real-time update speeds include the use of data dependency networks, extensive caching to exploit inter- and intra-frame coherency, and lazy evaluation.

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

Additional Key Words and Phrases

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

 The authors gratefully acknowledge the support of this work by grants from IBM, NCR, Sun Microsystems, and Hewlett-Packard.

1 Introduction

Three major directions in graphics research over the last two decades may be loosely categorized as image synthesis, shape modeling, and behavioral modeling. While the area of image synthesis (i.e., rendering) received the most attention in the late seventies and early eighties, the emphasis has recently shifted to modeling of various objects and phenomena — indeed, many researchers believe that graphics today *is* modeling.[1] We wish to generalize "modeling" into the realms of simulation, animation, rendering, and user interaction.

Our research in graphics has focused since the mid-sixties on authoring tools for creating electronic books such as hypermedia documents with interactive illustrations [YMvD85]. 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. This kind of model includes inherent notions of time and behavior, so much so that we term it four dimensional (even though time cannot be treated in the same way as space). Interactive illustrations thus require interactive, real-time simulation and animation.

Because we want to create interactive illustrations for a wide range of topics, our tools to create 4D models must handle large (and extensible) sets of objects and operations on those objects that change any of their geometric or appearance attributes over time. We also need a rich collection of methods for controlling time-varying structure and behavior of the objects, especially as they interact under various application-dependent systems of rules. In other words, we cannot expect to 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 only be implicitly specified 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 geometric and appearance properties to vary over time. They also tend to be limited in the set of objects, properties and behaviors they support. Most importantly, they force the designer to conceptualize the animation process as a traditional pipeline of modeling, animation, and rendering phases.

We have instead concentrated on the integration of modeling, animation, and rendering into a unified, coherent framework, as well as on individual geometric and behavioral modeling techniques. Our system provides such an integrated framework as well as machinery for integrating various types of geometric and behavioral modeling techniques and their means of specification.

We first consider a sample problem, to see how a complex application requires a variety

[1] This is especially true since photorealism also requires very detailed geometric modeling.

of control paradigms working together. We then survey the functionality of our system, and give a more technical discussion of the system's mechanics. A discussion of efficiency considerations is next, with related discussions of object evaluation. We then detail the mechanisms implemented to support the integration of controllers, and conclude with comments on the current state of the implementation and our ideas for future work.

2 A Sample Problem

As an example of the kind of animation that we would like to see in an electronic book, consider a simulation of blood flow through the human body. The viewer would experience the circulatory process from the perspective of a blood cell. The important anatomical features such as the walls of the arteries and veins, the valves of the heart, and the other blood cells would all be represented. The viewer would be able to move about to investigate and interact with these features. The animation could be used for educational or diagnostic purposes, and the opportunities for viewer interaction would be tailored to the task at hand.

The creators of such an animation will need a variety of powerful modeling and behavioral-specification tools. To model the anatomical features, they will want convenient ways to manipulate smooth curves and irregular blobs. It must be possible to control the rigidity of the models so their tissue composition is accurately reflected. The creators will want to specify the motion of the models in high-level terms, letting the system handle the subtleties of making the motion look realistic. One of the simpler motions necessary will be the opening and closing of a heart valve. By modeling the valve as a rigid mechanical device, inverse kinematics may be sufficient for motion control; a more anatomically correct nonrigid model would require more advanced motion control, perhaps muscle modeling with dynamic constraints. Another important motion is the flow of blood cells through the fluid plasma; the system will need a fluid dynamics simulator to handle this sort of motion. Collision detection and response would be an important part of these motions, and the nonrigid models should deform appropriately. Finally, it must be possible to specify how the viewer can interact with the model. Allowable interactions could involve increasing the amount of plaque clogging the arteries to decrease the rate of blood flow, or removing designated parts of the model to reveal hidden structures.

3 An Overview

Everything in our system is treated as an object in the object-oriented programming sense. Visible objects include quadrics, superquadrics, constructive solid geometry objects and other hierarchical collections of objects, spline patches, objects of revolution, prisms, generalized cylinders (ducts), and implicit surfaces. Non-visible objects include cameras, lights, and

renderers. Behaviors such as gestural controls, spring constraints, finite-element methods for cloth simulation [Wei88], dynamics [MW88], inverse kinematics [Bor90] [PZB90], and constraint solvers [BB88], are also encapsulated as objects.

Geometric objects can be modified by interpreting a wide variety of modifications. Objects can be transformed (with scales, rotations, translations, shears, and reflections), deformed (with bends, twists, tapers, waves [Bar84], and free-form deformations [SP86]), colored, shaded [Str90], texture-mapped, dynamically moved (with forces, torques, velocities, and accelerations), and volumetrically carved [Gal90]. The system includes provisions to vary and interpolate values and parameters over time (linear, quadratic, spline and quaternion-based interpolation methods). Anything that can be modeled, even object type and the composition of hierarchical structures, can be varied over time.

