Dynamics Modeling and Culling

Stephen Chenney, Jeffrey Ichnowski, and David Forsyth
University of California at Berkeley

Animating geometry procedurally, using a dynamical system (which consists of a set of state variables and a set of equations for describing how those variables change over time) rather than with keyframes, offers the advantages of physical realism, interactivity, compact descriptions, and infinite variety of motion. However, the use of dynamical systems has been inhibited by two factors: their computational cost, even when out of view, and the difficulties of implementation, partially due to a lack of standard modeling and runtime environments.

VRML is particularly prone to these problems. The most suitable, currently available, scripting language is Java. Controlling a VRML world from a Java object requires knowledge of a large number of methods and types, and is highly error prone. A lack of debugging environments, or even consistent exception handling, only exacerbates the problem.

Our research aims to develop modeling tools that enable incorporating large numbers of efficient dynamical systems into virtual environments, while abstracting the modeling process as much as possible. To achieve efficiency, we concentrate on *culling* dynamical systems: if the system is not in view, we do not compute any dynamics for it.

As a concrete example of the benefits of culling, consider a museum of kinetic sculpture (such as mobiles). Such a world has a large number of dynamic objects—the exhibits—placed in mutually nonvisible rooms. Traditional dynamic simulation requires computing the motion of all the sculptures on every frame, even if some sculptures aren't visible to the viewer. Naturally, this would severely limit the size of the museum. Culling avoids dynamic state computation for the exhibits not in view, just as geometry culling avoids rendering their geometry. Exploiting culling allows the museum to be arbitrarily large, provided only a few rooms are visible at once. Similar examples occur in factory simulations or dense city environments.

In this article we describe three tools that together provide an environment for authoring cullable, dynamic, rigid-body objects in VRML and Java:

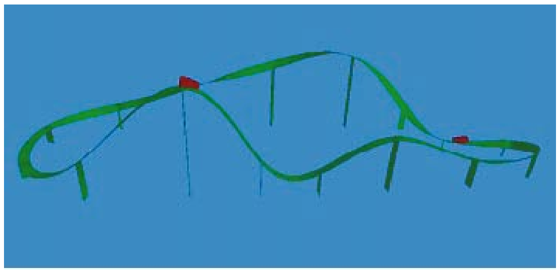
- A *code transformation tool* that exploits approximations to dynamical systems to enable culling. The frame rate for a world using these systems depends only on the number of systems in view. This results in significant speedups over conventional models and enables very large virtual worlds.
- A *runtime layer generator*, which defines a simple standard interface between a VRML browser and dynamical systems described in Java. This hides the complexities of the underlying interface from authors, while also allowing for a library of commonly used dynamical systems. Both reduce the difficulties of generating dynamic content.
- A *rigid-body modeler*, which allows users to interactively design the runtime layer and preview the dynamic behavior. The modeler environment is written in Java, so some degree of interactive debugging is possible.

The tools described here permit including large numbers of complex dynamic models in a VRML world easily and efficiently while maintaining high frame rates.

This article describes these tools, including some example systems, and discusses the runtime performance improvements obtained. Our tools are applicable if the spatial range of the dynamic model can be bounded by a static volume, the model is closed to outside influence, the underlying equations are continuous, and the dimension (number of degrees of freedom) of the system is small. Note that while this article focuses on VRML and Java as the target environment, the underlying techniques apply to any rendering and language environment.

Culling and prediction

In previous work[1] we described the use of approximations to cull moving objects. Those techniques employed statistical models and neural networks in hand-coded systems to enable the culling of dynamics. This allowed rapid generation of the new dynamic state



1 The roller coaster model.

even after long periods out of view. In this article we describe automated tools to perform the same task for systems whose range of motions can be bounded tightly. Where such bounds exist, the key problem—the *consistency* problem—is as follows. (Previous work[1] defines two other problems: completeness and causality. These problems are not relevant if the objects can be tightly bound, and we don't consider them in this article.)

> When a viewer turns away from a dynamic object, which is culled, and then turns back to the object, the new dynamic state should be consistent with a viewer's reasonable predictions for the behavior of the object during the time it was out of view. A solution to this problem requires the generation of a new dynamic state over very long time-steps (1) within an error bound determined by a viewer's ability to predict and (2) without significantly delaying rendering of the next frame.

It helps to consider the viewer as an adversary who is always seeking to discover a contradiction in the state as objects move in and out of view. It's the system's job to make sure that the viewer never spots such an inconsistency and never experiences significant lag.

The naive way to generate a consistent state is to determine exactly how the system behaved while out of view and display the resulting state. This satisfies (1) in the consistency problem above, but the resulting state may take a very long time to compute—arbitrarily long if an object can be out of view for arbitrary periods of time—and the method fails on point (2) above.

A better way of generating a new state when an object reenters the view comes from observing that viewers cannot accurately predict the behavior of a system over time.[2] If accurate predictions were possible, a viewer could know all of a world's future simply by observing it for a short period of time, which is obviously not the case. Solving consistency means satisfying viewer predictions, so if viewers cannot predict everything, we need not compute everything. All we need do is compute those things a viewer can predict. This is a fairly flexible requirement—we can do more or less work depending on the quality of the simulation required. For entertainment purposes, larger errors in the state may be acceptable, whereas visualization tasks or training simulations generally require higher accuracy.

