Controllable and Scalable Simulation for Animation

by

Stephen John Chenney

B.Sc. (University of Sydney) 1995
M.S. (University of California, Berkeley) 1997

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor David A. Forsyth, Chair
Professor John F. Canny
Professor David J. Aldous

Spring 2000

The dissertation of Stephen John Chenney is approved:

Chair	Date
	Date
	Date

University of California at Berkeley

Spring 2000

Controllable and Scalable Simulation for Animation

Copyright 2000
by
Stephen John Chenney

Abstract

Controllable and Scalable Simulation for Animation

by

Stephen John Chenney
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor David A. Forsyth, Chair

Simulation is an important means of generating animations. For example, we might use simulation to animate a virtual city in order to train emergency response personnel. Such an application requires the responsiveness and realism that simulation offers, and it also requires the ability to stage specific events in a very large animated environment. Traditional simulation technology fails on the latter count: it is difficult to direct a given simulation toward a desired outcome, and existing simulations rarely scale well to large virtual worlds.

This thesis addresses controllable and scalable simulation for the purposes of computer animation. We describe a technique for directing the outcome of simulations by formulating the problem as one of probabilistic sampling. A Markov chain Monte Carlo (MCMC) algorithm is used to perform the sampling, which allows the generation of multiple animations from a desired distribution. Furthermore, if the distribution assigns probabilities according to the plausibility of an animation, then we can be certain that most of the sampled animations will appear reasonable to a viewer. A range of examples are presented from the domain of collision intensive rigid-body simulation, the majority of which could not be produced using previous technology. We also describe a new rigid-body simulation algorithm that was developed for this work.

Scalable simulation is achieved through simulation culling, a method for focusing the computational effort on visible parts of a simulation. Aspects of the simulation that are not in view are not explicitly computed, thus saving large amounts of work. Approximations and random models are used to ensure that objects that leave the view re-enter when necessary in a plausible state, even though their full motion was not computed while out of view. A virtual fairground and a large virtual city are presented as case studies. These examples raise a number of open problems, which we discuss in some detail.

This thesis treats control and scale as largely independent problems, yet we show that both may be viewed as sampling problems. We conclude with a look at how direction and culling might be integrated to enable large-scale virtual environments for realistic training and entertainment.

Professor David A. Forsyth
Dissertation Committee Chair

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

Many people have contributed to the development of this thesis. David Forsyth, my advisor, has been an invaluable source of knowledge and ideas. I particularly appreciate his flexibility in allowing me to pursue my own research ideas.

Several fellow students have assisted in the development of the systems described here. Jeff Ichnowski implemented much of the dynamics culling software, and Jimmy Ho helped out with modeling and documentation. Okan Arikan continues to bring great expertise to various aspects of the city model described in chapter 6, and Ryan Farrell also contributed.

I thank the many people who have reviewed this work in various forms, particularly Ronen Barzel, John Hughes, Joe Marks and Brian Mirtich for their feedback and discussion over the years.

I would also like to thank the faculty, students and staff at Berkeley who have provided assistance and friendship, particularly members of the graphics and computer vision groups.

This work was funded by a BMDO and ONR MURI grant, number N00014-96-11200. Members of the review panel have provided great feedback over the course of the grant.

Chapter 1

Introduction

Animation, the process of generating a sequence of moving images, is central to many applications of computer graphics. Feature films and television use animation to tell a story. Training environments, such as those used by pilots, require real-time, highly realistic animation, intended to create a sense of presence in the virtual world. Entertainment applications, such as computer games, use animation for both storytelling and presence. Visualization applications use animation to aid in the understanding of data, be it time varying or not.

As a specific scenario, imagine the task of creating a virtual city, and using it to train ambulance drivers to perform their job safely and effectively. Such an environment must have at least the following properties:

- A viewer should see the same quantity and quality of motion in the virtual world that they would experience in a real city.
- The environment should respond to the user, just as the inhabitants of a real city respond.
- It should be possible to direct specific events, such as a car running a red light, repeatedly across many training runs, each time consistent with the real behavior of motorists, yet each time different in some way.

An environment with the above properties would be realistic enough to ensure that a user's experiences in the virtual world carry over to the real world, as is desired for a training environment. Over multiple sessions, the viewer would also be trained to the general aspects of their job, rather than any one specific instance (as would be the case if there was only ever one "red light runner" animation for the environment). The requirements might be less stringent for an entertainment environment, because there is no need for the experience to match reality, yet some realism is required to establish and maintain a viewer's sense of presence in the virtual world.

