BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-91-M13

The Addition of Simulation to BAGS

by
Nathan Huang

The Addition of Simulation to BAGS

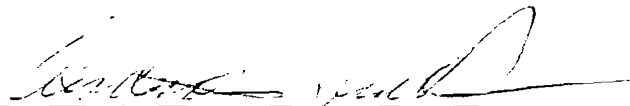
Nate Huang

Brown Graphics Group
Department of Computer Science
Brown University

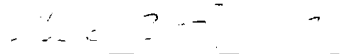
Submitted in partial fulfillment of the requirements
for the Degree of Master of Science in the
Brown University Department of Computer Science

Date printed: May 30, 1991

This research project by Nathan T. Huang is accepted in its present form
by the Department of Computer Science at Brown University
in partial fulfillment of the requirements for the Degree of Master of Science.



Professor Andries van Dam
Advisor



Date

Contents

1 Introduction 2

2 Previous Work 3

3 Project Goals 4
- 3.1 The addition of simulation 4
- 3.2 Integration of disparate control methods 5

4 Implementation 7
- 4.1 Review of the previous version of BAGS 7
- 4.2 Scopes 7
- 4.3 New data structures 8
  - 4.3.1 Chop Scopes 8
  - 4.3.2 State Scopes 8
- 4.4 Preserving Keyframing in `STATEtraverse()` 9
- 4.5 Adding Simulation 9
  - 4.5.1 `STATEtraverse_dynamic()` 9

5 Subtle Issues in Simulations 10
  - 5.0.2 Conserved Data 11
  - 5.0.3 The Dynamic Chop 11
  - 5.0.4 Partial SXF Forms 12
- 5.1 Using Multiple Scopes 13

6 Conclusions and Future Work 14

7 Acknowledgements 15

1 Introduction

Computer-generated animation embodies a wide variety of disparate motion-synthesis techniques. The earliest computer animation systems employed kinematic approaches, in which the animator explicitly specified a number of keyframes, and the computer calculated the in-between frames. Various methods were used to specify the interpolation methods for the smoothness of the in-betweening.

The quality of keyframed animations is directly proportional to the number of keyframes used. Thus, much of the burden of producing a quality animation is placed on the animator. In an effort to achieve increased realism and relieve the animator of this burden, procedural-based motion synthesis has gained popularity. These methods simplify motion specification at the expense of an increased computational cost. These methods, which include inverse kinematics, are popular in articulating human and other joint-based movement.

However, the greatest realism in motion for computer-generated animations is achieved through physical simulation. In a simulation, the user specifies a set of initial conditions and invokes the physical simulation. The simulation method is responsible for calculating the relevant data for the objects after each of a series of finite time steps. The calculations are based on a set of rules that will generate the realistic motion, for example, Newton's Laws. Although computationally expensive, physically-based simulations have produced the most realistic computer-generated animations to date.

The Brown Animation Generation System (BAGS) is a large-scale time-parameterized modeling and animation system. Previous versions of BAGS allowed the creation of animations through purely keyframed techniques. This project adds the ability to perform dynamic simulations to BAGS without compromising the previous keyframing functionality. This requires modifications to existing algorithms in BAGS, and the addition of new data structures to deal with the flow of control through time. In addition, this project provides a partial solution to the problem of integrating disparate control-method techniques.

The reader is assumed to be familiar with the Brown Animation Generation System, as described in [13] and [4].

2 Previous Work

The earliest computer animation systems were designed to compute the "in-between" frames from specified 2D keyframes [3]. 3D animation systems were originally logical extensions of the 2D keyframe systems, but higher levels of control were necessary for more complex models [10]. Parameterized models [8] [9] allowed a slightly higher level of control, as the values stored at each keyframe were parameters which controlled the positioning of the model, as opposed to being the positions themselves.

While parameterized keyframing is still a widely used technique, and will probably remain the paradigm of choice with many animators [7], researchers have sought higher-level motion specification techniques in order to simplify the animation development process and increase the realism of the resulting motion. Procedural methods can be used to specify the keyframes based on simple instructions rather than explicit positions. One such technique, inverse kinematics [2], simplifies the specification of keyframes.

