

Efficient Dynamics Modeling for VRML and Java

Stephen Chenney

Jeffrey Ichnowski

David Forsyth*

University of California at Berkeley

Abstract

Using dynamical systems rather than keyframing to animate a world is a desirable yet computationally expensive approach. We present techniques for culling dynamical systems that avoid unnecessary computation, and describe tools for automating much of the required work. Based on qualitative observations of how viewer's predict dynamical state over time, we identify methods for generating state while ensuring *consistency*, which we define as ensuring that a viewer's predictions are satisfied. Our tools take as input a description of a dynamical system, and produce an alternate description that may be efficiently culled. We also describe an interactive modeler in which authors attach dynamic variables to geometric transformations, allowing the modular re-use of dynamical systems. Together, our tools enable large numbers of complex dynamic models to be efficiently and easily included in a VRML world while maintaining high frame rates.

CR Descriptors: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism - *Virtual reality*; I.6.5 [Simulation and Modeling]: Model Development - *Modeling methodologies* I.6.8 [Simulation and Modeling]: Types of Simulation - *Animation*

1 Introduction

To date the majority of animation in VRML worlds has been achieved through key-framing, which can offer only a finite number of motion sequences in any world. The alternative, which allows for constantly changing and adaptable motion, is to express the motion of objects through a set of state variables and descriptions of how these variables change over time - a *dynamical system*. Such a model offers significant advantages: motion need only be designed once for arbitrary animation times, and the representation is compact. This allows long running, interesting environments to be authored and described in a format that is small enough to download over modem speed lines.

Dynamical systems have been extensively used for generating animation [1][5][9][13] yet in real-time environments the cost of dynamics can easily dominate the cost of rendering. This situation is likely to worsen with the use of PC hardware rendering and advanced visibility schemes. VRML

is particularly prone to this effect - while a browser can improve rendering performance, it has no control over the cost of evaluating animation scripts. Hence, to improve the efficiency of dynamics, optimization must be part of the authoring phase. Incorporating dynamics into the authoring process also abstracts the dynamic simulation above the level of scripts, routes and naming required for the VRML file.

If we are to place large numbers of complex dynamic models in an environment, then we should compute state only for objects in view, and cull dynamics for objects that are out of view. This is the traditional approach in graphics: compute only what is important to the current view. Knowledge of what to cull is provided in VRML through the `VisibilitySensor` mechanism. In current implementations this culls to the view volume, but future implementations may include occlusion culling and other advanced techniques. If a world is designed as multiple files which are loaded in memory only as required, then it must be possible to cull dynamics, because the script will not even be present in memory if the subsection of the world it animates is not visible.

In a previous paper [4] we showed how to build complex dynamical models that can be culled using any visibility scheme. That paper demonstrated significant and scalable speedups, allowing worlds of arbitrary size to be built without sacrificing dynamic content or frame rate. Sudarsky and Gotsman [12] demonstrate mechanisms for incorporating dynamic models into a BSP tree visibility scheme. They assume models that are closed-form functions of time, both to generate space-time bounding volumes and to generate new state quickly. This excludes most interesting models. Papers by Carlson and Hodgins [3] and Setas et. al. [11] have shown how to achieve speedups by applying level of detail techniques to a dynamic game environment and a forest environment respectively.

For all the previous results, the models and approximations used were hand generated for a specific environment. This paper describes methods that automatically generate models for culling in a standard environment. We present two tools that work together to generate efficient dynamic models for VRML worlds. These tools take a basic description of the dynamical system, modify it to support efficient culling, then allow a user to associate dynamic state with geometric transformations - thereby specifying what moves and how. The result is a VRML prototype and Java class structure which is suitable for culling. The process, apart from the interactive modeling, is completely automatic, avoiding the need for an author to write highly complex and error-prone code to implement the dynamics efficiently. The final output of the process may be included seamlessly in any VRML environment.

We now provide an overview of issues involved with culling dynamics. We then discuss our dynamics optimization approach; we give a detailed description of its components, including our dynamic transformation modeler, and conclude with example systems and comments on future work.

*[schenney|jeffi|daf]@cs.berkeley.edu

1.1 Culling Dynamics

Culling dynamics means computing dynamic state only for systems that influence the current view. This leads to three problems:

- **Consistency:** if the view turns away from a system of moving objects whose dynamics are then culled, in what state should those objects be when the view turns back? In an ideal system, an observer should not be able to obtain contradictions by looking away from an object, and then back at it. For example, consider a damped pendulum. When a viewer turns back to the pendulum, it should have less energy than the last time it was sighted, and it should satisfy some phase relationship.
- **Completeness:** objects that are out of the view volume often travel into it of their own accord - for example, consider a view into a room full of balls bouncing off walls. If the dynamics of a ball are culled as soon as it leaves the view volume, a fixed view could contain a steadily decreasing number of balls. This is probably not what the simulation author intended. Avoiding this difficulty requires some way of telling where an object might be, without necessarily solving its equations of motion.
- **Causality:** relationships often exist between the states of distinct objects, and these relationships must be maintained, even if one or other object is culled. Lights and their switches are one example.

In practice, the extent to which these problems manifest themselves is highly dependent on the system being culled. The ability to solve these problems depends both on the systems involved and the requirements of the simulation. If a system is intended to give very accurate measurements about the behaviour of a system, most culling may be unacceptable. If a simulation is intended to produce visually convincing renderings, then these problems are significantly easier to solve - particularly if the viewer is not actively seeking errors in the simulation. The majority of VRML environments fall into the latter category, so we can expect to achieve significant savings through culling.