Simulation, the generation of animation from a procedural description, offers several advantages for animating a virtual city. Simulation is realistic if based on suitable models, responsive when the motion is generated in real-time, and space efficient because a relatively compact description of the world can be used to generate animation of arbitrarily long duration.

The computer graphics literature contains techniques for dynamically simulating a wide range of phenomena, including humans [14, 43], creatures [57, 59, 86], fluids [27, 28, 48, 70, 79, 92], rigid and deformable bodies [4, 62, 85, 97] and recently fractures [71] and explosions [68]. In principle, anything that can be described by a solvable system of equations or a set of rules can be simulated. In practice, however, two major technological problems prevent us from building a virtual city: the inability to direct the outcome of many types of simulation, and poor scaling properties as large numbers of moving objects are placed in one environment. The body of this thesis addresses both concerns.

1.1 Simulation

Modeling may be viewed as the task of producing estimates for the properties of a process. Some tasks, such as climate modeling, make predictions of future behavior, so the aim is to produce estimates that match the correct, as yet unknown, outcome of the system. For instance, global climate models attempt to predict the distribution of temperature over time, as well as other measurements like sea level. For animation tasks, however, we can frequently obtain the correct values for any quantities by measuring the real-world, and the modeling task is to produce estimates that are close to the known, true values. in a city traffic model, for example, we might seek to match the expected density of traffic over different times of the day. It is not generally the case that such estimates can be computed directly from the model in an analytic form, so we typically view a simulator as the generator of statistics from which quantities will be estimated.

The statistical view of simulation admits a means of measuring the quality of a simulation model:

> The quality of a simulation model can be judged by how well the estimates it produces match important estimates in the real world.

In this vein, climate models are validated by testing their performance on historic climates, for which records exist. A city model for entertainment purposes may be of sufficient quality if it produces something like peak hour behavior, or increasing density around major intersections.

Statistics also provide a means to compare two different simulation models intended to capture the same behavior. For example, Hodgins et. al. [42] uses statistics gathered from a user experiment to explore the effect of geometry on the perception of simulation quality, as does Chenney [16] to validate a model for simulation culling (see chapter 4). Statistical approaches are preferable for two reasons: they provide a level of abstraction away from individual simulation runs toward the overall behavior of a simulator, and they avoid the potentially impossible task of setting up identical initial conditions and parameters when comparing simulation to reality.

The following things determine the outcome of a simulation, and we are free to manipulate both in order to improve the quality of a simulation:

- The model to be simulated. That is, the set of equations to be solved or rules to be followed. For the city simulation, the model includes rules for how drivers react to various events, and physically based equations of motion for how cars respond to the current road conditions.

- The values of any variables in the model, including the initial conditions for any dynamic variables and other parameters. In the city this includes the initial state of every car and driver, and also the parameters to the model such as the properties of each road surface.

The designers of simulation models tend to adjust both the model and the parameters to it. Much of the research on simulation for computer graphics has focussed on developing models that are fast to compute and then setting parameters for the model to generate realistic results. For example, in rigid body simulation (chapter 3) the models used for the interaction between colliding bodies make gross simplifications to the physics of the situation, but provide parameters for things like friction and restitution to allow visually acceptable results.

Much of this thesis is concerned with statistical measures of simulator behavior. Chapter 2 uses probabilistic techniques to generate multi-body constrained animations, while ensuring that the animations produced are plausible according to the user specified model of the world. The latter half of this thesis addresses scalable simulation, where the goal is to generate simulation models that scale to very large environments while remaining statistically indistinguishable from poorly scaling variants.

1.2 Directing Simulations

It is frequently necessary to construct a simulation that generates a specific outcome. For instance, we might like our city simulation to include a car running a red light in front of the trainee ambulance driver. The red light runner must at all times respect the correct behavior of the system. Failure to do so might tip off the trainee to the impending event, or even break the sense of presence.

Generating a specific outcome while respecting the behavior of the system requires choosing parameters to the model that, as the simulation evolves, lead to the desired outcome. This may be difficult for several reasons:

- The outcome may be highly sensitive to changes in the parameter values, or *chaotic*. For instance, slightly changing the speed of a car can cause it to be stopped by a red light, which has a significant effect on its subsequent location. Many search or optimization algorithms perform poorly under such circumstances.

- There may be many different ways to generate the desired outcome, but only one is necessary. The problem is in sensibly deciding which path to take.

- Derivative information relating the outcome to the parameters may be difficult or impossible to obtain. The outcome may also be discontinuous with respect to the parameters, as is the case with the location of the car with respect to its speed, due to the effect of red lights. This makes it difficult to employ any method the explicitly or implicitly depends on derivative information.