Realistic, as in real life, motion requires that the objects obey the laws of physics. In recent years, a great deal of research in computer graphics has focused on techniques to produce motion based on the laws of physics [1] [6]. In a dynamic simulation, objects are given a set of initial conditions and a forward simulation is begun. At each time step, the objects' new positions (and/or shapes) are determined from their previous positions and their current physical properties, such as mass, force, and torque.

Dynamic simulations provide vastly more realistic motion, but they pay the price of being more computationally expensive. In addition, the simulated objects often require some form of control [11] [12], as they may be constrained to exhibit specific behaviors.

3 Project Goals

3.1 The addition of simulation

The primary goal of this project is to add simulation to BAGS. BAGS is a parametric keyframing system, but since all BAGS object attributes are inherently time-varying, BAGS objects are well-suited for use in dynamic simulations. In fact, it is possible to add simulation to BAGS without compromising the keyframing functionality.

As a keyframing system, BAGS can quickly obtain information from an object at any given time in a random-access fashion. That is, an inquiry for information from an object at time t will find the two keyframes whose times enclose time t, and interpolate a value for time t. The value at time t is not dependent on the value at any previous time, and can therefore be evaluated quickly.

In contrast, dynamic simulations require a past history to be maintained. A dynamic simulation begins with the specification of initial conditions. When an advance in time of dt is made, new values are calculated based on the previous values and the time step dt. Thus, if an inquiry for information is made at time t, it is necessary to simulate from time 0 to time t, one time step at a time, until time t is reached. The value at any time t is dependent on the value at the time $t - dt$.

As a simple example, consider the motion of a ball moving through space with a velocity $\vec{v}$. If the simulation's time step is 0.1, then the position of the ball at any time t is the position at time $t - 0.1$ plus the integral of $\vec{v}$ from time $t - 0.1$ to t. More complex dynamic simulations also take into account attributes like mass, force, torque, moment of inertia, and friction.

Although dynamic simulations provide a much greater degree of realism for motion, the advantages of the previous keyframing system should be preserved in a general animation system, since motion based on physical simulation requires intense floating-point computation and does not necessarily produce the *desired* motion. In some cases, parametric keyframing can provide an animator with the exact motion desired. For example, a clock's hour hand and minute hand motion can be exactly specified by through simple rotations - it is not necessary to preform a dynamic simulation to achieve this motion.

This project strives to provide the maximum flexibility to create:

1. Animations where all motion is keyframed.

2. Animations where all motion is calculated by simulations.
3. Hybrid animations where some objects' motion is under the control of simulations and other objects' motion is keyframed.
4. Hybrid animations where a single object's motion is specified by *both* simulation and keyframe specifications.

Note that (4) offers a form of control for a dynamic simulation. An object's initial conditions can be specified in a traditional keyframe fashion. Then, after each simulation step, additional keyframe specifications can be given to the object to adjust or constrain any attributes that the user desires.

3.2 Integration of disparate control methods

This project evolved from a series of discussions by a working group in the Brown Graphics Group addressing the problem of integrating disparate control methods. The aim of the discussions was to better understand the problem of integrating different animation paradigms into one large framework, and to make an initial attempt at solving the problem. A detailed summary is given in [5]. The main points relevant to this project are restated here.

In theory, a *controller* is an object that can read and modify the states of other objects. For example, given a set of joint and linkage objects, an inverse kinematics controller writes chops into the states of the joint and linkage objects, and thereby controls some or all of the behavior of those joint and linkage objects. As another example, a collision detection controller inquires the states of a set of objects and determines if there is any penetration between the polygonal representations of the objects. If there is, the collision detection controller can modify the states of those objects by writing in appropriate response chops, or it can simply relay the collision information to a collision response controller.

