Integration of High-Level Animation Controls, Simulation Methods, and Gestural Specification

Brook Conner, Henry Kaufman, Matthias Wloka
Bob Zeleznik, Daniel G. Aliaga, Wm. Scott Draves
Philip M. Hubbard, Brian Knep, Michael J. Natkin
Paul S. Strauss, John F. Hughes, Andries van Dam *

September 9, 1990

Abstract

We present an interactive modeling and animation system that enables the close integration of a variety of simulation and animation paradigms. The system models diverse objects undergoing a wide variety of changes over time. These changes can be specified by different methods of control. Real-time update speeds are facilitated through maintenance of data dependency networks combined with a detailed caching and data invalidation scheme. The system is an extensible testbed, enabling research in the interaction of disparate controller types.

1 Introduction

The field of computer graphics has produced many useful methods of modeling objects and modeling changes over time. Modeling methods have been

*This work was sponsored in part by grants from IBM, Sun Microsystems, NCR, and Digital Equipment Corporation

devised for many kinds of objects, using techniques such as implicit surfaces, procedural models, and traditional solids modeling. Methods of modeling changes over time are equally diverse, encompasing traditional animation styles such as keyframing, inverse kinematics, constraint methods, and inverse dynamics.

What has been lacking is a consistent paradigm within which such modeling and animating methods can work together. General tools, such as traditional NURBS or solids based CAD/CAM systems, are typically very low-level, producing models that are a set of simple objects with little semantic meaning to the model. However, higher-level, procedural systems, such as grammar-based tree-modeling systems, are too specialized to handle arbitrary modeling and animation tasks.

BAGS, the Brown Animation Generation System, addresses this problem by enabling sophisticated objects to interact with complex methods of controlling changes over time in a flexible and extensible manner. We begin by describing the problems that **BAGS** was designed to solve, followed by a detailed description of the design and implementation of the system.

1.1 Electronic Books — The Goal of BAGS

The Brown Animation Generation System[Str88] was conceived as part of our ongoing research to develop the technologies necessary for interactive electronic books. We believe that electronic books used for teaching, research, and general instruction[1] will commonly involve "illustrations" that are animations of scientific, mathematical, and engineering visualizations[YMvD85]. **BAGS** is designed to create and allow interaction with this kind of illustration.

Many current media for animations in electronic books, like CD ROM and video disks, are limited, in that they present the user with a predefined sequence of frames, a canned movie. A book utilizing such an animation is inherently limited to presenting the information as the author thought it should be presented. But the author cannot and should not be expected to anticipate all possible situations in which the book might be used.

[1] We restrict ourselves somewhat, here, as there is no real reason "electronic books" cannot serve in every capacity that standard books do. We come from a background of technical and scientific illustration, but the system is not limited to such work.

An "illustration" in an electronic book must therefore be richly interactive. If it is not a canned animation, it allows the user to direct the animation, tailoring it to her needs. An abstract *model* of the phenomena being illustrated can be such an illustration. The model itself must be independent of any tools used to interact with it.

We characterize such a paradigm as *four-dimensional modeling*, since it specifically includes the specification of user-controlled, time-varying structure or behavior as an integral part of representing the model.

1.2 What is Four-Dimensional Modeling?

A system with separate modeling and animation phases makes it difficult to create animations that go beyond the application of linear transformations to rigid bodies. Rather than building a model, then allowing a small set of parameters to vary over time, a four-dimensional modeling system allows any part of the model to change at any point in time. Anything that can be modeled, can be modeled over time.

A full-blown four-dimensional modeling system is a highly flexible system to create a variety of non-trivial animations. But its very power presents problems in specification, especially when the system is used for electronic books. Interactive illustrations can be very useful and flexible tools. Current systems tend, however, to be difficult to use, having long and steep learning curves before the author becomes truly capable. We must reconcile this difference in the skill of the user with the complexity of the system by allowing the four-dimensional modeler to build animations using a wide variety of control methods[Zel89]. We must seek to minimize the necessary prerequisites. **BAGS** seeks to maintain a flexibility of specification, although our initial emphasis has been on designing the framework of the system and implementing track-based animation controls.