Several properties of our objects are worth discussing. First, any object can have a graphical user interface. An object supports a user interface to its own specialized information; e.g., a dynamics simulator may permit a user to specify its initial conditions with sliders. For some objects, such as graphs of time-varying quantities, the interface is the primary purpose.

Second, all objects support and exploit communication. Objects contain information that is often essential to other objects. A renderer needs information from other objects in order to make global lighting calculations. Simulation methods also often require global information. Constructive solid geometry objects need information about the boundary representation of their component objects in order to compute their own boundary representation. A camera might need to know the position of another object, so that it can track it.

Third, 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 a 4D model makes such a static relationship as class-instance too restricting. For example, having a sphere become a torus requires a change in class. We use a delegation system instead [Weg87] [HN87]. It has suggested [Bor86] [Wis90] that delegation might provide a simpler more elegant method of solving problems of computer graphics, and our system demonstrates that this is indeed the case.

4 The System Architecture

4.1 Objects and classes

Every object is represented by its *state*, which contains a list of changes applied to the object. All objects, when first created, are identical and embryonic, with an empty state. The state must be modified in order to give the object interesting behaviors or appearances. Changes made to an object's state in order to change the object's properties are called *chops* (*ch*ange

*op*erators).

To determine how to apply a chop to an object, the semantics of the object must be taken into consideration. A screen-aligned text object, for instance, should not be rotated in the same manner as a cube; the rotation of the text must be projected onto the plane of the screen. Similarly, different kinds of objects must answer requests for information differently: a sphere should produce an answer in a different manner from a cube when asked to intersect a ray with itself. An object has a set of methods associated with it, used to interpret the chops and the inquiries. This set of methods determines the object's behavior; it is the object's *class*.

At any particular time, an object is characterized by a particular class. Like every other property of an object in the system, an object's class can change over time, since the class is associated with an object through a chop, much like any transformation or deformation. Clearly, an object's class is much more ephemeral than the class (i.e., data type) of an object in a typical object-oriented programming language like Smalltalk or C++. Classes in our system contain no local data, since an object's state contains all the information an object needs to answer questions about itself. Changing from one class to another involves changing the collection of methods being used, or even simply altering the contents of the current list of methods. Changing class in C++ involves a cast operation between possibly incompatible data structures, and is generally possible only in the special case of casting an instance of a class to an instance of its superclass.

This ability to change class enables an object to adapt to its situation. For example, a deformed torus can no longer perform ray intersections with itself by finding the roots of a quartic equation. Rather than require the torus ray-intersection method to handle all eventualities, the system lets the torus change to a class with a more suitable ray-intersection method, such as a class that performs ray intersection with a set of polygons. Note, however, that if the torus changes to an arbitrary class that performs ray intersection with polygons, it loses any notion of "torus-ness," clearly undesirable behavior. Instead of changing to an arbitrary polygon-based class, the torus should change to a "deformed torus" class. Now the radii of the torus can still be changed, even though the torus has been deformed and can no longer perform ray intersection by finding the roots of a quartic. This might seem to produce a large number of classes, but a class is simply a distinct list of methods, and thus changing one method (in this case, the ray intersection method) by exchanging the method in the list with the desired method is sufficient to change the class.

Classes are related to each other, in their own inheritance hierarchies. For example, many simple objects, such as spheres, cubes, and cylinders, have many methods in common. They will 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 coordinates. These objects inherit their common methods from a generic class. Since each

class is simply a list of methods, we allow this inheritance to occur on a method-by-method basis. A new class might conceivably have no unique methods, instead obtaining all its methods from different existing classes. The only restriction in such multiple inheritance is that cycles for a given method are prohibited.

4.2 Chops

At its simplest, a chop is a type code and a list of time-value pairs specifying the magnitude or specific nature of the chop as it varies over time. A chop thus acts as a time-varying specification of some property of an object. We call the time-value pairs comprising the chop *control points*, as discussed in the next section.

An important function of chops is communication between controllers and controlled objects. Large scientific visualization projects, for example, are often run in batch mode, separating a supercomputer analysis run and interactive visualization of the results. Such a system can interface with ours by simply providing a list of positions (or whatever information is appropriate). A group of objects can copy these positions into chops and produce a purely script-based animation [CB90].[2] Other more sophisticated controllers can apply a set of chops and then edit the values of those chops and add new ones as the controllers derive new results. (Controllers are discussed in more detail in Section 6.)

There is a variety of chops to support the many classes available. Class-specific parameters are specified through class-specific chops. More general changes, such as transformations, deformations, and dynamics (such as force and torque), provide controllers with a wide variety of ways to communicate with the objects they control.