In practice, the controlled object makes a data inquiry to the controller object. The controller's data inquiry method supplies the requested data. For example, the object `ball` has its acceleration controlled at time 0 (and on) by the controller `gravity`. In a SCEFO script, this is specified in the following manner:

```
ball: acceleration 0 = gravity.acceleration;
```

If the controller is controlling multiple objects, the chop reference may be parameterized:

```
ball: force 0 = ncr.force(ball);
```

Here, **ncr** is a Newtonian collision response controller that might be used for many objects to calculate the appropriate responses for collisions between any two objects.

The working group found the problem of integrating disjoint controllers hard, probably unsolvable. The biggest difficulty arose from the desire to consider each of the multiple controllers as a *black box* - a separate entity that contains no knowledge of the mechanics or even the existence of the other controllers. The working group came up with a simple example in which it is impossible for two black box controllers controlling a single object to correctly solve a given problem. Some form of communication between the two controllers is necessary to determine the correct solution.

While this was a disappointing result, it is still possible to have multiple controllers affecting a single object. BAGS chops are prioritized, and state traversal is performed in strict priority ordering. For example, an object `ball` could be controlled by the controllers `gravity` and `wind`:

```
ball: acceleration 0 = gravity.acceleration
      force        0 = wind.force;
```

Due to chop priorities, the end effects of multiple controllers on a single object may depend on the priority ordering of the object's controller references. In the above example, gravity affects the ball before the wind does. Script priority ordering of the controllers offers the user more flexibility than BOLIO's strict ordering of kinematics before dynamics.

The working group concluded that an initial attempt should be made at integrating different motion controllers. The first step was to add the ability to perform simulation to BAGS.

4 Implementation

The actual implementation of a framework to provide BAGS with a technique to perform simulations involves extensive modifications to many low-level systems packages, including `STATE`, `CHOP`, and `SXF`. These modifications involve the creation of new data structures and algorithms, as well as modifications to existing algorithms.

4.1 Review of the previous version of BAGS

Before presenting the new data structures required to add simulation to BAGS, it is useful to review some of BAGS' previous mechanisms for maintaining chops and performing data inquiries.

In the previous version of BAGS, there was a single linked list of all the chops presently in the database, ordered by chop priority. When a chop was created, it was added into this chop list at the appropriate location based on its priority.

Each state maintained a single list of chops, also ordered by chop priority. Inquiries to the state were made at a given time and chop priority. The state's chop list was traversed up until the given priority, each chop being evaluated at the requested time, and an response to the inquiry was accumulated during this traversal.

4.2 Scopes

The notion of a *scope* is introduced in order to differentiate between the keyframing and simulation data evaluation paradigms. A scope is a group of chops that is either *static* or *dynamic.* A *static scope* is one in which data evaluation at a given time t is not dependent on any data at previous times. A *dynamic scope* is one where data evaluation at time t is based on data at a previous time $t - dt$.

A data inquiry into a static scope will cause a state traversal, as in the previous version of BAGS. Since the inquired data is not dependent on data at previous times, the data can be quickly evaluated and returned in a random-access fashion. A data inquiry into a dynamic scope, however, will require data from previous times. If these values have not been computed (i.e., this is the first inquiry into this dynamic scope), a simulation is performed up until the requested time.

4.3 New data structures

Two new data structures, the CHOPscope and the STATEscope, are introduced to provide a more general framework for chop management that will allow for both keyframing and simulation paradigms through the concept of scopes.

4.3.1 Chop Scopes

Recall that the goal is to not just to add simulation to BAGS, but to create a general mechanism that will also preserve the previous keyframing functionality. That is, a SCEFO script that previously worked in the purely keyframing system should also work in this modified system.

A new CHOP data structure, the CHOPscope, is introduced. A CHOPscope has the following fields:

1. a name for this CHOPscope
2. a pointer to a CHOPscope, used to maintain a linked list of all the CHOPscopes in the database
3. a linked list of chops in this CHOPscope
4. a field denoting the CHOPscope as either a *static* scope or *dynamic* scope

The previous version of BAGS maintained an ordered list of all chops in the database. This is now represented by an ordered list of CHOPscopes, where each CHOPscope maintains an ordered list of chops. A CHOPscope is created by the SCEFO statements *scope_static* and *scope_dynamic*. Any chops defined after one of these statements will be placed in the chop list of this CHOPscope, until another *scope_static* or *scope_dynamic* statement is encountered.