This paper is concerned with solving only the consistency problem. For systems that don't move very far, and whose behaviour is largely independent of other objects, this is the only problem that arises. We use amusement park rides as examples, but kinetic sculptures or factory machines are equally applicable.

The methods presented will ensure that when a system moves back into view its state is consistent with what a viewer previously knew about the system. The naive way to generate consistent state is to determine exactly how the system behaved while it was out of view, and display the resulting state. This will work for systems expressed as closed form functions of time (simply evaluate at the new time), but for systems that are evaluated by summing incremental changes over time, 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. This introduces substantial lag, and is a poor solution.

2 Generating Consistent State

A better way of generating new state when an object re-enters the view comes from observing that viewers cannot accurately predict the behaviour of a system over time. If accurate predictions were possible, a viewer could know all

the future of a world 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.

A viewer's inability to predict is the result of several factors: a viewer's uncertainty in the last known state of the system; a viewer's uncertainty in the details of the model; and a viewer's 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 is 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 a viewer's perception. By taking the uncertainty, and propagating it forward through the time an object is out of view, it is possible to find out how uncertain an object's state is when it re-enters the view. Provided the new state we generate is within this re-entry region of uncertainty, a viewer cannot detect an inconsistency.

Based on viewer prediction we qualitatively identify three regimes of viewer prediction, characterized by short, medium and long time periods out of view.

2.1 Short periods out of view

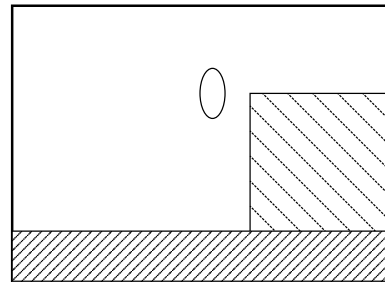


Figure 1: An example of short term prediction: A falling rock may be accurately predicted until it hits the ground.

Over short periods of time, viewers can make accurate predictions of smooth dynamics. Consider the case of a falling rock, where the viewer sees the rock go over the edge (figure 1). While the rock is in flight, but before it hits the ground, a viewer can make quite accurate predictions about its state. Uncertainty doesn't grow very much over short periods of time for smooth dynamics.

To ensure consistency in this situation we must use the most accurate model of the system available, saving no computation. To avoid lag, we attempt to run the dynamic simulation ahead of the rendering and buffer values (figure 2). When an object goes out of view, we stop filling the buffer, but if it re-enters the view soon after, we have a value ready in the buffer without incurring lag.

A dynamical system is free to buffer values at any fixed time-step, and by interpolating between these values the rendering can run at a different, varying time-step. This is important in VRML because the frame rate is not constant within a given simulation nor across the same simulation on different platforms, yet we would like to do a constant amount of work on the dynamics between each rendered frame.

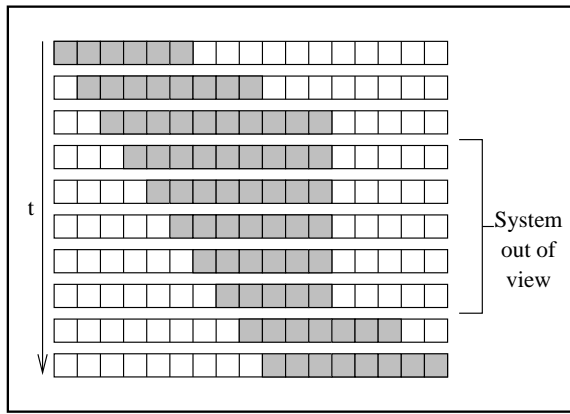


Figure 2: We buffer dynamic state ahead of the rendering, as indicated by the shaded regions. When the object is in view, we fill the buffer faster than the renderer consumes values - three times faster in this case. When the system is not in view, no new values are computed, but the old values become redundant as time passes. If the object re-enters the view before the buffer is empty, as is the case here, then there is no lag. We also interpolate between values in the buffer to make the frame rate independent of the simulation time steps.

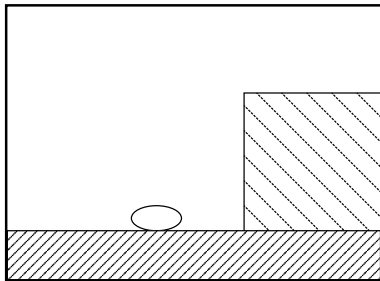


Figure 3: An example of medium term prediction: Once the falling rock hits the ground, its position and orientation are uncertain. However, a viewer does know for certain that the rock is on the ground at the bottom of the hill.

2.2 Medium periods out of view

Over the medium term, viewers may still be able to predict some things accurately, but for most parameters they can predict little. Following the rock example (figure 3), once the rock hits the ground a viewer can no longer say accurately where it is, nor how it is oriented. More complex motion may have several intermediate, medium term regimes, reflecting a viewer's decreasing ability to predict.

To provide a consistent state in this situation we are free to use approximations to the system. The approximation must generate a new state for the system given an old state and the time out of view. The errors in the approximation should reflect a viewer's uncertainty. Conversely, because a viewer is uncertain, they cannot detect reasonable approximation errors. Many methods exist for approximation; we use neural networks.

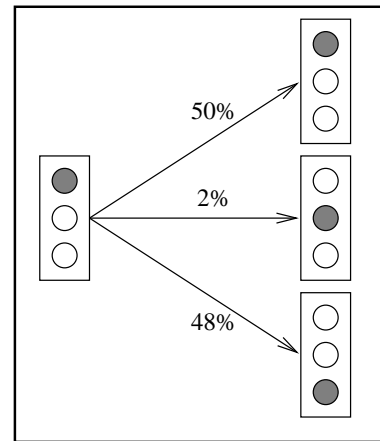


Figure 4: A traffic light is an example of long term prediction. If the light has been out of view for more than a few minutes, a viewer cannot say exactly which state it is in. However, a viewer does expect the light to be red or green more often than it is yellow.