4.3 Control points

The value field of a time-value pair in a control point may contain scalars, vectors, or arbitrarily complex expressions. For example, a scale 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. Such function-based control points allow useful behaviors like objects that follow or track other objects or adjust their color to match other objects. Many systems support such tracking behavior as a special case for cameras, but our system allows any object to behave in this fashion.

[2] As will be detailed shortly, every object has an ASCII representation. An ASCII representation of every object in a model comprises a script of that animation.

The example above involving CSG trees 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 rich, permitting the use of mathematical expressions based on values in the scene.

Values at times not specified explicitly can be derived using a weighted sum of control points, i.e., an interpolation method. 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. This produces a series of evaluated values that can then be interpolated. By using different interpolation methods, the value of the chop over time can change dramatically (see figure 1).

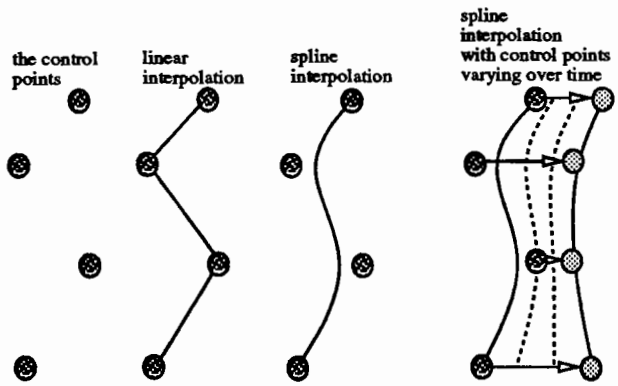

Figure 1: Different interpolation methods can make control points look like sample points, spline control points, or characteristic values of a function

4.4 An example of making objects

A delegation system has straightforward mechanisms for object hierarchy [Bor86] [HN87]. An object, the *extension*, is made from another, the *prototype*. Initially, the extension behaves identically to the prototype, although it can be modified to make it behave differently, or the prototype can be modified to make both objects change behavior. In our system, an object can be given a chop stating that the object is to inherit all the chops of another object, thus becoming that object's extension. The extension implicitly inherits the prototype's behavior as well, since the class is specified with a chop in the prototype's state. Since the prototype-extension relationship in our system is specified with a chop, it can vary over time, a feature not normally present in a delegation system.

Objects can also be made from several objects. Consider a figure sitting in a chair.[3] The chair could be a CSG made from several objects. If these objects change over time, the CSG itself will change over time. The figure's eyes can also be made from CSGs. The figure itself could be made of several duct objects, giving it a smooth, stylized appearance. A duct is made from several spline path objects: one path describes the spine of the duct, and others describe the duct's cross-section along the length of that path. Paths are themselves composed of point objects used as the control hull for the spline. As these points move, the hull changes shape, changing the spline path. As the paths change their orientation and shape, the ducts change shape.

Suppose a fly is flitting about the scene, and we want the figure to watch the fly. In our system, the fly can ask a spline path for the position of points progressively further along the path, and for tangent information at those points. Thus the fly can be oriented on the path, as if it were flying through the air. The figure's eyes can ask the fly for its position and use that information to track it. The points and paths comprising the ducts of the figure can also ask the fly for its position, and change their orientation using that information. This in turn affects the shape of the duct made from these splines, and the figure's head and eyes turn to follow the fly.

Objects such as CSGs, ducts, or implicit surfaces are dependent on several objects at once. As in standard multiple inheritance, such aggregate objects must decide which, if any, of their components' methods to use in responding to one of their own methods. When asked for surface attributes at a point, a CSG must determine which object the surface at that point came from, and ask the corresponding object for the surface attributes.

5 Evaluation Model

To use an object, we must ask it about its attributes. These attributes are not simply stored in a state vector we can index into — this would be insufficiently flexible for objects that can change radically and whose properties can depend on other objects. We must instead evaluate the object's list of abstract chops, taking into account their values as well as how the object's current class interprets those values.

An object can compute the answer to an inquiry directly by evaluating each of its chops in turn, starting at the beginning of its state and continuing forward until it reaches the end of its state. This simple *state traversal* is satisfactory until we begin utilizing references to other objects and making multiple inquiries of an object. At this point, work starts being repeated unnecessarily, and coherence must be incorporated.

[3]This scene is animation 1 in the accompanying video.

5.1 Simple state traversal

In principle, the system starts at the beginning of the state and evaluates each chop in the list in turn. In practice, we exploit coherency by caching data, which alters the simple execution model presented here (caching is explained in detail in the next section). Each chop is evaluated by obtaining its value through interpolation, then passing that value to the method corresponding to that chop. These methods are determined by the object's class, which is initially the default, generic class.