2 An Overview of BAGS

All models created with **BAGS** are expressed in the **SCEFO** (SCEne FOrmat) language. A **SCEFO** script is a human-readable text file which describes the set of *objects* that make up the four-dimensional model. Objects supported by **BAGS** include solids (cube, sphere, torus), surfaces (objects of revolution, spline patches, swept and extruded objects), and constructive solid geometry objects.

Changes over time are specified by applying *change operators* to the objects which modify some of their properties. Change operators include standard linear transformations, as well as deformations, and operators that specify visual properties and other object-specific parameters.

The values associated with the change operators are specified as time-value pairs (i.e., *control points*), which are interpolated with a user-specified interpolation method. Note that this script-based description is an editable modeling history, since the model is specified as a set of changes rather than an initial and final description (i.e., as sets of polygons, or as objects each with a single final transformation).

BAGS maintains a database for a given model. The database is an internal representation of the **SCEFO** script, consisting of a list of objects in the scene. Each object is associated with a list of changes. Thus, the database is a complete representation of a time-varying scene. The information can be presented either serially, as a movie, or in a random-access form at specific times.

The interface to this database is through a well-defined API (Application Program Interface). This API allows the transparent distribution of the model across multiple processors and even machines. The programmer/user can inform the database of changes to objects, create new objects, or inquire various relevant data about an object for a particular time. Flexible and powerful user interfaces to this database can be built on top of the API.

BAGS is designed in an object-oriented fashion. Thus, the API communicates directly with the objects in the scene. Each object is aware of the changes that are applied to it, and knows how to handle requests made by the API. Using the API, a program can apply changes to objects, or request information from objects, such as position, UV maps, or polygonal boundary representations.

The API interface also allows multiple users to access and change the database at the same time. Thus, several people can collaborate (possibly remotely) on creating and interacting with one complex scene. This interaction is not limited to persons, as the database can easily be changed by a program querying data through the API, making decisions based on that data, and then making changes to the data through the API.

The various tasks in creating a four-dimensional **SCEFO** model, such as building new objects, positioning them, and specifying their changes over time, all involve the manipulation of a subset of the information that defines the complete model. **BAGS** provides clients of the database specialized to present and interact with some particular aspect of the model. Typical clients include a modeling hierarchy view, a graph-based editor showing interpolation methods for change operator values, and a hardware-based solids modeler.

3 Objects as Structural Elements

Any scene must have displayable objects in it.[2] In **BAGS**, these objects are instances of *object classes*. When an object is created, its class is a default, generic class. An object can be explicitly assigned its class through the use of a special-purpose change operator we call a *chrep* (for *ch*ange *rep*resentation). Alternately, new instances can be told to inherit all the changes made to a previously created instance, including any chreps made to that object. This sort of inheritance can be compared to the object-oriented programming concept of delegation[Weg87][HN87], where a new object is created by *extending* a *prototype* object. Through extensions of prototypes, **BAGS** supports the usual notion of object hierarchy.

BAGS supercedes this hierarchy, however, because objects, once created from other objects, do not forget about their prototype objects. Changes can be applied to the prototype, *after* the new object has been created, and these changes will be reflected in the new object. This is enabled by **BAGS**'s manner of storing the changes applied to the object, and not simply the object itself. This provides **BAGS** with some very powerful features.

For example, consider a set of rivets (shown in figure 2), each created from a single prototype rivet. Then each rivet is transformed independently, spun around, scaled, bent, etc. We can now translate *every* rivet in the scene by translating the prototype rivet. This change will occur in world space if we apply it *after* the changes to the new rivets. A system like

[2]We will see examples of useful, non-displayable objects in section 4. Also, while it is possible to consider auditory models, or tactile models, or perhaps yet more esoteric methods of presenting data, we restrict ourselves to primarily visual data. It is worth noting that the methods presented here apply equally well to any method of data presentation, as the model is abstract.