2.3 Long periods out of view

In the long term, a viewer can no longer use information from a previous sighting to predict a new state, nor are they completely ignorant of the system's behaviour. Traffic lights provide a good demonstration (figure 4). A viewer's prediction reflects the general behaviour of a traffic light, which is to be red more often than green, with yellow least likely. To exploit this, we sample a new state from some statistical distribution over states. The sample is independent of any previous state, but the distribution reflects the long term behaviour of the system - generally referred to as the *stationary distribution*. If a fully computed system was sampled many times at random intervals, these samples would be distributed according to the stationary distribution. We can use standard density estimation techniques from various sources to model these distributions [2][8].

The use of a stationary distribution reflects that eventually a small region of uncertainty in a viewer's knowledge of state may grow over time to match the stationary distribution. We base some of our techniques for analyzing systems on that observation.

2.4 Overview of the Modeling Process

To build a dynamic model using our system, a user provides a description of the basic dynamics and the geometry to be animated. As output, our system generates a VRML file containing the geometric transformations and event routing, and a set of Java classes implementing the dynamics. The process we use consists of two phases (figure 5):

- The *dynamics optimization* process takes the dynamics code provided, analyses it, and transforms it into a new Java class which implements the same dynamics but in a way that can be culled efficiently.
- The *dynamic transformation modeler* is an interactive system in which a user describes the transformation hierarchy and identifies which dynamic state variables apply to which geometric transformations.

All our software is implemented in Java, to take advantage of platform independence and other language features. However, the underlying techniques are language and rendering

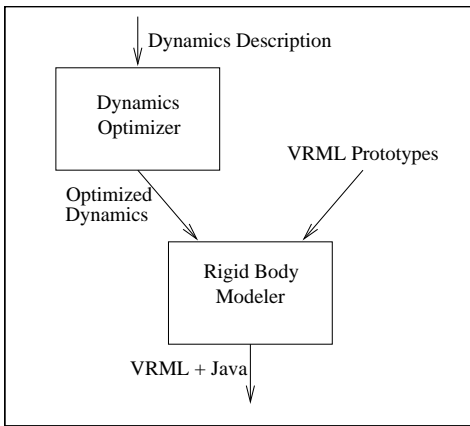


Figure 5: *The process for building dynamic models. Our tools take a Java class describing the system dynamics and analyze it to produce a more detailed description of the system, but one that may be culled efficiently and consistently. This is combined with the results of an interactive modeling process, by which an author has identified which variables in the dynamical system map to which values in a set of VRML transformations. The final result is a VRML prototype describing the geometry, transformations and routes, and a set of Java script files for evaluating the state of the system. Note the loose correspondence to an optimizing compiler. We take a system description in one format, optimize it to improve performance, and produce another system as output for our target machine - the VRML browser.*

system independent. We now describe each sub-system in detail.

3 Optimizing Dynamics for Culling

The optimization software takes as input the *accurate model* - a description of the dynamics which we assume to be the most accurate required for simulation. As output, we produce the *efficient model* - an expanded description of the system now suitable for efficient culling. Our software is useful only if the accurate model is expensive to evaluate over long time intervals. This is the case with, for instance, systems of differential equations evaluated by numerical integration, or systems described by state machine transitions. As a further restriction, our software currently assumes a continuous, low dimension state space.

The sequence of steps in analyzing a system is:

1. **Find the range** of the system. This allows us to build cell structures over the space and to scale values if required.
2. **Build the stationary distribution** that will be sampled to generate 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.
4. **Build approximations** for generating 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 stationary distribution for sampling, the approximations, buffers for short term evaluation and control logic for determining which method to use for a given time out of view.

3.1 The Accurate Model

Our software takes as input an accurate model, consisting of a single Java class file which may instantiate other classes. The class must implement the `DynamicsSystem` interface, which defines functions for querying important features of the dynamics (such as the state space and initial conditions) and a function for evaluating the system using the accurate model. This function must be able to determine the final state given an initial state and time, and the final time. It is an abstraction of the system to a simple mapping function from one state to another over a specified, variable time interval. The function must as a minimum be able to generate state in real time (it will be used when the system is in view). However, for best results with the buffering techniques we use, the system should be able to generate state at twice the rate the renderer consumes it.

3.2 Finding the Range

The range of the system is important because it restricts the region of state space we must concern ourselves with, and it allows us to re-scale each of the state variables to the range $[0, 1]$. The latter is necessary because the neural network approximations we use that map $[0, 1]$ inputs to $[0, 1]$ outputs. Some variables may be bound by the author in the input description, particularly angular variables (which are bounded by $(-\pi, \pi]$).

For each variable not bound by the author, we run a simulation for a large number of time steps and maintain the maximum and minimum defining the range. This method is not foolproof - the simulation will not visit regions of state space that are 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 affect the quality of our optimizations.

3.3 Building the Stationary Distribution

The stationary distribution is the distribution indicating how much time a long running system spends in any region of the state space. To model the distribution, the reachable regions of state space are divided into constant size cells, and 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 is within that cell. Given that a viewer cannot distinguish differences in state smaller than the size of a cell, we assume that the distribution on points within a single cell is uniform. This approach is similar to that of cell-to-cell mapping [6] as a means of analyzing dynamical systems, but we seek to discover different types of system behaviour.