4.3.2 State Scopes

In addition to the CHOPscope data structure, a new STATE data structure, the STATEscope, is introduced. A STATEscope has the following fields:

1. a pointer to the CHOPscope associated with this STATEscope
2. a list of chops

3. the number of chops in the chop list

4. two pointers to STATEscopes in order to maintain a doubly-linked list of STATEscopes for a particular state.

In the previous version of BAGS, each state maintained a single ordered list of chops. Now, a state maintains an ordered list of STATEscopes, where each STATEscope contains an ordered list of chops. When a chop is added to a state, it is placed in the STATEscope associated with the chop's CHOPscope. A state only needs to maintain a STATEscope for a particular CHOPscope if it includes a chop from that CHOPscope.

4.4 Preserving Keyframing in STATEtraverse()

With the new modifications, all data inquiries are now made at a given time, STATEscope and chop priority. STATEtraverse() is used to step through each chop in a given STATEscope. In any given inquiry to the state, STATEtraverse() might be called recursively to inquire data from a STATEscope previous to the one the original inquiry was made at. For example, if a state has two scopes, *scope*1 and *scope*2, and an inquiry is made for information as of priority 2112 in *scope*2, a recursive inquiry for the same information as of the last priority in *scope*1 will be made (unless there is a valid cache of this data in *scope*2).

4.5 Adding Simulation

Up until this point, two new data structures and an algorithm modification have been introduced. These changes preserve the keyframing functionality of BAGS. This section presents a new traversal routine that allows BAGS to perform dynamic simulations.

4.5.1 STATEtraverse_dynamic()

An inquiry for data in a static scope is treated as in the previous BAGS. The data is inquired at a given time t, and the chops are evaluated at that time. STATEtraverse() is used for inquiries into static scopes. In pseudo-code, the routine is:

```
/* STATEtraverse()                                      */
/* Handle an inquiry at time t and chop priority pri */
find most recent cache;
```

```
for each chop from most recent cache until priority pri
   call the chop method, accumulating the response;
```

For dynamic scopes, an inquiry for data at time t requires the value of this (and possibly other) data at a previous time $t-dt$. To handle this, a new routine, `STATEtraverse_dynamic()`, is introduced. `STATEtraverse_dynamic()` contains a for-loop that performs the simulation's time-stepping - this loop is called the *simulation loop*. In pseudo-code, `STATEtraverse_dynamic()` is:

```
/* STATEtraverse_dynamic                                */
/* Handle an inquiry at time t and chop priority pri */
/* in a dynamic scope                                   */
find most recent cache;
lt = time of most recent cache in this dynamic scope;

/* The simulation loop*/
for sim_time = lt to t step dt

   /* inquire initial conditions */
   inquiry data at end of previous scope;

   /* traverse the chop list */
   for each chop from most recent cache until priority pri
      call the chop method and accumulate response;
```

Thus, if an object's position and velocity are controlled by a dynamic simulation method, then the object contains chops that are placed in a dynamic scope. When an inquiry to this object is made at this dynamic scope, `STATEtraverse_dynamic()` is called and a simulation is performed from the time of the last valid values (or time 0 is there are no previous values) up until the requested time.

5 Subtle Issues in Simulations

The above pseudo-code for `STATEtraverse_dynamic` is a simplified version of the actual implementation. There are a few subtle problems that must be dealt with - these are discussed in the following subsections.

5.0.2 Conserved Data

A dynamic simulation requires the retrieval of certain data at previous time steps. In a sense, this data must be *conserved* from one time step to the next – the value of the data at the end of the simulation loop iteration at time $t - dt$ must be the same as the value at the beginning of the next iteration at time t. Physical attributes such as position and velocity are examples of conserved data.

At the end of each iteration of the simulation loop, conserved data must be stored for use the next iteration. The BAGS caching mechanism [13] is used to store this data. At the end of each iteration of the simulation loop, any conserved data that has been accumulated is stored as a cache at time *sim_time* (a point time interval). Therefore, this data may be retrieved during the start of the next iteration.