A viewer's inability to predict results from several factors: uncertainty in the last known state of the system; uncertainty in the details of the model; and a lack of knowledge about factors that influence the system. Our methods are based primarily on uncertainty in the last known state of the system. In other words, when a system leaves the view, it's not possible for a viewer to have a completely accurate picture of, for instance, how fast it's moving, because of inaccuracies in the rendering (timing inconsistencies and pixel sampling errors) and limitations in the viewer's perception. By taking the uncertainty and propagating it forward through the time an object is out of view, it's possible to discover the uncertainty of an object's state when it reenters the view. If the new state we generate lies within this reentry region of uncertainty, a viewer cannot detect an inconsistency.

Our methods for generating a new state will rely heavily on the approximation of dynamical systems. Approximations have previously been exploited by several authors for a variety of purposes. Hodgins and Carlson[3] and Setas et al.[4] employ approximations for dynamical systems that are further from the viewer—level of detail for dynamic simulations. Grzeszczuk et al.[5] employ neural networks as approximation functions for dynamical systems, thus reducing the cost of generating the state for each frame, but ignoring issues such as level of detail and culling.

Generating approximations

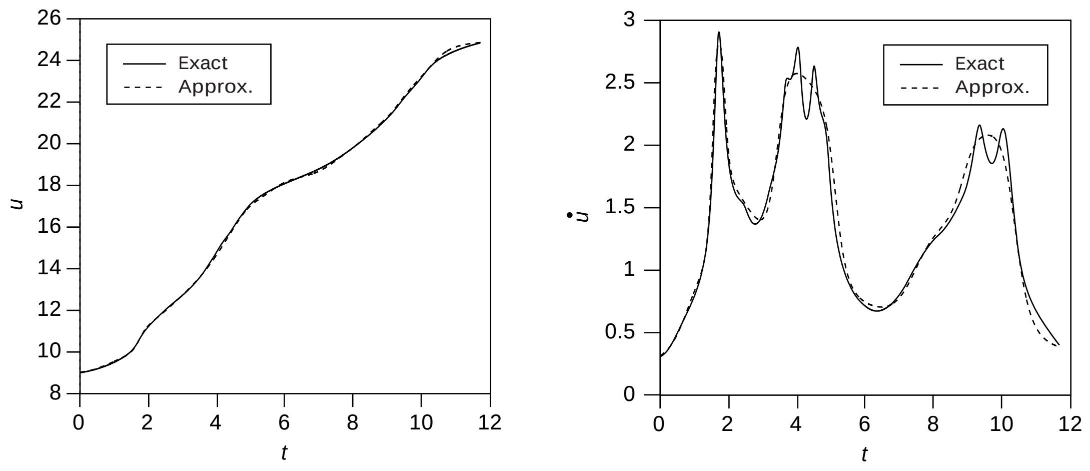
Our tools take as input a basic description of the dynamics and produce an alternate description suitable for culling. We assume the input description is the most accurate required for simulation and refer to it as the *accurate model*. We call the output description the *efficient model*. The tools we describe work for dynamical systems that are free from external influence and have a state space of low dimension. They prove useful only if the accurate model is expensive to evaluate over long time intervals. This occurs with, for instance, systems of differential equations evaluated by numerical integration or systems described by state machine transitions.

In our implementation, the accurate model consists of a Java class file that implements a function for evaluating the system at some given time when also given an initial state and time. The function should be able to generate the state in real time (it will be used when the system is in view), but need not be significantly faster. In specifying the accurate model, the user also defines other information helpful in analysis, such as initial conditions (which may be fixed or random), the dimension of the system, and a number of other parameters.

We distinguish systems according to whether they are periodic or not. A system is periodic if its state at time t, S_t, is identical to that at time $t + T$, where T is the period of the system. If $S_t = S_{t+T}$, then $S_t = S_{t+nT}$ for any integer n. If no such T exists, then the system is not periodic.

Periodic systems

As an illustrative example of a periodic system, consider a roller coaster model. The car runs on a track described by a uniform cubic B-spline (see Figure 1), under the influence of gravity but without friction. The car's "upright" direction is also described with a B-spline. Two state variables describe its motion: the parametric position on the track, u, and the derivative of u with respect to time, $\dot{u}$.



2 The exact functions for the parametric position *u* of a roller coaster car on its track and its derivative $\dot{u}$, and the neural network approximation to them, are plotted for one complete period.

Consider what a viewer might be able to predict about a system like the roller coaster. Over very short periods, on the order of a few seconds, the smooth nature of the motion makes accurate prediction easy: a viewer can simply extrapolate the last seen position and velocity. But if the roller coaster has been out of view for longer, the viewer must rely on what they know about roller coasters in general, such as how fast the cars move at a given position on the track. Predictions of this type will generally be uncertain: as long as the car appears to be doing about the right thing, the viewer will think everything looks reasonable. Our task is to find functions whose output is close enough to the true dynamics to fool the viewer, while still being fast enough to evaluate at runtime without introducing lag.