PHIGS normally requires such changes to occur *before* the changes to the new rivets. We could perform such a change by switching the order of matrix concatenation, but then we could not apply changes to all the rivets *before* they were otherwise changed. We call such changes *out of order*, to reflect the flexibility in specification found in **BAGS**.

3.1 Changes to Objects

We refer to the list of changes applied to an object as its *state*. These changes completely determine every aspect of the object. The state tells the object both how to behave and how to appear at any point in time. Since the state completely describes the object, an object does not need any sort of "private data" internal to itself.

A chrep is a special change that associates a set of functions to the object. These functions have a one-to-one correspondance with functions supported by the database API. A general API call simply maps to the appropriate object-specific function. The set of functions also contains a function for each kind of change operator, providing a technique to evaluate it. Notice that a chrep is an example of delayed binding. The sets of functions are associated with the object at the last possible moment. We will return to this concept of sets of functions and their association to the object's class and state in section 3.2.

3.1.1 Change Operators

Change operators are ways of communicating arbitrary information to objects. They can also be used to store arbitrary data in the state, which proves useful in dependencies and hierarchies, when several objects might wish to obtain the same data from one object. Uses of this stored data will be discussed further in section 5.

Changes include standard linear transformations (scale, translate, rotate, shear, mirror, or an arbitrary transform specified by a matrix), nonlinear changes (twists, bends, waves, tapers, and free-form (Bernstein-Bézier) deformations[Far90]), and changes to affect surface attributes (color, reflectance, transparency, bumpiness, and others dependent on the lighting model).

Additionally, changes can specify *parameters* to objects. These parameters give data, in the form of name-value pairs, to an object that is relevant only to very specific classes. As an example, a super-quadric takes three parameters, giving toroidal radius and the two exponents.[Bar81] Objects that had been chreped to a super-quadric would see this named data and understand how to interpret it. Objects that had been chreped to another type of object would see the data, not recognize it, and thus ignore it.

3.1.2 Passing Information Between Objects

A direct extension to parameters is simple message-passing. By allowing objects to export data, we enable communication between objects. We have implemented an efficient message-passing system that allows arbitrary named values to be sent to instances of object classes, as well as exported from them. These values can be either atomic values, or lists of values (i.e., essentially a Lisp expression). Atomic values supported include numbers, strings, various functions, pointers to data structures, and object identitiers.

An example of exportable information is an object's current position. One object can ask another object to export its position. The receiving object can then use this value to perhaps set its own position, or set its orientation so that it "looks at" the second object, or it can use the exported value as part of a more complicated expression. In short, an object can export data about itself, and receive such data from other objects, which can then be used to set or alter any portion of the object's data. This provides a medium-level motion control technique, which can provide simple constraints.[3]

These messages are specified explicitly in the **SCEFO** script and are therefore stored as part of the model. The recipient object receives a new change operator, corresponding to the message. The sending object (presuming the message does not come from a database client) creates a new change operator for itself, so that it can remember who it sent the message to, and what it was. Thus, when an instance in a model is exporting some data, the database can make sure to notify any dependent instances (that is, an instance using the data) when the data changes. This data dependency network is discussed further in section 5.

[3]These constraints are typically geometric, but there is no reason the cannot be otherwise. As an example, consider an object that changes color as a function of another object's position.

3.2 Object Classes

BAGS supports a variety of object classes. By definition, each object class is determined by a set of methods. Methods determine how to handle change operators, as well as how to handle requests by the database API.

The class can be initially determined in one of two ways. If the first change to an object is a chrep operation, the class is determined explicitly. A chrep *is* an assignment of class. Alternately, an object can be used as a *prototype* of another object. The second object is an *extension* of the first. An object that is an extension of another object receives the entire state of the prototype object.[4]

By knowing its class, an object knows which set of methods it should apply to its data. We have discussed this before in reference to a set of functions that correspond to the database API. Setting an object's class thus determines how the object will respond to requests from the API.

The methods of a class can be viewed as operations on its state. A method takes the state as input and produces a value. This value could be a vector representing position, a matrix representing the current transformation applied to the object, or a set of polygons representing the shape of the object. The object is little more than a handle to the complete description found in the state.