To build the distribution, we begin with a large number of paths at random starting points and integrate them for a short period of time to eliminate startup transients (*burn in* the paths). We continue to integrate for fixed time-steps, maintaining a counter for each cell measuring how many times a path is in that cell at the end of a time-step. Then,

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

According to the weak 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 falls below a small threshold. As a measure of the difference between two distributions, we use the L_1 norm,

$$dist(x, y) = \sum_{i=1}^n |P_i^x - P_i^y|$$

where n is the number of cells and P_i^x is the probability of cell i in distribution x . We represent cells using a probability tree data structure [8]. To reduce the overall size of the final data structure, we recursively combine neighboring cells with similar probability densities.

3.4 Determining t_{long}

The sampling threshold, t_{long} , is 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 is equivalent to the time taken for a small region of viewer uncertainty to evolve into the stationary distribution. To see why this is the case, consider what a viewer 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 do different things, 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 a viewer can't detect that we sampled from the stationary rather than the exact distribution defined by their knowledge.

To determine t_{long} , we sort the cells of the stationary distribution according to P_i , then take the N most probable cells such that $\sum_{i=1}^N P_i < \text{important}$, where *important* is a constant in $(0, 1]$. For each of these cells, we take a large number of paths with starting points uniformly distributed in the cell, then integrate these paths for fixed time-steps. At each time-step, we look at the instantaneous distribution of paths. If the difference between this distribution and the stationary is less than some threshold, we stop and record the simulation time as t_i . Finally, we take t_{long} such that it is greater than 80% of the t_i values. This effectively ignores a few cells with very long t_i , which reduces the maximum time over which we must approximate the system.

There are several parameters implicit in this approach:

- The cell size. The size of the cells used as initial regions of uncertainty encodes how well a viewer can measure the state of an object leaving the view. Adjusting this value is a trade-off between the quality of the simulation and the efficiency gains achieved by culling. Large cells assume larger errors. However, larger cells will also give smaller thresholds, which is desirable.
- The number of paths to trace from each cell. Larger values are likely to give a more accurate measure t_{long} , but also increase the analysis time.
- The difference between the instantaneous distribution and the stationary distribution. We allow quite large distances, because viewers cannot accurately compare the distributions in the course of a running simulation.
- *important*, which determines how many cells will be traced. Using a value of 1 would trace every cell, but it is unlikely that every cell is equally important, because

the system is more likely to leave the view in some cells than in others. With *important* less than 1, some cells will not be tested, but these will be cells that the system rarely visits, and hence the viewer rarely sees.

We allow these parameters to be set as part of the description of each system.

3.5 Building Approximations

The approximation functions we build in this step will be used to generate new state quickly, with some error allowed. After a short period of time, they must be cheaper to evaluate than the most accurate routine supplied by the user, and we want the cost of evaluating them to grow slower than the time period over which they are evaluating. For this task we use neural networks. They have the advantage of near constant evaluation time for a given network, and they represent well the highly non-linear functions we wish to model. Specifically, we use a standard feed-forward neural network with two hidden layers and a variable number of nodes per hidden layer.

As our approximation, we use a hierarchy of neural nets. One net evaluates a function over a period of half the sampling threshold, $t_{long}/2$. The next function evaluates over half this time, the next over half of that and so on until we reach some minimum time step (specified as part of the system description). By chaining neural nets together, we can evaluate over any period to within some small amount, which we will clean up using the accurate model. This structure of networks has the following advantages:

- We expect the cost of evaluate networks to grow at a slower rate than the time interval they evaluate over, 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. This ties in with our model for viewer prediction - the increasing errors in the networks correspond to a viewer's greater inability to predict.
- We can learn networks concurrently, with significant improvements in training time and efficiency.

To train our networks, we take a set of paths and integrate them through time. Let t_{min} be the shortest time period over which any network must evaluate the system. We store values at multiples of time t_{min} , up to some maximum number of values, N . Within this set of values, there are $N - 1$ intervals of length t_{min} , $N - 2$ intervals of length $2t_{min}$, $N - 4$ intervals of length $4t_{min}$ and so on (figure 6). For every interval of a given length, we store the scaled starting value and change in value for that interval as a training example for the appropriate network. After randomizing the order, we train the network using the examples. We evaluate the error and if necessary repeat the whole process, starting with further integration of the paths through time (we never reuse training data).

This process will generate training examples that are distributed in state space according to the stationary distribution. In other words, we are implicitly using a form of importance sampling to generate our training samples. This is a desirable outcome, because we prefer our network to do better on values that are frequently seen by a viewer, and

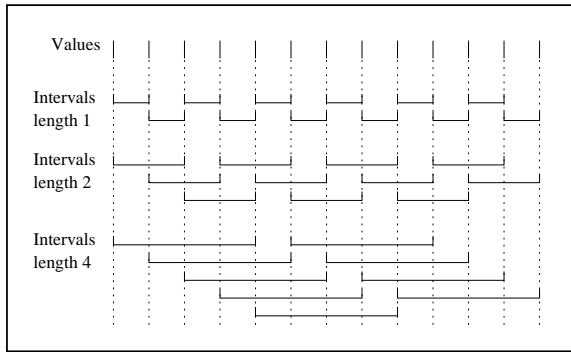


Figure 6: In order to generate training patterns for the neural networks, we generate a sequence of values at fixed intervals. From this sequence, we can extract intervals of length 1, 2, 4 and so on. By generating training patterns in this way we get the maximum amount of training data out of each evaluation of the accurate model.

are not interested in how well it approximates state values that a viewer never sees.