The behavior of a periodic system can be completely described by its state function over the course of one period: $S_t = \phi(t),\ t \in [0,T)$. If we build a closed-form approximation, $\hat{S}t=\hat{\phi}(t),\ t \in [0,T)$, then we can approximate the state of the system at any time, t', simply by evaluating $St' = \phi'(t' \bmod T)$. Provided the error in $\hat{\phi}$ with respect to ϕ is not large, a viewer will not detect the approximation error.

In our tools, approximation of periodic systems involves four steps:

- Determine the period.
- Bound the range of each state variable over one period, to aid in approximating.
- Learn a neural network approximation, $\hat{\phi}(t)$.
- Generate code for the efficient model.

Determining the period of a dynamical system, if one exists, is a common operation in numerical analysis. You may use Fourier analysis or an approach that searches for *return values*—places where the function takes on values it has taken before. Our tools take the latter approach.

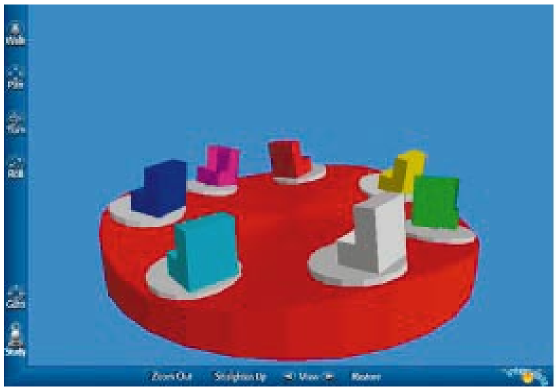
Given the period, it is possible to determine $\hat{\phi}$, the approximation function, which we represent as a neural network.[6] Neural networks have the advantage of near constant evaluation time for a given network and can approximate well the wide variety of functions our tools may encounter. Specifically, we use a standard feed-forward neural network with two hidden layers and a fixed number of nodes. The network has one input node, corresponding to the time at which we wish to evaluate, $t \in [0, T)$, and as many output nodes as there are state variables for the system. Our tools use networks with 10 hidden nodes per layer, which is a trade-off between the quality of the approximation and the time taken to evaluate the network. Other network topologies and training schemes could also be used.

The neural network will perform best if the function it is trying to represent has each component in the range (0, 1). Rescaling the state variables for a dynamical system to this range requires finding the minimum and maximum possible values for each component. You can do this by taking each variable, one at a time, and searching for global minimum and maximum values of that variable over the interval $[0, T)$.

To train the network, N samples of $\phi(t)$ are generated for random times within the period. We then repeatedly apply a standard back-propagation[6] with momentum algorithm on these samples. Each sample is used $N - 1$ times, after which we replace it with a new sample. We do this to reduce the number of dynamical system evaluations, since they're generally slower than a neural network training iteration. The learning process is terminated when the error falls below a user-defined threshold or a maximum number of iterations is reached.

Figure 2 shows the exact functions for the parametric position of a roller coaster car on its track, $u(t)$, and its derivative, $\dot{u}(t)$, and the neural network approximation to them, plotted for one complete period (12.13 seconds). The neural network was trained for about 30 minutes to achieve this result. The network approximates u with great accuracy, but does less well with the local maxima and minima of $\dot{u}$. That doesn't concern us in this example because the underlying dynamical system can correct any error based on energy constraints. Currently, our tools use the same network to approximate all the state variables for one system. A separate network could be used for each state variable, which would improve the error at the expense of additional code and evaluation time.

In the final step, code generation, we create a new Java class that evaluates the state of a periodic system



3 A Tilt-A-Whirl amusement park ride.

efficiently regardless of the time between evaluations. This new system can choose to use either the accurate model or the approximation. It applies the former if the interval between a viewer last seeing the system and the current time is less than 10 percent of the period; otherwise, it applies the latter.

Many approximation strategies other than neural networks are possible. In particular, if the system's period is short, state variables corresponding to a fixed set of times could be stored and a new state generated simply by interpolation. Such a scheme is directly supported by `Interpolator` nodes in the VRML language. However, in the general case, the neural network approximations we use are smoother and more compact.

Nonperiodic systems

As an example of a nonperiodic system, we use the Tilt-A-Whirl (see Figure 3), an amusement park ride that exhibits highly complex motion despite a simple dynamics description. The ride has seven cars, each attached to a platform on which it rotates freely. The platforms are driven around a circular hilly track. As the platforms move around the track, they tilt so as to remain tangential to the surface, which results in complex motions for each car.

Consider a Tilt-A-Whirl that moves out of view, then reenters. If it is hidden for only a short period of time, an observer can simply extrapolate from its state when it left their view, and hence quite accurately predict the new state. In this case we must use the most accurate model to update the Tilt-A-Whirl's state. However, as the Tilt-A-Whirl stays out of view for longer periods, an observer makes increasingly poorer predictions for what its new state should be. The system can make larger errors in generating the new state without contradicting the viewer. In other words, you may use approximations to the true system, and the approximation error can grow as the time interval out of view grows.