One drawback of this technique is memory usage. For example, a simulation with 500 time steps will produce 500 caches each of position, velocity, angular velocity, and potentially other conserved data. This is highly wasteful, since each cache is typically used only once, during the iteration after it was created. A more efficient strategy is desirable, and of high priority for future work.

5.0.3 The Dynamic Chop

At each time step in a simulation, current values are derived from previous values, and often involve some form of computation. For example, a physical simulation that obeys Newton's Laws will derive an object's current position from the position at the previous time step and the current velocity. The current velocity is derived from the velocity at the previous time step and the current acceleration. By definition, these computations involve integration of velocities and accelerations with respect to time.

To perform this integration, a new *dynamic* chop is introduced. The dynamic chop is automatically inserted at the beginning of a dynamic scope. Its function is to inquire previous values of data (i.e., retrieve the values cached at the previous time step) and perform the integration over time to derive new values. Since the dynamic chop is placed at the beginning of a dynamic scope, it will always be handled before the rest of the chops in the scope during a state traversal. Thus, integration is done at the start of each new iteration of the simulation loop.

In this implementation, the dynamic chop handling method performs

a simple Euler integration technique in order to determine position from velocity, orientation from angular velocity, velocity from acceleration, and angular velocity from angular acceleration. The following set of equations summarizes this technique:

1. $\vec{x}_t = \vec{x}_{t-dt} + \vec{v}_t \times dt$
2. $R_t = R_{t-dt} + \vec{\omega}_t \times dt$ [1]
3. $\vec{v}_t = \vec{v}_{t-dt} + \vec{a}_t \times dt$
4. $\vec{\omega}_t = \vec{\omega}_{t-dt} + \vec{\alpha}_t \times dt$

where $\vec{x}$ is position, R is the orientation, $\vec{v}$ is velocity, $\vec{a}$ is acceleration, $\vec{\omega}$ is angular velocity, and $\vec{\alpha}$ is angular acceleration.

5.0.4 Partial SXF Forms

BAGS stores database information in a data structure called a `SXFform`. During a state traversal in a static block, a `SXFform` is passed along as each chop is visited, and the inquired data is accumulated into this `SXFform`.

However, when caching conserved data in the simulation loop, the entire form should not be cached. If the entire form, which includes the initial conditions, is cached and reinquired in each successive iteration, an undesired "multiplying" effect is achieved. That is, if the simulation loop runs for ten iterations, then the initial conditions will be applied ten times. If the initial position of an object is (5,0,0), then by the tenth iteration of the simulation loop, the object will be translated out to (50,0,0).

The problem is that the initial conditions should not be conserved. The conserved data should only represent the *partial product* of the accumulated data, that is, the portion of the data that has been obtained only by the simulation, not from the initial conditions. To accommodate this, a `SXFform` can now contain two parts, one for the initial conditions, and one for the partial product in a dynamic scope. At the end of each iteration of the simulation loop, the partial product portion of the form is the only part that is conserved and cached.

After the simulation loop is completed, the two parts of the `SXFform` are usually compressed into one complete `SXFform` which represents the answer to the inquiry. The function `SXFcompress_form` accomplishes this –

[1] Although this equation is not valid mathematically, the idea is what is important - rotations are based on angular velocities.

it contains the knowledge of how to merge the initial condition information and the partial product into one `SXFform` for all the types of conserved data.

5.1 Using Multiple Scopes

In practice, a dynamic simulation usually contains a static scope that sets up the initial conditions of the simulation, and a dynamic scope where the simulation is actually performed. In addition, a final static scope may be placed after the dynamic scope in order to specify kinematic adjustments to the objects in the simulation. The scope mechanism gives the user the flexibility to add kinematic user control as necessary.

The scope mechanism allows some objects to be under the control of a simulation and others to be purely keyframed in the same animation. This is due to the feature that an object performs a dynamic simulation if and only if it contains chops declared in a dynamic scope. Thus, it is simple and straightforward to create a scene in which the scenery (walls of a room, floor, and other "immovable" objects") does not get involved with the objects being controlled by a dynamic simulation. This can save considerable computation time.