We terminate learning if the error falls below a threshold value. We grow the network by adding new nodes each time its learning rate slows, up to a maximum size. We also force termination after a fixed number of cycles if the network has not reduced its error to an acceptable level, and allow a user to stop the training at any time.

We could use other approximating functions, such as radial basis functions, wavelets or splines. These methods have the advantage of elegant subdivision schemes, but in general lack the ability to compactly represent highly non-linear functions. In future versions of our tools, it is likely that a range of approximators will be tried, with the best one chosen dependent on the dynamical system.

3.6 Determining t_{medium}

The approximation threshold, t_{medium} , is the time at which two conditions are satisfied:

- It is more efficient to evaluate the approximation function than the accurate model.
- The error in the approximation function is within a viewer's ability to predict.

The neural network learning procedure ensures that the latter condition is met for each network. We simply assume the first condition is met by the first network, and use its time step as t_{medium} . To be more accurate, we could look at each network in turn, starting with the cheapest, and compare its expected evaluation time with that for the accurate model - taking the first network that was faster to evaluate.

3.7 Code Generation

As output, the optimizer produces an extension of the accurate model class that incorporates additional evaluation routines and control logic for choosing which routine to use. The components of the new model are:

- A buffer for state variables and code for interpolating between values within it.
- A representation of the stationary distribution and code to sample from it.
- Code to evaluate the series of neural networks for time intervals in the range $[t_{medium}, t_{long}]$.

- An evaluation routine that takes the desired evaluation time as input, and sets state variables as output.

The evaluation routine examines the difference between the desired time and the last time the system was evaluated. If the difference is greater than t_{long} it samples new values and returns. Otherwise, if the difference is greater than t_{medium} , it uses a neural network approximation and returns. Otherwise, it tries filling the buffer using the accurate model, ensuring that there are at least values up to the desired time, and maybe beyond if there is enough time left to compute extra values.

4 Adding Dynamics to Geometry

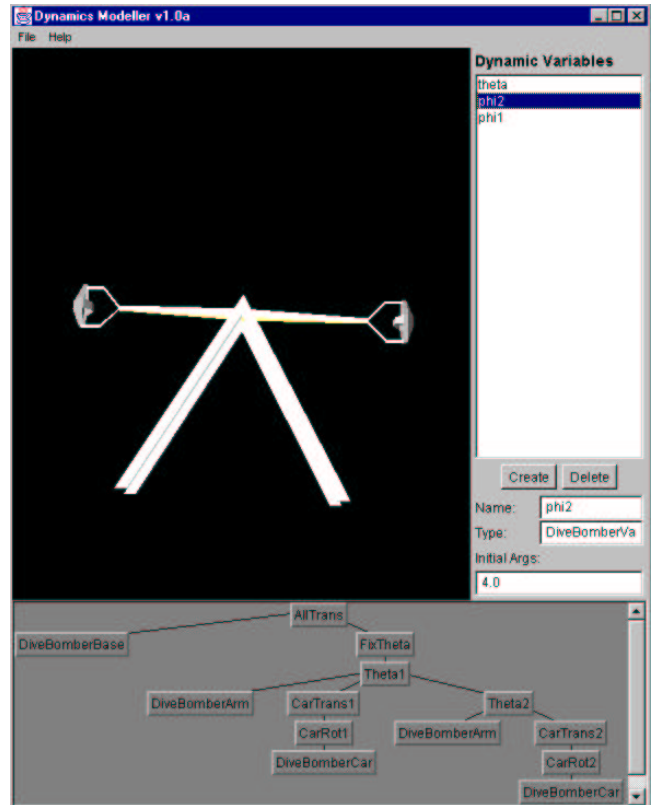


Figure 7: The interface for our dynamic transformation modeler consists of three regions. In the top left the current geometric arrangement of the system is displayed. Top right is a list of all the dynamics variables defined for this system, which allows the user to edit the Java class that evaluates that variable, plus its initial value. The transformation hierarchy for the system is displayed across the bottom. By clicking on a node in the tree, a user can edit the fields of that transformation, entering a constant or a variable to animate the field.

The dynamic transformation modeler allows a user to load objects described by the PROTO mechanism, display them, and build transformation hierarchies consisting of objects, rotations, translations and bounding boxes. The interface is shown in figure 7.

The fields of transformations are specified either as constant values, or as references to variables. Each variable is in turn associated with a Java class to evaluate that variable. Each variable is an array of floating point values, and

the mapping between variables and fields specifies not only the variable but also the element within it. It is assumed that each variable will correspond to a complete, independent dynamical system, and that each element of the array is a state variable for the system.

The bounding boxes in the hierarchy are used to 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.

As output, the modeling program produces a file describing the variables used and the transformation hierarchy. We have a filter program that takes that description and generates a VRML prototype node implementing the complete object, along with a script file in Java to manage the dynamics. The prototype node includes `VisibilitySensor` nodes to direct culling, and all the event structure for setting transformations from within the script. The prototype defines `eventIns` for receiving timer events. The Java script produced instantiates all the variables when it initializes, and then manages the visibility and evaluation of dynamics.

The modeler expects any variables to be of a class that implements `DynamicsVariable`, an interface that defines functions for evaluating the state at a given time, and for returning the value of a state variable. Our dynamics optimization code produces classes of this form, but system authors are also free to define classes directly. This is desirable for systems expressed as closed-form functions of time, which do not benefit from optimization yet are still important for modeling dynamics.