3.2.1 Basic Objects

Object classes can be divided into two broad categories, basic and aggregate. Basic objects need no changes, other than the initial one telling them what class to act like. The class indicator is a complete definition for a basic object.

This definition does not rule out the possibility of object-specific parameters. It simply requires the object to behave in a reasonable fashion if the parameter is not specified. These defaults are coded into the methods of the class. Such an example is a super-quadric, which can take parameters indicating the squareness of the corners. If these are not specified, then the super-quadric should pick a reasonable default.

Other examples of basic objects supported by **BAGS** include spheres, cubes, cones, cylinders, torii, and points.

[4]Circularities, such as A extension of B, B extension of C, C extension of A, are not permitted

3.2.2 Aggregate Objects

Aggregate objects are sets of objects combined and treated as a single new object. The simplest aggregate is a group, several objects being treated as a single object. Another familiar aggregate is a CSG (constructive solid geometry) object. Other aggregates supported by **BAGS** include spline paths (composed of groups of points), prisms, objects of revolution (both small groups of spline paths), ducts (multiple cross-sections interpolated and extruded along an arbitrary path), bicubic patch meshes, and blobby surfaces[WMW86] (blending the surfaces of the objects of which it is aggregate).

Aggregates and their subparts maintain knowledge about the relationships between each other, in a fashion similar to the aforementioned prototype hierarchy. Consider a spline path. It is made up of several point objects, used as control points for the spline. Translating a point, after the path has been made, will change the shape of the path. If, in turn, an object of revolution has been created from this spline path, it too will change shape to reflect the change in the point. Had a CSG been created with this object of revolution, it too would change. The extreme example of this dependency is when every object in a scene is derived from a single root object.

3.3 Reinheritance

Recall that objects inherit their methods in one of two ways. First, it can be directly told what its methods are through the use of a chrep statement. Second, an object derived from another object through extension has as an implicit chrep, the chrep applied to its parent. This sort of inheritance is similar to inheritance in traditional object-oriented programming.

Certain changes, however, can modify this inheritance. A sphere that has been bent and twisted by deformation operators may not have the same properties it once had. Its center of mass, or its position, may have shifted. It probably cannot be rendered the same way it was before (for example, it can no longer be ray-traced by the same code). In short, its methods must deal with a new set of problems. This can be solved in two ways. Firstly, the methods can be arbitrarily general. This is not efficient, since the simplest method becomes burdened with a series of special cases. When changes are compounded, the number of special cases can become prohibitively high.

Instead, certain changes alter the set of methods associated with the object. In short, the objects change their inheritance in response to the changes applied to them. Note that the previous inheritance is not lost, as it can be derived from subsets of the state of the object. The correct set of methods to use can be determined by starting at the beginning of the state of an object and accumulating all chreps and all changes that otherwise change the set of methods. The sum total produces the correct set of methods to use when evaluating any change at any point in the object's state.

This scheme means that we are dynamically creating new object classes to fit the needs of the environment. Note that the user of a model does not see this creation. As far as the user is concerned, the object is still of the type specified by an earlier chrep. These new classes are the object's ways of performing more efficiently. A simple analogy is a trench digger digging trenches in soft dirt. The digger will use a shovel. However, when rocky soil is reached, the trench digger will probably switch to a pickaxe. Compare this to a sphere performing ray intersections with itself. At first, it will directly solve the ray intersections. When it is bent by a non-linear change, it will realize that the method (or tool, to maintain the analogy) it was using is no longer effective. It will use a new method, perhaps polygonizing itself, applying the deformation to the polygonal object, and using the ray intersection with that polygonal object as its own ray intersection.

The methods used in building object classes come from a common set of building block methods. The methods include functions such as polygonization routines, ray-object intersection routines, and transformation matrix accumulation routines for a variety of situations. Rather than organizing the methods as a structureless function pool, we have found that it makes more sense to provide them with a software hierarchy of their own. The structure we have provided is a traditional delegation structure, that is, the interface to the object does not change, only the set of methods that implement the interface at a particular instant may change.