Some chops, like the deformations mentioned in Section 4.1, change the class of the object, and this changes the methods used to evaluate chops further along in the list (*during* the state traversal). Note that different classes change class in different situations. For example, a deformation applied to a spline patch might not cause a change to a "deformed patch" class, if the inaccuracy of applying the deformation to the control hull (and not the patch itself) is acceptable [Far90].

When objects depend on other objects, traversal of the state becomes a recursive procedure. Consider the figure following a fly discussed in Section 4.4. To determine the orientation of an eye, the position of the fly must be determined. When the state of the eye is evaluated and the chop containing a reference to the fly is reached, a recursive evaluation of the fly's state begins. The fly's position is determined, and then evaluation of the eye's state proceeds. Dependency cycles (i.e., A depends on B's position and B depends on A's) are not allowed. The next section discusses methods of improving this recursive evaluation and preventing duplication of work through caching data the first time it is computed.

The state traversal mechanism implicitly utilizes lazy evaluation. No calculations are performed until an object is actively asked for information. For example, there is no point in computing polygonal boundary representations of CSG objects if all inquiries are going to concern the tree hierarchy of the object. We utilize this lazy evaluation and the caching scheme discussed next simultaneously to avoid unnecessary computation and exploit inter- and intra-frame coherency. The first time an object is asked for a particular piece of data (and not before), it is computed, and it is then cached for later reuse.

5.2 Caching

Chops can be used to store arbitrary information. We let objects add chops to themselves in order to cache useful but computationally expensive data. The data is stored as the value of the control point, and the time value corresponds to the time interval over which the data is valid.

Thus the first time an inquiry is made, the object computes the value for the time of inquiry. If a second inquiry is made within the valid interval, the object simply returns the

pre-computed value (see Figure 2). Note that some chops contain data relevant to the cache. If such a chop is modified, the cache is marked as invalid (see Figure 3). Multiple edits to these chops simply flag the cache as invalid multiple times. Thus, several edits can be "batched" into one, and interactive updates become much faster, since the data isn't computed until actually requested.

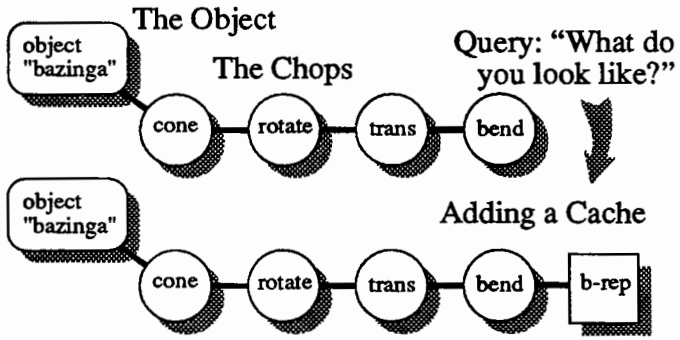

Figure 2: An inquiry adds a cache to the end of a state. In this example, the cache is of a boundary representation (b-rep) of the object.

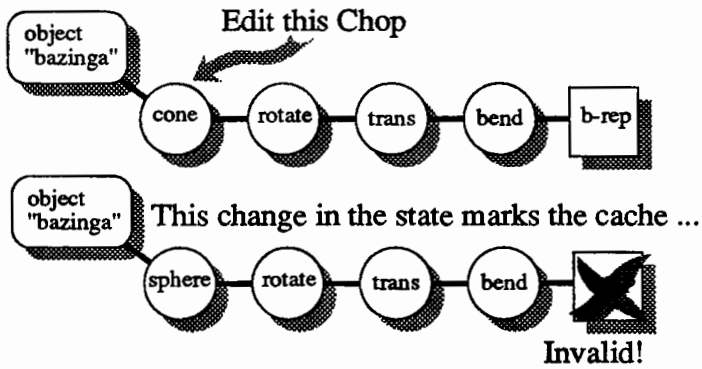

Figure 3: Editing a chop before a cache can invalidate the cache.

Caching and invalidation of caches is a selective process. For example, editing a translation invalidates only a cache of a transformation matrix, not a cache of a polygonal boundary representation. Each kind of chop for each object class contains information about what caches will be invalidated by editing the values of the chop's control points. Further, since each cache stores the interval over which it is valid, invalidation may merely reduce the size of the interval, instead of completely invalidating it. If the edit changes the value of a chop halfway through the time span of the cache's interval, the interval will be halved. There are provisions for more detailed manipulations of intervals, such as scaling them and performing

LIMITED CACHING			FULL CACHING		
what	% time	secs	what	% time	secs
State Traversal	15.4	16.96	State Traversal	43.8	11.8
Chop Inquiry	16.9	18.62	Chop Inquiry	18.8	5.1
State Inquiry	3.7	4.1	State Inquiry	17.4	4.69
State Find Caches	2.7	2.98	State Find Caches	15.7	4.23
Caching	5.6	6.17	Caching	15.6	4.2
Memory Allocation	42.0	46.27	Memory Allocation	8.9	2.4
Total Running Time = 110.19 secs			Total Running Time = 26.95 secs		

Table 1: Profiling statistics for Example 1, a scene with 12 objects, translating, rotating, and scaling over a period of eight hundred frames.

various Boolean operations.