We also emphasize that the dynamics are largely independent of the geometry they are attached to. There is some dependence if the simulation is to be physically plausible, such as lengths of geometric 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 re-use 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 could be used by authors in the same way 3d geometry libraries are used today.

5 Example Systems

We used our tools to produce efficient dynamic models of four amusement park rides. We describe one, the Tilt-A-Whirl, in detail and present summary information for the other three.

5.1 The Tilt-A-Whirl

The Tilt-A-Whirl (figure 8) is 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 is free to rotate. The platforms are driven around a circular hilly track according to known functions of time. 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.

The state variables of this system are elapsed time, t , the position of the car on the platform, ϕ and its first time derivative, $\dot{\phi}$. The fixed parameters to the system are: r_1 , the radius of the track; r_2 , the distance from the center of the platform to the car's center of mass; α_1 , α_0 , the size of the hills; θ_0 and $\dot{\theta}_0$, the platform's initial angular position and velocity around the track; and ρ , a damping constant.

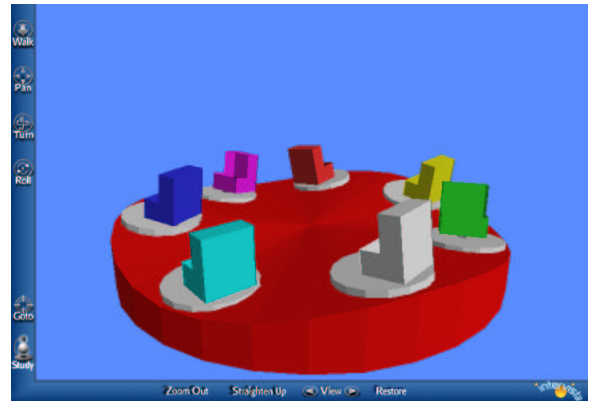


Figure 8: A Tilt-A-Whirl amusement park ride.

The governing equation of motion for the car, derived from Lagrange's equation and making use of small angle approximations (following [7]) is:

$$r_2^2 \ddot{\phi} + \rho \dot{\phi} - gr_2(\alpha \sin \phi - \beta \cos \phi) + r_1 r_2 \dot{\theta}^2 \sin \phi = 0$$

$$\begin{aligned} \theta &= \theta_0 + \dot{\theta}_0 t \\ \alpha &= \alpha_0 - \alpha_1 \cos 3\theta \\ \beta &= 3\alpha_1 \sin 3\theta \end{aligned}$$

In modeling this system, we use one `DynamicsVariable` class for θ , α and β . These values are all evaluated by closed form functions of time, so there is no point in performing the culling analysis on them. We use another system for ϕ and $\dot{\phi}$, which is coded as a `DynamicsSystem`, and which we analyze for culling. ϕ is an angular variable, so it is bounded by $(-\pi, \pi]$. The evaluation code uses a numerical integration routine (taken from [10]) to generate new state.

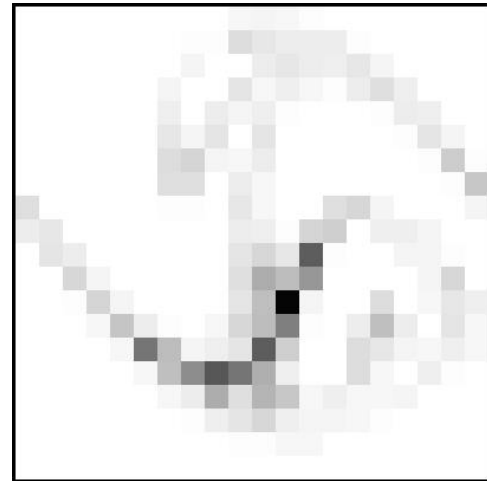


Figure 9: The stationary distribution for the Tilt-A-Whirl system. The intensity of a cell indicates its probability. Darker cells indicate that the system's state is more likely to be in that cell. Note that many cells are completely white, indicating that the system never enters a state within that cell. If we were to use a sampled state from within such a cell, the viewer might detect an inconsistency in the ride's behaviour.

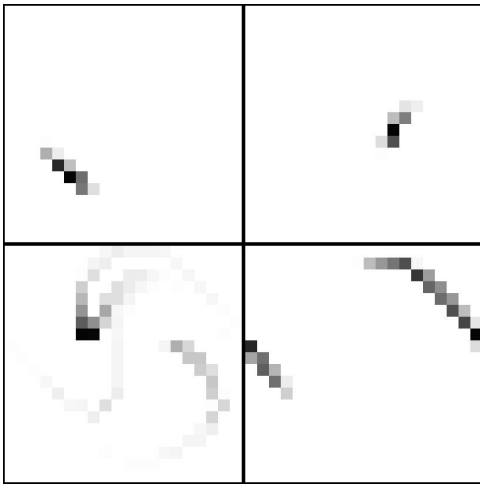


Figure 10: *The convergence of one cell into the stationary distribution. Starting top left and moving clockwise, the plots show the distribution of 5000 paths after 3.07 seconds, 6.15 seconds, 9.23 seconds and 12.31 seconds. The distribution in the lower right is sufficiently close to the stationary distribution to stop testing for this cell. Other cells take up to 24.6 seconds to converge.*

After writing the initial evaluation code, we run the culling analysis software on the system. The bounds for ϕ are found to be $[-4.21, 3.59]$. The stationary distribution is found (figure 9) along with a sampling threshold of roughly 18.46 seconds. The convergence of a small set of initial paths to the stationary distribution is shown in figure 10. In this analysis, the cell size was approximately $\pi/15.0$ by 0.3, $s_{important}$ was 0.8 and the maximum difference between distributions used to measure convergence was 0.8.