6 Conclusions and Future Work

The Brown Animation Generation System was previously a purely keyframing system. This project has added the ability to perform dynamic simulations, while preserving the previous keyframing functionality. The new technique involves the creation of static and dynamic scopes, and a new state traversal routine to perform the simulation. The BAGS caching mechanism is utilized to carry data over from one time step to the next.

There is much opportunity to improve on the existing framework. One area for improvement is reducing the amount of cached data used to carry information over from one time step to the next. These caches are typically used only once, during the iteration after they were created. Thus, it would be useful to remove some or all of the caches immediately after they are used. Along the same lines is the notion of a *time 0 simulation*, in which the system behaves as a forward simulator running for an indefinite length of time. This would certainly require a better caching strategy.

This project represents an important step towards the development of a general animation system. Although simulation methods are becoming increasingly popular, keyframe techniques will remain an important paradigm in the creation of computer animations. By merging the two paradigms into one general framework, this project is the first step towards the creation of a unified graphics architecture in which multiple forms of motion control can be specified.

7 Acknowledgements

I would like to thank the members of the Brown Graphics Group for providing me with ideas, support, and friendship throughout this project. Very special thanks go to Bob Zeleznik for teaching me the intracacies of BAGS and eating chicken. Also thanks to W&SH, SD, PG, SB, VM, the CSAGB, AVD, and MC.

References

[1] Ronen Barzel and Al Barr, *A Modeling System Based on Dynamic Constraints,* SIGGRAPH '88 Conference Proceedings 22(4), August 1988, pp. 179-188

[2] Lisa K. Borden, *Articulated Objects in BAGS,* Master's Thesis, Brown University, May 1990

[3] Ed Catmull, *The Problems of Computer-assisted Animation,* SIGGRAPH '78 Conference Proceedings 12(3), August 1978, pp. 348-353

[4] Philip M. Hubbard, Matthias M. Wloka, Robert C. Zeleznick, Daniel G. Aliaga, and Nathan Huang, *UGA: A Unified Graphics Architecture,* Brown University Technical Report CS-91-30

[5] Philip M. Hubbard, *Notes on Controller Integration,* Brown University Graphics Group Internal Document

[6] Paul Isaacs and Michael Cohen, *Controlling Dynamic Simulation with Kinematic Constraints, Behavior Functions and Inverse Dynamics,* SIGGRAPH '87 Conference Proceedings 21(4), July 1987, pp. 215-224

[7] John Lasseter, *Principles of Traditional Animation Applied to 3D Computer Animation,* SIGGRAPH '90 Conference Proceedings 24(4), August 1990, pp. 35-44

[8] Scott Steketee, *Parametric Keyframe Interpolation Incorporating Kinetic Adjustment and Phrasing Control* SIGGRAPH '85 Conference Proceedings 19(3), July 1985, pp. 255-262

[9] David Sturman, *Interactive Keyframe Animation of 3D Articulated Models* Proceedings Graphics Interface '84, May 1984, pp. 35-40

[10] David Sturman, *A Discussion on the Development of Motion Control Systems,* In Computer Animation: 3D Motion Specification and Control, SIGGRAPH '87 Course Notes 10, pp. 3-15

[11] Michiel can de Panne, Eugene Fiume, and Zvonko Vranesic, *Reusable Motion Synthesis Using State-Space Controllers,* SIGGRAPH '90 Conference Proceedings 24(4), August 1990, pp. 225-234

[12] Andrew Witkin and William Welch, *Fast Animation and Control of Nonrigid Structures,* SIGGRAPH '90 Conference Proceedings 24(4), August 1990, pp. 243-252

[13] Robert C. Zeleznik, D. Brookshire Conner, Matthias M. Wloka, Daniel G. Aliaga, Nathan T. Huang, Philip M. Hubbard, Brian Knep, Henry Kaufman, John F. Hughes, and Andries van Dam, *An Object-Oriented Framework for the Integration of Interactive Animation Techniques,* SIGGRAPH '91 Conference Proceedings August 1991