Caching is not simple to implement, and thus the time-space tradeoffs must make it worthwhile. Profiling indicates that the increase in performance when caching is employed more than justifies its expense. The tables describing profiling statistics compare full caching with limited caching.[4] If caching were removed entirely, performance would quickly degrade further, since (as noted earlier) simple state traversal is clearly inefficient. Note that these statistics are produced by `gprof`, and some of the routines involved are recursive, and thus percentages do not add up to 100%.

Example 1 (Table 1) is a purely kinematic scene, demonstrating the performance of the system with classical animation. Example 2 (Table 2) is a model of a desktop toy, consisting of a framework supporting a series of heavy, dangling spheres and cylinders:[5] one ball is kinematically lifted and released, and its collisions with the other items trigger dynamic responses. Tables 1 and 2 are summarized in Table 3, which also includes a third example, a simple pendulum modeled with spring constraints. The increase in memory allocation times for limited caching reflects the time used to allocate structures already allocated in the full caching case.

The tables show that full caching is significantly faster. By monitoring memory usage (see Table 4), we see that animations using full caching use approximately thirty percent more memory and achieve as much as a ten-times speedup. The full caching mechanism essentially caches *all* inquired data for *every* frame of an animation, thereby minimizing the need to recalculate coherent data. For large simulations, this liberal caching scheme will become prohibitive due to the increased memory usage.

[4]Limited caching eliminates all caching of data except for the most outright computationally expensive results (polygonal boundary representation generation) and the caches necessary for dynamic simulations.

[5]This model can be seen as animation 3 on the accompanying video.

LIMITED CACHING			FULL CACHING		
what	% time	secs	what	% time	secs
State Traversal	33.1	194.45	State Traversal	37.5	15.56
Chop Inquiry	3.8	22.32	Chop Inquiry	5.6	2.324
State Inquiry	1.8	10.57	State Inquiry	1.5	0.62
State Find Caches	7.1	41.71	State Find Caches	5.9	2.45
Caching	3.4	19.97	Caching	8.7	3.61
Memory Allocation	39.0	229.12	Memory Allocation	9.0	3.74
Total Running Time = 587.45 secs			Total Running Time = 41.5 secs		

Table 2: Profiling statistics for Example 2, a scene of a desktop toy, a series of suspended objects of which one is lifted and allowed to fall into the others. Times are for a period of one hundred frames.

TIMING STATISTICS

	Example 1	Example 2	Example 3
Number of Frames	800	100	100
Dynamics?	No	Yes	Yes
Limited Caching	110.19	587.45	24.71
Full Caching	26.95	41.5	7.84
Ratio	0.245	0.07	0.317

Table 3: The time to run various examples, showing marked improvement when caching is employed. Example 3 is a simple dynamic pendulum.

Because this is a serious problem, we are currently investigating techniques to reduce the amount of caching performed and selectively remove cached values from an object's state. One important factor is the data type's cache relevance. Only data that is computationally expensive or frequently inquired should be cached. Another factor is the data type's cache history, which refers to the number of caches at previous times that are retained. For example, in a dynamic simulation, it is necessary to cache an object's velocity at the current time step in order to ascertain the object's new position at the next time step. Again, the current implementation of the system asserts that essentially all data is relevant and also caches the entire history of an object, hence the increase in memory usage reflected in the table. When cache relevance and cache history are utilized, the memory usage is expected to drop significantly. Because the system's architecture is flexible enough to allow any type of data to be cached, it will be up to the object class or even the user to determine exactly what values will be cached.

MEMORY STATISTICS

	Example 1	Example 2	Example 3
Limited Caching	1506640	218796	1081304
Full Caching	1960208	286200	1726172
% saved without cache	23 %	23 %	37 %

Table 4: Memory usage statistics, showing the low cost of additional caching.

5.3 State traversal with caching

To see how the system works with caching, let's go through a typical example: rendering all objects in a scene with a hardware z-buffer. The renderer asks each object in the frame for its polygonal representation. When first asked, each object computes the data 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 the next frame, the object can return the previously computed boundary representation, without computation.

If this sample z-buffer renders a CSG as well as several of the objects used to make the CSG the boundary representations of the subparts are only computed once. Either the z-buffer asks for the representation of the CSG which would ask the subpart for its representation (which would be cached and later returned to the z-buffer), or the renderer asks the subpart first, so that the subpart's boundary representation would be cached and quickly returned when the CSG needs it.