After a Tilt-A-Whirl has been out of sight for a long time, a viewer can no longer use information from the previous sighting to predict the new state. However, viewers can use their general knowledge of how a Tilt-A-Whirl behaves. To satisfy a viewer's prediction, we must choose a state that is typical for the Tilt-A-Whirl. To represent such typical states, we use a probability distribution over the state space of the Tilt-A-Whirl: states seen more often by a viewer will have a higher probability than states seen infrequently. To generate a new state, we simply sample according to the distribution. We refer to the distribution as the stationary distribution for the system. (Note, we borrowed the term "stationary distribution" from the theory of Markov processes, to which our distribution is related.[7])

While we phrased the preceding discussion in terms of a Tilt-A-Whirl's behavior, the observations made are typical for any nonperiodic system. Various system-dependent parameters will change, based on how easy or hard it is to predict the specific system, but the system can be analyzed to find values for these parameters. Our tools do exactly that.

Analysis begins with an accurate model, as for a periodic system, and proceeds through the following steps:

1. Find the range of the system—the bounds of its state variables in the state space. This lets us build cell structures over the space and scale values if required.
2. Build the stationary distribution that will be sampled to generate the new state when a system has been out of view for a long time.
3. Determine t_{long}, the time an object must be out of view before we can sample a new state from the distribution.
4. Build approximations for generating the new state when a system has been out of view for a medium period of time.
5. Determine t_{medium}, the time a system must be out of view before we use approximations instead of the accurate model.
6. Generate code incorporating the distribution for sampling, the approximations, and control logic for determining which method to use for a given time out of view.

Finding the range. The range of the system is important because it restricts the region of the state space we must concern ourselves with, allowing discrete cell structures to be built on the state space. To bound an individual variable, we search forward through time for local minima or maxima for each variable, updating the global minimum and maximum as we go. We stop looking for new local minima and maxima when the global values cease to change significantly.

This method is not foolproof—the simulation will not visit regions of the state space reachable from a different starting point. However, we can be arbitrarily certain of how good the bounds are by tracing a larger number of trajectories from appropriately distributed starting values. We find in practice that small errors in the bounds do not harm the analysis. Also, some variables may be bounded by the user in the input description, particularly angular variables (which lie in $(-\pi, \pi]$).

Building the stationary distribution. The stationary distribution indicates how much time a long-running system spends in any region of the state space. To model the distribution, divide the reachable regions of the state space into constant (user specified) size cells; a probability, P_i, is attached to each cell i. The result is a discrete distribution on cells, where P_i is the probability that, at a random point in time, the system lies within that cell. We assume that the distribution on points within a single cell is uniform.

To build the distribution, we begin with a large number of paths at random and integrate for fixed time-steps. We maintain a counter for each cell, measuring how many times a path lies in that cell at the end of a time-step. Then,

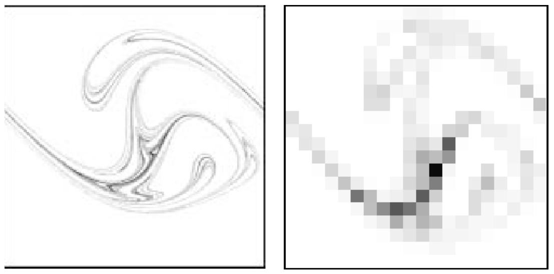
$$P_i = \frac{count_i}{\sum_i count_i}$$

According to the statistical law of large numbers, the P_i will converge to fixed values as the system is integrated for longer periods of time (assuming a stationary distribution exists). We monitor how much the distribution changes between time-steps and stop when the change becomes small as measured by the L_1 norm.

The discrete cell approximation to the exact stationary distribution performs well in practice, even with quite large cell sizes. Figure 4 shows the stationary distribution for the Tilt-A-Whirl, in which the discrete distribution succeeds in capturing the swirling nature of the exact distribution, but with only a small storage cost. The left image shows a high-resolution image of the distribution. Darker points correspond to higher probability, indicating that the system's state is more likely to take on that value. The right image shows the discrete cell approximation to the distribution. The discrete approximation still captures the overall character of the distribution, but with far smaller data storage requirements.

Determining t_{long}. The sampling threshold, t_{long}, represents the period of time that must elapse before a new state may be sampled from the stationary distribution, rather than computed based on some initial conditions. It equals the time taken for a small region of viewer uncertainty to evolve into the stationary distribution. To see why, consider what an observer knows when the object leaves the view. There is some error in this knowledge, which means that the system could be moving on one of several different paths. As time moves on, these paths diverge, until finally the distribution of possible paths looks like any other distribution of paths for the system—the stationary distribution. Because the two distributions are now the same, sampling from one is the same as sampling from the other, and observers cannot detect that we sampled from the stationary rather than the exact distribution defined by their knowledge.

To determine t_{long}, we sample a large number of starting values from within a small region of the state space, then integrate these paths for fixed time-steps (see Figure 5). At the end of each step, we check the difference between the distribution of the paths and the stationary distribution. If these are nearly the same, the total integration time is a candidate for t_{long}. We then repeat this procedure for other starting regions, until we have sampled from enough of the state space. The actual value used is the maximum t_{long} found for any region. Our methods directly examine the propagation of uncertainty, but other approaches based on more theoretical considerations[8] might also apply.