The software learns three neural networks, one for each of approximately three, six and twelve seconds. The mapping learned by the six second network and the errors it makes are shown in figure 11. Training the neural network is the most costly part of the optimization process. The complete analysis process, from bounding the variables to generating code, took approximately two hours.

The resulting `DynamicsVariable` class is several hundred lines long. This code is ready for use with the interactive modeling program. Producing the same code by hand would not only take a significant amount of time, but would also be very prone to errors.

Independent of the dynamics modeling, the various geometric components were modeled. Any geometric modeler that produces VRML prototype nodes could be used.

To complete the process, we use the dynamic transformation modeler to describe how the transformations are controlled by the dynamics variables. In this case there are nine objects connected by 43 transformations. There are 26 variables to animate the transformations. One variable animates the rotation of all the cars around the platform. There are three variables for each car, one for the orientation of the car on its platform and two for the orientation of the platform on the track. Bounding boxes are defined for all of the variables involved. The resulting description file is transformed into VRML and a Java script file.

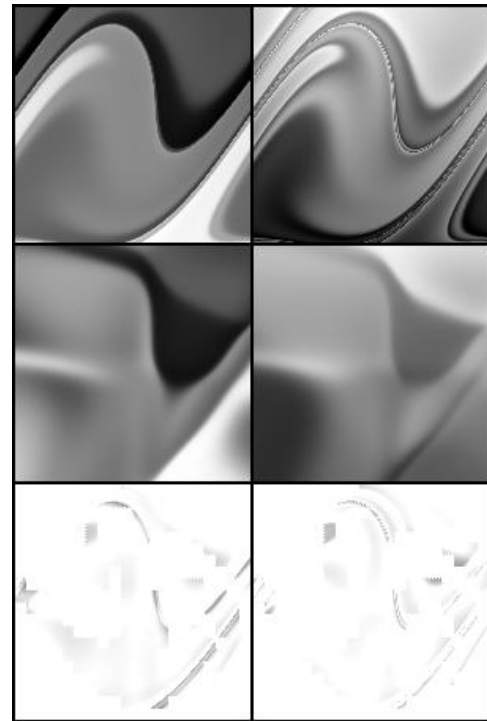


Figure 11: *An example of the function learned by a neural network. The network shown is attempting to learn the change in orientation and change in velocity of a Tilt-A-Whirl car over a 6.15 second interval. The left column plots the change in position as a function of initial conditions, and the right 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 9. 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 are not concerned with errors that are masked out by the stationary distribution, because these errors will never be seen by a viewer. Such regions include the top left and bottom left corners of each frame.*

5.2 Other Models

We modeled three other rides with our system:

- The octopus consists of eight cars, each free to rotate about a vertical axis at the end of a rotating beam. The beam is attached to a central frame, which itself rotates.
- The dive-bomber consists of four cars held in cradles at the end of a rotating frame. Each car is free to rotate about an axis perpendicular to the frame's axis.
- The pendulum is based on a forced double pendulum. It produces violent motion, too dangerous for anything but a virtual ride.

The key parameters inferred by the analysis software, with some information about the systems, are shown in table 1.

Name	Octopus	DiveBomber	Pendulum
Vars	10	5	3
Dim	2	2	2
t_{medium}	15.0s	8.0s	1.92s
t_{long}	30.0s	16.0s	23.1s
Nets	1	1	4
Time	11h53m	10h47m	6h46m

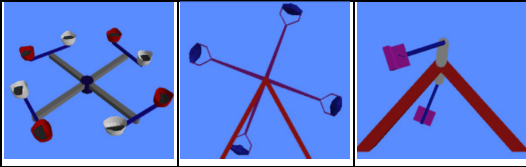


Table 1: Various statistics for three fun park rides. Vars is the total number of animated variables in the system. Dim is the dimension of the dynamical system that is optimized. It is lower than vars because each ride consists of several independent systems, and some variables are modeled as closed-form functions of time. t_{medium} and t_{long} are the thresholds for approximation and sampling respectively, and nets is the number of networks trained. Time is the total time taken for the optimization process. Note that a longer t_{medium} , which corresponds to a longer forcing function period, implies longer running times, because we integrate in steps of one period.

6 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. We used all our example models as test subjects, beginning with one instance of each in the world, then adding additional objects up to a maximum of 20 (five of each example). We used a single point for each of the geometric components of the rides, as an approximation to ignoring the cost of rendering. In each test, the average frame time was measured 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° angle. Rides were added so that the density of rides in the world was 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 were computed for every frame using the accurate model for the system.

Table 2 presents the timing values recorded in our experiments. Figure 12 plots the speedups obtained by culling. The speedups obtained are between 1.5 and 2.0, and increase as the number of systems increases. This is mostly due to the fact that, as the time per frame increases, the time per frame without culling increases superlinearly: each system must integrate over a longer period per frame, which takes more time.

We expect the speedups to be roughly proportional to the ratio of the number of systems in view to the number of systems out of view. In our experiments, this ratio is near constant because we keep the density of systems constant and the ratio of volume in view to volume out of view the same. We hence see near constant speedups. Using more advanced visibility schemes, not currently offered in VRML, we would expect the number of systems in view to be con-

Total	Culling	No Culling	In View	% in View
4	0.018s	0.028s	1.0	25%
8	0.034s	0.057s	2.0	25%
12	0.053s	0.092s	3.2	26%
16	0.068s	0.130s	4.0	25%
20	0.088s	0.176s	4.9	25%

Table 2: Average time per frame with and without culling, and the number of systems in view, for increasing numbers of systems. The time per frame with culling on grows approximately linearly with the number of systems in view, as we would expect from our set up. The time per frame with culling turned off does not scale linearly with the total number of systems. This is because, in addition to there being more systems, each system must evaluate over a longer period of time for each frame.