Each initial class that an object can have (each class specified by a chrep statement) has a corresponding set of functions in the larger set of all building block functions. Each of these sets has other, smaller sets of overriding functions associated with it, corresponding to the modifications made in an object class in response to a particular change. For instance, the sphere has a direct ray intersection method, but it may also use the general polygon intersection method if the sphere is deformed. This structuring simplifies the addition of new initial object classes[5]. It also allows the set of change operators to be able to act on all available object classes in a reasonable way. Note that this structuring does not prevent us from gathering functions for a new class from any part of the source area, but this structure corresponds to the way the user conceptualizes what the different classes of objects are.

4 Objects as Controllers

As noted, **SCEFO** supports a number of intrinsic operations which may be used to change objects over time. A complete four-dimensional modeling system, however, should provide other, higher-level control methods.

We can implement control methods as new object classes, or as clients of the database. By using objects as controllers, we make the control method inherent in the model. The controller becomes part of the semantics of the model. By using clients as controllers, the semantics are missing from the model. Changes added or modified by a client have no identifying features to indicate the semantics of the client. The changes are normal change operators. A control object, however, encodes sematic meaning into the model, by being stated explicitly in the **SCEFO** script. Note that this conceptual distinction is not as distinct in the implementation of **BAGS**. Both implementations use the same message-passing techniques first described in section 3.1.2. They are just on opposite sides of the API. The choice of which to use becomes a decision based on the desired semantics of the model.

To dynamically simulate the interaction of the *visual* objects representing a ball and a wall, for example, we could create an instance of a dynamic simulator object and use the general message-passing system supported by object classes to pass the ball and the wall to the simulator object. The simulator object would then compute the interaction between the ball and the wall and update their positions accordingly. Other relevant information, such as the force of gravity to use, can also be passed to the simulator object using the message-passing abilities of **BAGS**. This simulator object could be as simple as a specialized ball-wall handler that could do nothing more, or a vastly more complicated arbitrary Newtonian dynamics system, or anywhere in between.

[5] Despite **BAGS**'s rich set of initial classes, we may wish to do this frequently. We discuss some reasonable new classes in section 4

We could also implement the same dynamics simulator as a client. It would be told by a user (or perhaps a program) what objects to consider. It would use the API to query information about the states of the relevant objects. When it had computed a solution, it would send information to the objects, telling them how to alter their states. Because objects in the database and clients all use the same message-passing protocol, integration of object-based behavior and client-based behavior is relatively easy.

4.1 When To Use Different Control Methods

Control objects lend themselves to automatic parts of a model, while control clients lend themselves to interactive parts of the model. A flag flying in the breeze can be easily implemented as a "cloth" object. The cloth would tell another object (most likely a spline patch) how to behave. Thus, the "cloth-ness" of the flag is inherent in the model. Alternately, constrained kinematic motion of objects, since it is more likely to be an interaction tool, (such as snap-dragging[Bie90]) is more logically implemented as a client of the database. In practice, how a controller is implemented is more a matter of convenience than anything else. By the nature of object communication and message passing, it makes little difference to the other objects in the scene whether they are talking to an object or to some external process. The objects simply receive messages and process them.

Control methods, whether clients or objects, can be used as either modeling tools or animation tools. When used as modeling tools, the methods continuously update the values of a single control point in a change to an object. If the model is being displayed at a *single* instant in time, this is visible to a user as continuous change in the scene. A simple example is a hardware dial controlling the translation of an object. When used as animation tools, controllers set the values of a *series* of control points. A simple example of this is a key-frame system, which sets values at a series of times entered by the user. Another example is a controller simulating gravity. Such a controller would set positions for the object at various times, according to the rules of gravity.

Since a controller is explicitly passed the objects it is to control, the scope of high-level motion-control methods can be restricted to the specific objects for which this motion control is desired. Consider a detailed room. We might wish to model a ball dropping in this room using dynamics. The simulation will be much more efficient if it only considers the ball and the floor, and ignores the walls and ceiling. The user or a heuristic program can determine that the ball will not interact with the walls and ceiling, and inform the simulator that it can safely ignore them. Such restriction improves the system performance perceived by the user when computationally expensive methods, like dynamic simulation[6], are used.