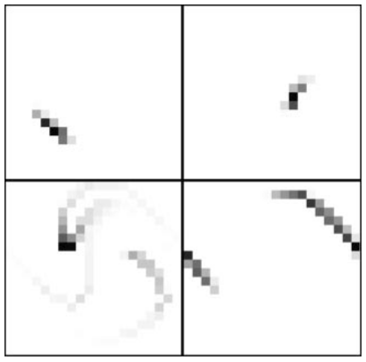

4 **The stationary distribution for the Tilt-A-Whirl system at high resolution (left) and a discrete cell approximation to the distribution (right).**

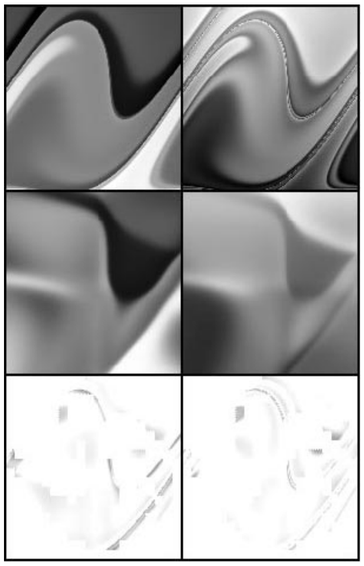
Building approximations. The approximation functions we build in this step will be used to generate a new state quickly, with some error allowed. After some short period of time, they must cost less to evaluate than the most accurate routine supplied by the user, and we want the cost of evaluating them to grow more slowly than the time period over which they evaluate. For this task we use neural networks, similar to those used for approximating periodic functions.

In this step we generate several neural networks, each of which evaluates over its own fixed time interval, Δt_i. As input, each network takes the state of the system at time t, so there are as many input nodes as state variables. On output, each produces the state at time $t + \Delta t_i$. One network evaluates a function over a period of half the sampling threshold, $t_{\text{long}}/2$. The next function evaluates over half this time, the next over half of that, and so on, stopping at a network that evaluates over either a user-defined minimum step or the period of any forcing functions, whichever is greater. By chaining neural networks together, we can evaluate to within a small distance of any time interval, which we will reach exactly using the accurate model.

Our structure of networks has the following advantages:

- We expect the cost of evaluating networks to grow at a slower rate than the time interval for which they evaluate, resulting in computational savings over the cost of evaluating one network over many steps.
- We can build the network's dependence on time into the network itself, rather than making it an input parameter. This significantly simplifies the network and allows lower errors for the same size net.
- We can tolerate larger errors in networks that evaluate over longer times, without sacrificing accuracy in networks that will evaluate over short periods.


5 **The convergence of one cell into the stationary distribution, for the Tilt-A-Whirl model. Starting top left and moving clockwise, the plots show the distribution of 5,000 paths after 3.07 seconds, 6.15 seconds, 9.23 seconds, and 12.31 seconds. The distribution in the lower left is sufficiently close to the stationary distribution to stop testing for this cell. Other cells take up to 24.6 seconds to converge.**


6 An example of the Tilt-A-Whirl equations of motion approximated by a neural network. Intensity indicates function value: lighter is higher valued.

- We can train networks concurrently, with significant improvements in training time and efficiency.

The samples on which we train the neural networks are distributed according to the stationary system (due to the method we use to generate them). This results in the networks having lower error rates in regions of high probability and higher error rates in regions of lower probability (see Figure 6). We judge this acceptable because the error will tend to be inversely proportional to the likelihood of a viewer seeing the error. We terminate learning if the error falls below a user-defined threshold value. We grow the network by adding five new nodes per layer each time its learning rate slows. We also force termination after a fixed number of cycles if the network has not reduced its error to an acceptable level.

In Figure 6, the network shown is attempting to learn the change in orientation and change in velocity of a Tilt-A-Whirl car over a 6.15-second interval. The left column plots the change in position as a function of initial conditions, and the right column plots the change in velocity. The top row shows the true function, the middle images show the function learned by the network, and the bottom row shows the difference image, masked by the stationary distribution shown in Figure 4. In each frame, the initial orientation increases from $-\pi$ to π along the horizontal axis, and the velocity increases vertically from –4.21 to 3.59. Note that we aren't concerned with errors masked out by the stationary distribution, because a viewer will never see these errors. Such regions include the top left and bottom left corners of each frame.

We could use other approximating functions, such as radial basis functions, wavelets, or splines, which have the advantage of elegant subdivision schemes, but lack the generality and ease of fitting offered by neural networks. For optimum approximation, the best method would be chosen based on how each performed on the target function.

Determining t_{medium}. To determine when we can use the approximation in place of the true evaluation routine, we may simply find the point at which it becomes more efficient to approximate, provided the approximation error is sufficiently low. The neural network learning procedure ensures that the error in the approximation function for the shortest evaluation time network is within a viewer's ability to predict. Since we also assume that neural networks are always cheaper to evaluate than the accurate system, we simply set t_{medium} to the smallest evaluation time of the networks we have trained.

Code generation. The code generated for nonperiodic functions allows efficient evaluation of new dynamic states over any time interval and within a viewer's ability to detect errors. The components of the new model are

- A representation of the stationary distribution and code to sample from it.
- Code to evaluate the various neural networks and a wrapper function that determines the set of evaluations required to step forward a given amount in time.
- Control logic that examines the difference between the desired evaluation time and the last time the viewer saw the system. If the difference exceeds t_{long}, it samples new values. Otherwise, if the difference exceeds t_{medium}, it uses a neural network approximation. Otherwise, it uses the accurate model to generate state.