As another example, let us consider updating an object like the figure in the fly animation discussed in Section 4.4. Suppose one of the points used in a path is translated. This translation invalidates its cache of its CTM (current transformation matrix). Since caches are generated when an object is asked for information, the identity of the inquiring object (here the path) is stored in the cache as well. Thus, when the cache is invalidated, the path is informed and invalidates its own caches. These caches include references to the duct that is made from the path, so the duct's cache of its boundary representation is also marked as invalid. When the duct needs to display itself again, it will see its invalid cache and retraverse its state, 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.

Note that objects are relatively independent. Caches and their associated references form a data dependency network. The references within a cache point towards all objects that, directly or indirectly, depend on the properties of the object containing the cache. All objects that do not depend on any other objects can have their states traversed entirely in parallel. When dependencies do exist, a high degree of parallelism is still possible, although care must

be taken to avoid duplicating work. This is not hard to keep track of in a coarsely parallel situation, e.g., when the most expensive objects are being computed on a separate machine. We have left the full implementation of widespread parallel processing of the scene for future work. Selected object classes, such as a spline editor and an interactive ray tracer, can connect to other objects remotely using stream sockets and thus run in parallel across our network.

6 Controllers

As mentioned in Section 3, behavior can be encapsulated in objects we call controllers. A controller affects another object via chops in that object's state that refer to the controller. Consider an inverse kinematics controller that must make an articulated arm reach for a goal. The controller will determine the translations and rotations that must be applied to each joint and link of the arm to make it reach. Each joint or link will be represented by an object, with a rotation and translation chop referencing the controller in the state of each joint. This dependency allows the controller to indicate the amount of translation and rotation that will affect the object at any given time.

The use of dependencies in this situation is analogous to the use in Section 5.1. Specifically, when the position of an object in the linkage is needed at particular time t, state traversal is begun. Upon reaching the translation or rotation chop referencing the controller, the traversal will ask the controller for the correct value. The controller will then supply the necessary translation or rotation at time t.

The chops in a controlled object's state that reference a controller are intentionally as abstract as possible. Consider, for example, a controller that conducts a dynamic simulation to control an object. The controller could be referenced via kinematic chops, since the controller ultimately just rotates and translates the object. Instead, the controller is referenced via a "force chop" and a "torque chop." The object itself knows how to integrate forces and torques to obtain translations and rotations; this integration is performed during a traversal of the object's state. Flexibility is promoted when controllers communicate with objects via the abstract chops that are interpreted by the objects themselves. A cube object can respond to a force by simply translating itself, while a cloth object can respond by locally deforming only a part of itself. The controllers do not have to know about any of these special responses, so controller complexity is reduced. Thus, controllers specify *what* is to happen (e.g., a force is to be applied), whereas the controlled object determines *how* to respond.

7 Controller Interaction

It has been a research goal in computer graphics for several years to allow heterogeneous controllers to coexist and communicate in the same environment. Such integration of 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.

There are a number of difficult problems that must be solved to achieve this goal. The first involves identifying the aspects of interacting controllers that hinder successful cooperation. As a simple example, consider the situation depicted in Figure 4.

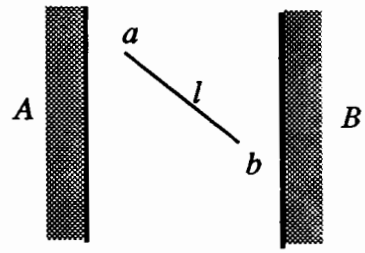

Figure 4: A example of incompatible controllers.

The rod with endpoints a and b has a length of l. The distance between the two walls A and B is also l, so it should be possible 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 seems to be as follows: invoke the two controllers independently and each will handle the task of moving one endpoint to the appropriate wall. This solution will 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 walls, so spanning will not be achieved. This problem will not be solvable until the controllers can be made to consider the status of both endpoints. Diagnosing and correcting this lack of cooperation currently requires human intervention, and automation of the process is not feasible in the near future.

The second problem with 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 deals 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 will *appear* to change. If the dynamic controller is not made aware of this apparent change in velocity, its computations may produce visual inconsistencies.

A third problem plagues the interaction of controllers that operate iteratively. 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.

7.1 Previous Work

A seminal system that integrated several heterogeneous modeling techniques was presented by Amburn, Grant and Whitted [AGW86]. Their system allowed procedural models to evolve while influencing each other. The models evolved to a static state and could not vary with time, but some of the issues involved in interactions of controllers over time were handled by this system.

Isaacs and Cohen [IC87] presented a system which integrated kinematic specification with dynamics. As long as the degrees of freedom of the system were either purely kinematically specified or purely dynamically specified, the system would fill in the unknowns in the motion equations. While this is not a general system for arbitrary controllers, its complexity serves to highlight the difficulties in getting even two methods of control to have a tight interaction, and to behave reasonably and predictably in unison.