4.2 Controlling Controllers

The message-passing system also permits control objects to be passed to other control objects, allowing nesting of motion-control objects that can then work in unison. As an example of how this might be useful, let us consider a sailboat model. We wish to model the sails with a cloth simulator, and the motion of the boat through the water with a fluid dynamics simulator. These two simulators should, of course, affect each other's results, as the sail pulls on the boat, and the boat serves as the anchor points for the corners of the sail. A "master coordinator" control object could be passed the two simulator control objects, which have already been passed the visual objects comprising the sailboat and sail. The master coordinator could then interleave the execution of the simulators during their iterative processes, as a means of relaxing the system. We refer to such master coordinators as *controller-controllers.*

Such an object can be viewed as a generalization of aggregate structural objects. Thus, different ways of combining controllers become different classes of controller-controllers. One class might prioritize controllers, executing all time steps of higher priority objects first, while others might interleave time steps, and others might use yet-unthought-of schemes for interfacing controllers.

One particular area which we have yet to resolve is controllers so disparate that their data is incompatible. As an example, consider Newtonian dynamics (the usual $F = ma$ physics) versus Aristotelian physics (where force is proportional to velocity, not acceleration, thus $F = mv$). Newtonian systems will wish to communicate accelerations, but Aristotelian systems have no notion of acceleration, and thus would be unable to handle such

[6]Dynamic simulation is typically $O(n^2)$ in complexity, and simulation with dynamic constraints is typically $O(n^3)$

data. When controllers following such incompatible systems wish to control the same object, the desired results become unclear. While interleaving time steps might resolve some such differences, it remains to be seen whether it can solve all such incompatibilities.

5 Efficiency Concerns

Realistic images, or even non-realistic images of interestingly complex scenes, require prohibitively high bandwidths. Take for example something as basic as pattern-mapping. Perhaps we wish to paint on an object as part of an animation, changing the pattern with each frame[HH90]. With a few such objects in the scene, memory bandwidth requirements quickly exceed the capabilites of anything outside of custom hardware. We require high computational efficiency as well. As noted earlier, many desirable modeling techniques (such as constraint satisfaction or dynamic simulation) require large amounts of calculation.

We have implemented our system with a variety of mechanisms to speed update of all relevant data while minimizing bandwidth requirements. We cache data which we consider expensive to recompute on the fly. Additionally, objects keep track of what other objects depend on their state. Thus, when an object is changed in some fashion, it broadcasts this to all dependent objects, producing the minimum effort to keep the entire model valid. Updating the model is done with lazy evaluation, where data is tagged as simply out of date. If no one ever wishes to see the invalid data, we will not waste time computing it. Finally, the entire database system is constructed to be transparently distributed across multiple machines.

5.1 Caching Data

Objects create cache data whenever the memory expense of retaining data is less than the processor expense required to compute it. A clear case of when caching is useful is boundary representations of CSG objects, something which can be slow to compute[LTH86]. Caching a transformation matrix might not be as wise, unless the matrix was obtained by a laborious series of calculations.

Cached data is kept in the state of the object itself. Each type of data that might be cached is supported by a special change operator designed to cache that particular type of data. These cache "changes" store the data and a series of time intervals, indicating when the data is correct.

If an object is asked for the data in a cache for a time not covered by the cache's intervals, the object determines that it must compute the data. It may then decide to cache this *new* data with a new interval. Data in a cache can be invalidated for very specific portions of time by removing sub-intervals of the intervals. Different operations on the cache intervals have other effects. For example, the effects of a change can be scaled in time (slowed down, or speeded up) by scaling the interval in the corresponding cache.

To speed propagation of changes throughout the model, an object that is referenced (through templating, aggregation, or direct reference of exportable information) maintains a list of the objects that depend on its data. Like caches, these are stored as special change operators, containing the identity of the object as well as the nature of the dependant data. Such a list, when maintained throughout the entire system, is the functional equivalent of a data dependency network.