The runtime layer

The tools for generating approximations for culling produce a description of the dynamical system that can efficiently evaluate the state at any given time regardless of the period between evaluations. The runtime layer aims to provide a standard interface between such systems, written in Java, and a VRML world. From the VRML side, a dynamical system is an `EXTERNPROTO` object, reducing to a minimum the complexity of incorporating a model into a larger world. From the Java side, it's only necessary to supply the dynamics evaluation routine in a standard format.

A runtime environment is generated for each dynamic object, consisting of a VRML file and a Java class file. Together, they must perform the following tasks:

- Store the geometry of the animated objects.
- Link the state variables of a dynamical system to transformations of the objects.
- When the VRML browser indicates a new frame (achieved through `TimeSensor` events, which we assume a browser sends at least once per frame), the runtime system must obtain dynamic state values for that frame.
- Track the visibility of the system and turn off evaluation of dynamic state (cull this system) if not in view.

The geometry of the animated object is stored within the VRML file, in standard VRML format. The VRML file also contains a `Script` node to interface to Java, and it defines all the necessary names and `ROUTES` for linking the script's output to transformations of the geometry.

To manage dynamic state, the runtime environment maintains buffers of state variables evaluated at fixed time intervals. Intermediate values required for rendering are interpolated between buffered states. We use linear interpolation, although higher order schemes are possible. Buffering state is advantageous because it provides a constant frame rate interface to the renderer, while allowing the underlying dynamic system to compute values at any rate and time-step. It also makes it possible to have useful values ready in the buffer if the object leaves the view and then reenters soon after.

When the runtime system receives a request for a new state, two possibilities exist, depending on whether the value is already buffered. If the value requested does not appear in the buffer, the dynamical system is evaluated twice, once for each of the values bracketing the requested time. If the values are already in the buffer, the system is still evaluated in order to fill future slots in the buffer. In practice, the system is evaluated repeatedly to fill the buffer until it signals that enough work has been done for one frame.

To track visibility, we use `VisibilitySensors` within the VRML file. These sensors send events each time a bounding volume enters or leaves the user's view. In turn, these events cause the Java class to mark individual dynamical systems that control the object as visible or invisible. Invisible objects will not be evaluated for a new state. The VRML specification does not allow a user to explicitly activate or deactivate Java scripts, so we use an automatically generated script to ensure that events are not sent to culled scripts.

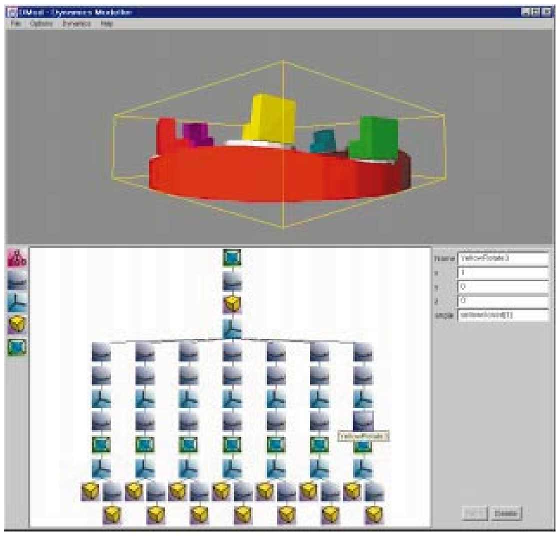
Each runtime environment is unique to its model. To create the runtime environment, the user provides a file describing the geometry to be animated and associating dynamic variables with transformations. A program then completes a template to create the runtime files. The file format used is simple enough to generate by hand, but the easiest method is through a modeler, which we describe next.

Attaching dynamics to geometry

The rigid-body modeler allows a user to load objects described by VRML's `PROTO` mechanism, display them, and build transformation hierarchies consisting of objects, rotations, translations, and bounding boxes. The values used for the transformations may be animated, thus creating dynamic models. As output, the modeling program produces the runtime layer, or it can save an intermediate file format.

The interface, which appears in Figure 7, consists of three regions. The top panel displays the current geometric arrangement of the system, including its bounding volume. The transformation hierarchy for the system appears in the lower left panel. A simple drag-and-drop interface is used to edit the hierarchy. The available nodes, shown on the far left, are (top to bottom): group, rotation, translation, VRML object, and bounding volume. By clicking on a node in the tree, a user can edit the fields of that node in the lower right panel, entering a constant or specifying a variable to animate the field. The user here is editing a rotation node, specifying a variable as the angle through which to rotate.

In describing systems for the modeling process, we distinguish between output variables and state variables.



7 The interface for our dynamic transformation modeler.

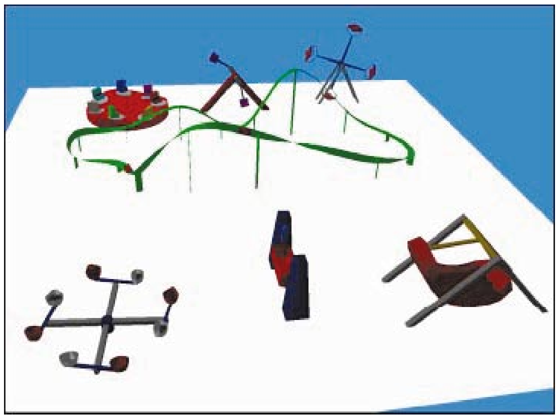
State variables are the set of values required to describe the system completely at any time, whereas output variables correspond directly to geometric transformations. The output variables are derived from the state variables through a user-defined function. For example, the roller coaster model has two state variables: the parametric position on the track and the parametric velocity along the track. The output variables are the actual position in world space and the various rotations to align the car with the track, derived by evaluating the track spline at the position indicated by the state variables.