Bolio is a system implemented to allow coarse integration of multiple controllers [BPZ87] [ZPS89]. Controllers interact through data dependencies, which are termed "constraints." When one controller receives a message that some data changed, it is free to update its internal state, and trigger additional constraints in other objects in response. This minimal protocol is sufficient for allowing heterogeneous controllers to coexist, but puts most of the burden of arbitration into the controllers themselves.

Kalra has proposed and implemented a prototype system for allowing heterogeneous constraint-solving controllers to interact on various levels [Kal90]. A user of that system is required to break up each new problem into separate subproblems that can be computed independently with simpler constraint-solving techniques. Kalra identified several broad classes of constraint interactions (i.e., independent, sequential, interdependent, temporally-sequenced), and suggested approaches for handling each class.

7.2 Interaction Mechanisms

Our system provides several mechanisms that facilitate controller interaction. These mechanisms allow us to conduct further research on devising interaction policies.

We have implemented a mechanism that automatically maintains *history* for the class of controllers that need it. History consists of data values from previous instants in time that are needed by a controller to compute a value at the current instant. An example of a con-

troller that needs history is a dynamic simulator that uses the velocity and acceleration of an object computed at a previous instant in order to perform numerical integration. The history mechanism allows the effects of controllers from this class to be combined in a meaningful manner. It also supports combination of these controllers with controllers that do not depend on history, e.g., inverse kinematics.

A difficult aspect of combining controllers that maintain history is keeping track of how one controller affects another controller's history. Two dynamics controllers, for example, might both change the velocity of a single object; inconsistencies will result if one controller does not know about the changes made by the other. To overcome this problem, each controller adds a special chop to the controlled object to indicate what data types constitute its history. The state traversal algorithm notices these special chops and maintains caches of the data types they indicate. As an example, consider an object affected by controllers that determine the object's velocity. A traversal of that object's state at time t will cause a cache to be built for the object's velocity at time t. During a subsequent traversal at time $t+1$, the cached velocity from time t is available to any controllers that need it; the cache is then updated with the new velocity.

Another difficulty arises because the history needed by a controller can be implicitly affected by a controller that does not care about history. This data incompatibility problem can arise when kinematics and dynamics interact, as discussed at the start of this section. A velocity corresponding to a kinematic change can be approximated by finding the most recent displacement created by the kinematics controller and dividing it by the elapsed time (in the animation's time units) from the previous displacement. This velocity approximation can be disabled when it is not desired.

The mechanisms for maintaining history allow a variety of interesting controller interactions to be created quite simply. To make a controller influence an object, only a few chops referencing the controller need to be added to the state of the object. These chops cause the controller to be invoked at the appropriate times (as described in Section 6). When multiple controllers have chops in the state of the same object, the relative ordering of the chops determines how the effects of the controllers combine to form the overall behavior of the object. A chop that comes earlier in the state will affect the object first, because of the order of state traversal. This type of interaction is known as *strict priority ordering*. By allowing different orderings, the system provides different effects. Consider a ball which is simultaneously subject to a kinematic translation and a dynamic spring force. If the translation chop precedes the chop referencing the spring controller, then the controller will see the translation and the spring force will pull against it. If, however, the spring's chop precedes the translation, the effect will be different. When a state traversal encounters the spring's chop, no translation will have occurred, so the spring will exert no force.

Our system also supports a more general mechanism for controller interaction. Multiple controllers can affect a single object indirectly through an intermediary controller. The multiple controllers are referenced by chops in the state of the intermediary controller instead of by chops in the state of the controlled object. The controlled object's state references only the intermediary controller. The job of the intermediary controller is to combine the effects of the multiple controllers into some meaningful result, and convey this result to the controlled object through its reference. Interaction objects 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. The main disadvantage of this approach is that creating an interaction controller requires programming. A toolbox of standard interaction controllers that perform useful functions (such as the weighted average) could be added to the system, however, and we intend to research this idea further.

7.3 Comparison to Previous Work

We provide more interaction functionality than the basic Bolio system. It is possible to build capabilities similar to ours on top of the basic Bolio system (and some of this work has already been completed), but we believe there are advantages to our approach. The efficiency measures inherent in our system, such as caching and lazy evaluation, are automatically utilized when controllers are interacting. Adding similar efficiency mechanisms on top of Bolio would be more work. Key features of our interaction mechanisms, such as the interpretation of chops and the maintenance of history, are associated with the objects being controlled, not the controllers themselves. This association reduces the complexity of the controllers, and facilitates efficient distributed processing at the level of the object (as mentioned in Section 5.3). Adding such a degree of distributed processing to Bolio would require extensive work because Bolio objects do not support the concept of a state or a chop.