Let us examine a brief example, illustrated in figure 3. A cube and a camera exist in a scene. The camera asks the cube for its position, orienting itself to look at that point in space. When we move the cube, it traverses its own state, marking certain caches occuring after the move as invalid for the time interval of the move. Not all caches will be affected by these changes. For example, a boundary representation cache will not be invalidated, provided a transformation matrix is applied as well.

When the cube reaches the point in its state where the reference by the camera is noted, it will tell the camera that the data corresponding to the reference has changed. The camera will then, in turn, traverse its state, marking data rendered invalid by the change in the cube's position.

It is clear that that altering data will update only those portions of the database that need to be updated. When working with a large model, it is crucial that time not be wasted on parts that have not changed. Caching data is a method to exploit coherence in the model. Heuristics (hard-coded or determined at runtime) can decide what to cache in order to balance memory expenditures against compute expenditures.

5.2 Distribution Across a Network

We are often compute-bound when involving large simulations. Physically based modeling techniques are often of a complexity that is a polynomial in the number of objects involved in the simulation. We can increase our computing resources by using additional machines.

The **BAGS** database is designed to be distributed across many machines of different architectures using stream sockets. Any message-passing between objects, or between the database and clients (that is, any message-passing at all) has the capability to occur across machine boundaries. We perform this by replicating the database on all machines. This is not as expensive as it might first sound, as the replication is not a fully-detailed one.

When a new copy of the database is created on a machine, objects are duplicated, but without their full state information. Rather, they store the net address, indicating where the full details might be found. A client requiring sporadic data from this new database would receive the data through the same API, but the real source of the data is the remote machine. If the client requests data persistently, then the state of the object will be copied over to speed later requests.[7]

Data transmission is further improved by grouping related requests going across the network into *transactions*. Each transaction is a single data structure that is transmitted along the network. This saves overhead generated by network events.

Note that, like the lazy evaluation scheme for general data maintenance, the networking abilities balance memory efficiency with processor efficiency. Sporadic requests are handled across the network to conserve memory on the local machine. Persistent requests choose to optimize speed, since they are often indicative of a need for interactivity.

6 The Prototype Implementation

We currently have an interactive distributed modeling and animation system running on the **BAGS** substrate[Bro90]. We have a variety of networked

[7] Currently, the client explicitly indicates whether its requests are sporadic or persistent on an object by object basis. There is no reason that good application of heuristic techniques cannot automate this procedure. The interface to the API remains the same.

clients to edit all aspects of the database, from time-varying parameters to model hierarchy to three-dimensional gestural specification[Gol90][GGH+89]. Articulated bodies can be built and manipulated gesturally in real time using an inverse kinematics algorithm[Bor90]. As a further example, we have implemented a finite-element cloth simulator which can interact with other objects in the scene through attachment and nonpenetration constraints based on [Wei88]. Currently we are integrating a rigid body dynamics simulator into the system as well[MW88][Hah88].

BAGS is implemented entirely in the C programming language. We consider portability to be of high importance, and thus we regularly maintain the **BAGS** code on many different architectures. We currently work on Sun SPARCstations, Stardent GS1000's, Hewlett-Packard 835 Turbo SRX's, and IBM RISC System/6000 POWERstations. This desire for portability is one of the principle reasons we do not use the increasingly popular C++ programming language, as there are not yet suitable programming environments on all the architectures we use.

Commercial systems, such as Alias and Softimage, emphasize batch animations. Research systems tend to emphasize procedural models and forward-simulated, physically-based models[BPZ87][Kal90]. **BAGS** attempts to integrate simulations and animations into one coherent framework. This provides facilities for complex animations highly suited for interactive illustrations.

Our system, while still in progress, already shows a variety of efficient integration methods. The Brown Animation Generation System establishes a powerful, extensible testbed for further research into the questions raised by integration of disparate animation paradigms.

7 Future Work

We have presented the essential design of the **BAGS** kernel. Aside from the continuous debugging and tuning inherent in a new, large system, we consider the design structurally sound. It handles many useful classes of problems, and supports a very rich animation-creation environment.