Passive systems (those without explicit control parameters), present additional problems for a director. The entire outcome of the simulation is fixed once the world model and its parameters are chosen. Direction is difficult in these cases because there may be few parameters to adjust, and each parameter potentially influences the entire animation.

Chapter 2 of this thesis is concerned with the direction of one particular class of simulation: multi-body rigid-body models in which many rigid objects interact through collisions. These simulations exhibit most of the problematic features described above. They are often chaotic, the outcome may be discontinuous with respect to the initial conditions, and there may be few obvious parameters to adjust.

The solution presented here extends simulation models to include plausible sources of uncertainty, and hence defines a probability density function over the space of possible animations. We then use a Markov chain Monte Carlo algorithm to sample multiple animations that satisfy constraints. A user can choose the animation they prefer, or applications can take direct advantage of the multiple solutions. The technique is applicable when a probability can be attached to each animation, with "good" animations having high probability, and for such cases a definition of physical plausibility is provided.

The approach is demonstrated with examples of multi-body rigid-body simulations that satisfy constraints of various kinds. The results indicate that it is possible to generate animations that are true to a physical model, significantly different from each other, and yet still satisfy the constraints. It is hoped that the technique will provide insight into other problems with similar characteristics, such as the city red light runner example.

1.3 Scalable Simulation

Simulation for virtual environments requires performing some amount of computation for every frame in order to update the state of the world. Frequently, the cost of computation is the limiting factor in deciding how many moving objects will populate the world, and how realistic their motion will be.

One obvious way around this problem is to make the simulation faster. This benefits not only interactive applications, but also off-line processes that perform iterative design of some sort, such as the Markov chain Monte Carlo algorithm used for direction. Chapter 3 of this thesis describes a new rigid-body simulation algorithm that is faster than most previous algorithms by a factor up to linear in the number of moving objects. An implementation of this algorithm was used for many of the examples in chapter 2.

However, regardless of how fast any one simulation is, it is expected that virtual worlds will require vastly more simulated motion than can all be computed at a reasonable cost. Scalability has long been a research topic for the rendering aspect of virtual worlds, and systems such as the Berkeley Soda Hall walkthrough project have been developed to demonstrate geometric culling [30, 84] and level of detail [29]. Scalable simulation for highly dynamic environments will require corresponding technology, ideally so that the cost of simulation grows with the complexity of the motion in view, regardless of the total size of the world. The latter part of this thesis is devoted to technology for *simulation culling*, by which motion that is not visible at any moment is not computed.

The basic premise behind culling is that most of an environment is not visible most of the time. In a typical city, for instance, at most a few blocks are visible at any moment, leaving most of the environment hidden behind buildings or hills. The same is true for most urban environments, and many natural ones. Given that the aim of a virtual environment is to present a realistic view of the world to a user, there is no point in performing work for objects that will never be seen. Geometric culling recognizes this by not attempting to render such objects. Similarly, simulation culling attempts to avoid performing work for motion that will never be seen.

Chapter 4 gives an overview of simulation culling. Two key problems must be addressed in order to cull systems efficiently and effectively: the completeness problem and the consistency problem. Completeness is concerned with making sure that the viewer sees everything they should, even when systems out of view are not fully computed. Consistency ensures that when the viewer does see something, it is consistent with their previous experience and current knowledge of the environment. At a high level, solutions to both problems take into account what a viewer can and cannot know about the environment by building probabilistic models of motion in the environment.

If the geometric extent of the objects in a simulation can be tightly bound for all time, then the completeness problem does not arise, because it is possible to easily determine which objects in the simulation the viewer should see. In such cases the consistency problem remains, and chapter 5 describes several possible solutions, each targeting a specific type of simulation that has been out of view for more than a certain time. With a range of standard solutions available, and rules for which tool to apply in a given case, a model transformation tool is described that automatically converts a basic model from a restricted class of systems into one suitable for culling. Implemented in VRML [90] and Java [2], the tools enable the rapid creation of cullable models for inclusion into any VRML environment.

Most interesting environments, however, contain objects that roam over relatively large distances, and the completeness problem must be addressed when culling. Chapter 6 discusses completeness in detail. To explore various aspects of the problem, a city model was designed and populated with cars that move around the streets. Through the use of probabilistic models of each car's motion while out of view, it is possible to concentrate most of the computational effort on those cars that are visible, leading to a more scalable environment. The work described only scratches the surface of modeling for scalable simulation, and a large set of future research directions are proposed.