Our system design arrives at many of the controller interaction mechanisms suggested by Kalra, albeit by different means. An advantage of our system is our emphasis on efficiency considerations. The lazy evaluation and caching inherent in our system should allow our system to come closer to the goal of real-time performance.

8 Summary

We have designed and implemented a 4D modeling system that provides an unusually close integration of modeling and animation. The system is object-oriented, and provides a delegation hierarchy to provide maximum flexibility. For example, a cylinder can be efficiently deformed by changing its class to a polygonal cylinder, and then carved with sculpting tools after changing its class to a volumetric cylinder. A number of efficiency mechanisms contribute

to real-time performance. In particular, we use lazy evaluation and cache results to take maximum advantage of inter- and intra-frame coherence. Behavioral control is supported by a strategy which gives a controller the responsibility for calculating what the objects under its control are to do, and then lets each object interpret the abstract instructions according to its own methods. Objects can be instructed to cache a history. Multiple controllers can operate independently by instructing their objects in priority order. Alternatively, special 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 graphical editors and controllers.

9 Future Work

Since our system is a framework for research, we have many opportunities for future work. One of our first projects is to finish the distribution of the database, which is currently limited to a few specialized classes. We plan to continue adding user interface objects and controllers, both those developed by ourselves, and those imported from other groups in the field, such as CalTech and MIT. Now that the framework is established, the path towards further research is clear.

References

[AGW86] Phil Amburn, Eric Grant, and Turner Whitted. Managing geometric complexity with enhanced procedural models. In *Proceedings of the ACM SIGGRAPH*, pages 189–195, August 1986.

[Bar81] Alan H. Barr. Superquadrics and angle-preserving transformations. *IEEE Computer Graphics and Applications*, 1(1), 1981.

[Bar84] Alan H. Barr. Global and local deformations of solid primitives. In *Proceedings of the ACM SIGGRAPH*, pages 21–30, July 1984.

[BB88] Ronen Barzel and Alan H. Barr. A modeling system based on dynamic constraints. In *Proceedings of the ACM SIGGRAPH*, pages 179–188, August 1988.

[Bor86] A. H. Borning. Classes versus prototypes in object-oriented languages. In *IEEE/ACM Fall Joint Computer Conference*, pages 36–40, 1986.

[Bor90] Lisa K. Borden. Articulated objects in BAGS. Master's thesis, Brown University, May 1990.

[BPZ87] Cliff Brett, Steve Pieper, and David Zeltzer. Putting it all together: An integrated package for viewing and editing 3d microworlds. In *Proceedings of the 4th Usenix Computer Conference*, October 1987.

[CB90] Ingfei Chen and David Busath. Animating a cellular transport mechanism. *Pixel Magazine*, 1(2), 1990.

[Far90] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design.* Academic Press, second edition, 1990.

[Gal90] Tinsley A. Galyean. Sculpt: Interactive volumetric modeling. Master's thesis, Brown University, May 1990.

[HN87] Brent Halperin and Van Nguyen. A model for object-based inheritance. In *Research Directions in Object-Oriented Programming.* The MIT Press, 1987.

[IC87] Paul M. Isaacs and Michael F. Cohen. Controlling dynamic simulation with kinematic constraints, behavior functions and inverse dynamics. In *Proceedings of the ACM SIGGRAPH*, pages 215–224, July 1987.

[Kal90] Devendra Kalra. *A Unified Framework for Constraint-Based Modeling.* PhD thesis, California Institute of Technology, 1990.

[MW88] Matthew Moore and Jane Wilhelms. Collision detection and response for computer animation. In *Proceedings of the ACM SIGGRAPH*, pages 289–298, August 1988.

[PZB90] 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.

[SP86] T. W. Sederberg and S. R. Parry. Free-form deformation of solid geometric models. In *Proceedings of the ACM SIGGRAPH*, pages 151–160, August 1986.

[Str90] Paul S. Strauss. A realistic lighting model for computer animators. *IEEE Computer Graphics and Applications*, 10(6), November 1990.

[Weg87] Peter Wegner. The object-oriented classification paradigm. In *Research Directions in Object-Oriented Programming.* The MIT Press, 1987.

[Wei88] Jerry Weil. A simplified approach to animating cloth objects. Unpublished report written for Optomystic, 1988.

[Wis90] Peter Wisskirchen. *Object-Oriented Graphics.* Springer-Verlag, 1990.

[WMW86] Brian Wyvill, Craig McPheeters, and Geoff Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4), 1986.

[YMvD85] N. Yankelovich, N. Meyrowitz, and Andries van Dam. Reading and writing the electronic book. *IEEE Computer*, 18(10), October 1985.

[ZPS89] David Zeltzer, Steve Pieper, and David J. Sturman. An integrated graphical simulation platform. In *Proceedings of Graphics Interface*, June 1989.