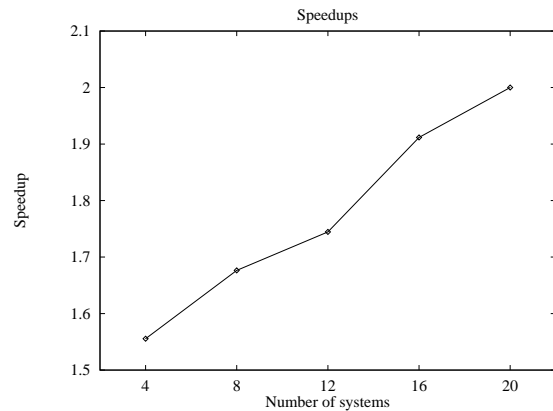


Figure 12: The speedups obtained plotted as a function of the number of systems in the world. The speedup is not constant, as we would expect in our experimental set up. This is mostly a result of the time per frame with culling turned off not scaling linearly with the number of systems. Note that the vertical axis does not start at zero.

stant, while the number out of view increased. In this case, the speedups would improve roughly linearly in the number of systems added to the world.

7 Conclusions and Future Work

We have presented a set of tools for authoring efficient dynamic models in VRML and Java. They make use of novel techniques for automatically generating dynamical models that may be culled when not in view. With these models the dynamics computation time may be halved on average using only view volume culling to determine visibility. Our interactive modeler allows geometry and dynamics to be combined by associating geometric transforms with dynamic state variables. This supports the modular re-use of both geometry and dynamics.

One significant extension to our optimization software is to add 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 be greatly increased. For example, the Tilt-A-Whirl ride would be able to 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 dynamics need never be explicitly stated by the user.

In incorporating our models into VRML environments we have pushed the boundaries of VRML's timing model and script evaluation specifications. When doing dynamic simulation, one would like to know the time the frame currently being computed will be displayed, so that the state shown is accurate for that time. In VRML this is extremely difficult, largely because the simulation does not run to a fixed frame rate. Furthermore, in our experience browsers do not send timer events once per frame, and we must write special purpose filters to distribute timer events that are guaranteed to be in chronological order and strictly increasing. Secondly, we do not know for certain that timer events will not be sent to scripts controlling systems out of view. To work around this, we include scripts to manage the visibility of top level objects and forward timer events only to visible systems.

Also of interest to the VRML community is the interaction of culling with multi-user networked environments. The methods for generating alternate dynamic models described here are independent of the determination of which model to use in a given environment. That issue depends on how viewer prediction is modeled in the environment. A conservative culling approach for multi-user worlds would keep track of the last time each system was seen by any viewer, and apply the same tests used in this paper to determine how the system should be evolved when it is 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 percept is slightly different, that is unlikely to lead to noticeable inconsistencies.

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 work that is 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.

8 Acknowledgments

Our work was performed on machines donated by Intel, and 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 200MHz Pentium Pro. Our Java code for optimizing the system was compiled using the just-in-time compiler for the JDK v1.1.4. We used Sced, a public domain constraint-based modeler, for the geometric modeling.

References

- [1] David Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *Computer Graphics*, volume 23(3), pages 233–232. ACM SIGGRAPH, July 1989. Boston, Massachusetts.
- [2] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.
- [3] Deborah A. Carlson and Jessica K. Hodgins. Simulation levels of detail for real-time animation. In *Graphics Interface '97*, pages 1–8, 1997. Kelowna, BC, Canada, 21-23 May 1997.
- [4] Stephen Chenney and David Forsyth. View-dependent culling of dynamic systems in virtual environments. In *Proceedings 1997 Symposium on Interactive 3D Graphics*, pages 55–58, April 1997. Providence, RI, April 27-30.
- [5] Jessica K Hodgins, Wayne L Wooten, David C Brogan, and James F O'Brien. Animating human figures. In *Computer Graphics: Proceedings of SIGGRAPH 95*, pages 71–78, August 1995. Los Angeles, CA.
- [6] Chieh Su Hsu. *Cell-to-cell mapping: a method of global analysis for nonlinear systems*. Springer-Verlag, New York, 1987.
- [7] R. L. Kautz and Bret M. Huggard. Chaos at the amusement park: Dynamics of a tilt-a-whirl. *American Journal of Physics*, 62(1):59–66, January 1994.
- [8] Michael D. McCool and Peter K. Harwood. Probability trees. In *Graphics Interface '97*, pages 37–46, 1997.
- [9] Brian Mirtich. *Impulse-based Dynamics for Rigid-Body Simulation*. PhD thesis, University of California, Berkeley, 1996.
- [10] William H. Press, Saul T. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, England, 2nd edition, 1992.
- [11] M. N. Setas, M. R. Gomes, and J. M. Rebordão. Dynamic simulation of natural environments in virtual reality. In *SIVE95: The First Workshop on Simulation and Interaction in Virtual Environments*, July 1995. University of Iowa, Iowa City, IA.
- [12] Oded Sudarsky and Craig Gotsman. Output-sensitive visibility algorithms for dynamic scenes with applications to virtual reality. In *EUROGRAPHICS '96*, volume 15(3), 1996.
- [13] Xiaoyuan Tu and Demetri Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. In *Computer Graphics: Proceedings of SIGGRAPH 94*, pages 43–50, 1994.