Each field of geometric transformation is specified either as a constant value or as an element of a field variable. Each field variable is an array corresponding to the output variables for a dynamical system, which the user specifies as a Java class name when defining the field variable. The class for field variables must implement an interface that defines functions for evaluating the state at a given time, and for setting the values of output variables. The approximation tools described above produce such classes, but users can also define them directly—the format is simple and compact. Hand authoring is desirable for systems expressed as closed-form functions of time, which do not benefit from the approximations described above yet are still important for modeling dynamics.

Within the modeler, a user can preview the effect of the dynamical system on the model. Such a preview takes place within a Java environment, as opposed to the VRML browser environment. This makes debugging significantly easier.

The bounding boxes in the hierarchy tell the system which dynamics may be culled. Each bound has associated with it a set of variables whose visual effect is contained within the bound. If the bound is not visible, the dynamics for those variables will be culled. The modeler can automatically determine this bound by running the system within the modeler and examining the maximum extents of geometry over time.

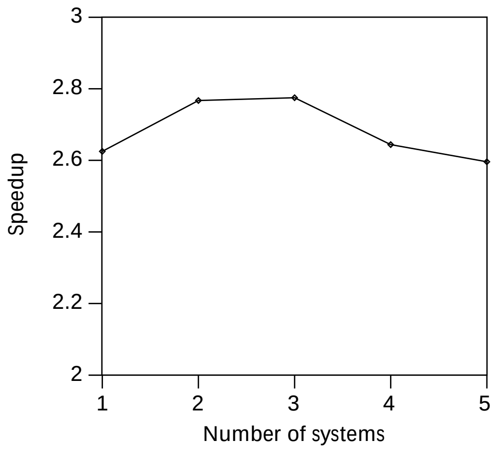
We emphasize again that the dynamics are largely independent of the geometry to which they are attached. Some dependence occurs if you want to make the simulation physically plausible, such as lengths of geometric



8 Fairground rides modeled with our tools: the octopus (front left), the pirate ship (front right), two versions of a double pendulum (center front and back), the dive-bomber (back right), and the roller coaster and Tilt-a-Whirl.

Table 1. Experimental timing values.

Total	Cull On (seconds)	Cull Off (seconds)	Number in View	Percent in View
7	0.048	0.126	1.42	20%
14	0.090	0.249	2.94	21%
21	0.142	0.394	4.75	23%
28	0.208	0.550	7.27	26%
35	0.280	0.727	9.12	26%



9 The speedups obtained plotted as a function of the total number of models in the world. Note that the vertical axis does not start at zero.

objects appearing as parameters in the dynamical system, but as parameters they are readily made available to a user and don't change the structure of the underlying equations. This allows reuse of parameterized dynamics with different geometries and vice versa. More importantly, it makes possible a library of dynamical systems, each with efficient cullable code, which authors could use in the same way they use 3D geometry libraries today.

Speedup results

To measure the performance improvement while culling dynamics, we studied the time spent computing the dynamics for each rendered frame of a simulation. The models used in our tests appear in Figure 8. We began with one instance of each in the world, then added additional objects up to a maximum of 35 (five of each example).

In each test, we measured the average frame time for a viewpoint animated such that the center of view moved in a circle around the world while the view direction oscillated through a 90-degree angle. We added rides so that the density of rides in the world remained approximately constant. With culling turned on, the dynamics for a ride were computed only if the ride was visible, and we used the models generated by our software to ensure fast, consistent evaluation. With culling off, the dynamics for all the models were computed for every frame using the accurate model for the system. The geometric rendering was not affected by the culling—we assume the browser culled geometry against the view volume.

Table 1 presents the timing values recorded in our experiments. It shows the average time per frame with and without culling, and the average number of models in view, for increasing numbers of models. The time per frame with culling on grows approximately linearly with the number of models in view, and the frame-rate speedup is roughly constant (see Figure 9), as expected with our setup. Our results demonstrate that the average time per frame is roughly linear with respect to the average number of systems in view, because we only perform computation for objects in view, and the browser performs geometric culling against the view volume.

Figure 9 plots the speedups obtained by culling over a world that does no culling. The speedups obtained are around 2.6 and remain roughly constant as the number of systems increases. We expect this result, as the percentage of systems in view remains approximately constant. We did not achieve the 4× speedup you might expect (only around 1/4 of the systems were in view at any time). This results in part because our underlying system must perform some computation to check whether each system is in view, as well as the overhead of Java scripts.

Future work

One significant extension we want to add to our approximation software is the capacity to handle state machines and hierarchies of systems. The optimization approach is similar, and the range of systems that can be modeled would greatly increase. For example, the Tilt-A-Whirl ride could stop to let off and collect passengers, rather than running indefinitely.