The support provided by the system enables experimentation, and allows us to seek new methods of specifying complex models. We plan to search for, in particular, better ways of enabling the user to exploit the full power of the system. Complex dependencies, in particular, as well as subtleties of time-dependent behavior, seem to be difficult to represent in a viable user-interface.

Specifying the interrelationships between dissimilar modeling methods is also an area ripe for further study. Currently, we have no method of indicating complicated relations between modeling methods outside of the programmatic method of typing in **SCEFO** source directly. While this will always give the user the most power and flexibility, it is not the way users wish to interact with their models at all times.

The **BAGS** system provides a rich environment for further research in the interraction of dissimilar modeling methods. In the years to come, we will be seeking to integrate wider and wider classes of modeling paradigms, in progressively more useful ways.

Acknowledgements

BAGS is a large project, and we could not begin to thank all those who have helped, and are helping, with its completion. However, we would like to thank the entire Brown University Computer Graphics Group for their contributions, both as implementors of **BAGS**, and as users. We would also like to thank the technical support staff of the Brown University Computer Science Department, for their help in maintaining the hardware and operating system.

References

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

[Bie90] Eric A. Bier. Snap-dragging in three dimensions. In *Proceedings of the ACM SIGGRAPH*, pages 193–204, March 1990.

[Bor90] Lisa Kay Borden. Articulated objects in bags. Master's thesis, Brown University, 1990.

[BPZ87] Cliff Brett, Steve Pieper, and David Zeltzer. Putting it all together: An integrated package for viewing and editing 3d microworlds. Technical report, The Media Laboratory, Massachusetts Institute of Technology, 1987.

[Bro90] Brown University Computer Graphics Group. Demo reel, May 1990. Undistributed video tape.

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

[GGH+89] Tinsley Galyean, Melissa Gold, William Hsu, Henry Kaufman, and Mark Stern. Manipulation of virtual three-dimensional objects using two-dimensional input devices. Technical report, Brown University, 1989.

[Gol90] Melissa Gold. Multi-dimensional input devices and interaction techniques for a modeler-animator. Master's thesis, Brown University, 1990.

[Hah88] James K. Hahn. Realistic animation of rigid bodies. In *Proceedings of the ACM SIGGRAPH*, pages 299–308, August 1988.

[HH90] Pat Hanrahan and Paul Haeberli. Direct wysiwyg painting and texturing on 3d shapes. In *Proceedings of the ACM SIGGRAPH*, pages 215–223, August 1990.

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

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

[LTH86] David H. Laidlaw, Willaim B. Trumbore, and John F. Hughes. Constructive solid geometry for polyhedral objects. In *Proceedings of the ACM SIGGRAPH*, pages 161–170, August 1986.

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

[Str88] Paul S. Strauss. Bags: The brown animation generation system. Technical Report CSS-88-22, Brown University, 1988.

[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. written while at Optomystic, 1988.

[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.

[Zel89] David Zeltzer. Physically-based modeling: Past, present, and future. In *Proceedings of the ACM SIGGRAPH*, pages 201–203, December 1989.

```
read ("sphere.off") sphere; /* reading default objects from files */
read ("cube.off") cube;
read ("light.off") light;
read ("camera.off") camera;

template (egg) sphere; /* making an object from another one */

/* values to change operators are of the form: < time, value > */
/* values can be atomic values or lists {enclosed in braces} */
change (egg) translate <0, {1,1,1}>;
change (cube) translate <0, {-2,1,0}>,
   rotate <0, {{-2,1,0},{0,0,1},45}>;
change (camera) translate <0, {0,0,-4}>;

change (sphere) shade_val <0, {LASH_SURF_COLOR, {1,0,1}}>;
change (light) shade_val <0, {LASH_SRC_INT, {.9,1.9}}>,
      set_parameter <0, {"light_type", "directional"}>,
      rotate <0, {{0,0,0}, {1,0,0}, 60}>;
```

Figure 1: A simple **SCEFO** script, showing object creation, and application of changes

Figure 2: Light arrows are performed before extending the prototype, and dark ones are performed after. Note the difference in size and direction of the light arrows.

Figure 3: A simple data dependency relation between a cube and a camera