For novice users, an authoring tool would ideally hide any equations of motion for the system from the author. Such a tool would approach modeling from the point of view of geometric constraints and common forces (such as gravity or motors). The modeler would then infer the dynamics and generate the accurate model required by our current system. With this architecture the optimization process is carried out as a subsystem of the modeler, and the user need never explicitly state the dynamics.

Some aspects of our work could be simplified by incorporating appropriate semantics into the VRML specification. In particular, our runtime layer uses special scripts to ensure that timer events are only sent to systems in view. The VRML browser could be supplied with all the information necessary to do that itself: scripts require bounding volumes with the semantics that scripts need not receive events when their bound is invisible. In many cases this information could be inferred by examining the bounding volumes for geometry influenced by events from the script.

Of interest to the VRML community is the interaction of culling with multiuser networked environments. A conservative culling approach for multiuser worlds would keep track of the last time any viewer saw each system and apply the same tests used in this article to determine how the system should have evolved when next seen. More aggressive approaches might allow different viewers to see different things, based on the viewers' ability to communicate. In particular, as long as each viewer perceives the same events, their accounts will agree. If the actual dynamic motion that led to the perception differs just slightly, noticeable inconsistencies are unlikely.

Conclusion

The set of tools presented here enables authoring efficient dynamic models in VRML and Java. They use novel techniques for automatically generating dynamical models that may be culled when not in view. Our interactive modeler allows combining geometry and dynamics by associating geometric transforms with dynamic state variables.

Our results show that large numbers of models can be included in a VRML world without sacrificing frame rate. We achieve this by performing only that work necessary to ensure a consistent environment for the viewer. Our tools make it possible for those inexperienced with dynamics to achieve similar results, while strongly encouraging the development of a library of dynamics for use with varying geometry. ■

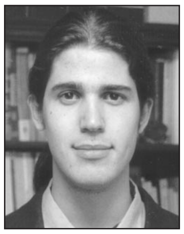
Acknowledgments

Our work was performed on machines donated by Intel and was funded by the Office of Naval Research grant ONR-MURI-N00014-96-1200. We used the Worldview for Netscape Navigator browser running under Windows NT on a 200-MHz Pentium Pro. We used Sced, a public domain constraint-based modeler, for the geometric modeling.

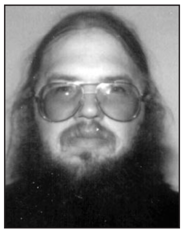
References

1. S. Chenney and D. Forsyth, "View-Dependent Culling of Dynamic Systems in Virtual Environments," *Proc. 1997 Symp. on Interactive 3D Graphics*, ACM Press, New York, April 1997, pp. 55-58.
2. S. Chenney, J. Ichnowski, and D. Forsyth, "Efficient Dynamics Modeling for VRML and Java," *VRML 98, Proc. 1998 Symp. on the Virtual Reality Modeling Language*, ACM Press, New York, Feb. 1998.
3. J.K. Hodgins and D.A. Carlson, Simulation Levels of Detail for Real-Time Animation," *Graphics Interface 97*, Canadian Information Processing Society, Toronto, Canada, 1997, pp. 1-8.
4. M.N. Setas, M.R. Gomes, and J.M. Rebordão, "Dynamic Simulation of Natural Environments in Virtual Reality," *SIVE 95: The First Workshop on Simulation and Interaction in Virtual Environments*, University of Iowa, Iowa City, July 1995, pp. 72-81.
5. R. Grzeszczuk, D. Terzopoulos, and G. Hinton, "Neuroanimator: Fast Neural Network Emulation and Control of Physics-based Models," *Computer Graphics*, (Proc. of Siggraph 98), ACM Press, New York, 1998, pp. 9-20.
6. C.M. Bishop, *Neural Networks for Pattern Recognition*, Oxford University Press, New York, 1995.
7. C.S. Hsu, *Cell-to-Cell Mapping: A Method of Global Analysis for Nonlinear Systems*, Springer-Verlag, New York, 1987.
8. J. Guckenheimer and P. Holmes, *Nonlinear Oscillations, Dynamical Systems, and Bifurcations of Vector Fields*, Springer-Verlag, New York, 1983.

Stephen Chenney *is a graduate student at the University of California at Berkeley. He received a BS in computer science from the University of Sydney, Australia, and an MS in computer science from UC Berkeley. His primary interests are physically based modeling and simulation, and the use of constraints in modeling.*



Jeffrey Ichnowski *is a chief product architect for Vita Systems, an Internet startup. He graduated with honors from the University of California at Berkeley in computer science and Asian studies. His areas of interest include haptic feedback devices, physical simulation, real-time networking algorithms, and compiler analysis and optimization.*



David Forsyth *is an associate professor of computer science at the University of California at Berkeley. He was educated at the University of the Witwatersrand, Johannesburg, and at Balliol College, Oxford. He is currently interested in computer vision and computer graphics.*

Readers may contact Chenney at UC Berkeley, EECS Computer Science Division, 387 Soda Hall, No 1776, Berkeley, CA, 94720-1776, e-mail schenney@cs.berkeley.edu. Contact Ichnowski by e-mail at jeffi@cs.berkeley.edu and Forsyth at daf@cs.berkeley.edu. Additional information, including an implementation of the tools described in this article, is available on the Web at http://www.cs.berkeley.edu/~schenney/dmc.