This thesis treats direction and scale as largely independent problems, yet we show that at an abstract level they are similar: both are manifestations of sampling problems. This thesis concludes with a look at how the MCMC direction algorithm might be integrated with a virtual world that does simulation culling, and the problems that must be addressed to make such a world possible.

Chapter 2

Directing Simulations

Collision intensive multi-body simulations are difficult to constrain because they exhibit extreme sensitivity to initial conditions or other simulation parameters. Furthermore, passive systems lack a large set of control variables, leaving only the model and the initial conditions under the control of an author. Random parameters to the model are one natural way of extending the range of animations that can be produced. Adding uncertainty to a model also helps when looking for animations that satisfy constraints [7], because it adds physically motivated degrees of freedom in useful places. For example, we can control tumbling dice by placing random bumps in specific places on the table, rather than by adjusting the initial conditions of the throw. The bumps are more effective because a small change to a bump part-way through the animation has a limited effect on where the dice land, but a small change in the initial conditions generally has an unpredictable effect. It is difficult to design efficient control algorithms for the latter case.

As discussed by Barzel, Hughes and Wood [7], adding randomness to a simulation gives additional benefits:

- The real world contains fine scale variation that traditional simulation models generally ignore. We can use randomness to model this variation by, for instance, replacing a perfectly flat surface with one speckled with random bumps (the same random bumps used for control above). Animations generated with the new model can more accurately reflect the behavior of the world. In training environments, this results in the subject developing skills more compatible with the real world: a driver trained on simulations of bumpy roads will be better prepared for real world road surfaces.

- Visually, procedural animations can be more believable when uncertainty is added. Without uncertainty, a perfectly round ball dropped vertically onto a perfectly flat table moves strangely, a situation that may be improved by slightly perturbing the collisions to make the ball deviate from the vertical.

In a world with uncertainty, we generally expect a constrained problem to have multiple solutions. It is difficult to know beforehand what solutions are available, which compounds any difficulties a user may have in codifying their preferences. Hence, it is perverse to use a solution strategy that seeks a single answer, rather, we prefer a technique that produces many solutions that reflect the range of possible outcomes. While for feature animation a user is expected to choose the one animation they prefer, other applications benefit directly from multiple solutions:

- Computer game designers can use different animations each time a game is played, making it less predictable and potentially more entertaining.

- Training environments can present trainees with multiple, physically consistent scenarios that reflect the physics and variety of the real world.

We generate multiple animations that satisfy constraints by applying a Markov chain Monte Carlo (MCMC) algorithm to sample from a randomized model. A user supplies the model of the world, including the sources of uncertainty and the simulator that will generate an animation in the world. The user also supplies a function that gives higher values for “good” animations — those that are likely in the world and satisfy the constraints. Finally, a user must provide a means of proposing a new animation given an existing one. The algorithm we describe in this chapter generates an arbitrarily long sequence of animations in which “good” animations are likely to appear.

In the following sections, along with the algorithm, we describe some of the models we use and how we sample from them, discussing examples from the domain of collision intensive rigid-body simulation. No previous algorithm has been shown for the range and complexity of the multi-body simulations we present.

2.1 Related Work

The idea of *plausible motion simulation*, including the exploitation of randomness to satisfy constraints, was introduced by Barzel, Hughes and Wood [7]. They show solutions to constrained problems where, for instance, a billiard ball is controlled by randomly varying the collision normal each time it hits a rail. The solutions are found by constructing the set of paths and then searching within the set using a path tracing approach. This method does not extend to more complex systems, because the path spaces rapidly become extremely complex. Instead, we extend their work by introducing the idea of sampling (instead of searching), giving a precise definition of plausibility, and by demonstrating MCMC's effectiveness on a wide range of difficult examples.

Motion synthesis algorithms aim to achieve a goal by finding an optimal set of control parameters and (sometimes) initial conditions. The goals described in the literature include finding good locomotion parameters [3, 18, 36, 41, 69, 77] and finding trajectories that satisfy constraints [6, 11, 20, 33, 37, 52, 94]. Some techniques [6, 11, 20, 33, 37, 52, 94] exploit explicit gradient information in a constrained optimization framework, but fail if the problem is too large (Popović discusses ways to reduce the problem size [74]) or the constraints are highly sensitive to, or discontinuous in, the control parameters. Randomized algorithms, such as simulated annealing [36, 41], stochastic hill climbing [18], or evolutionary computing [3, 69, 77, 83], do not require gradients and may be suitable for collision intensive systems — Tang, Ngo and Marks [83] describe an example in which a set of frictionless billiard balls are constrained to form a pattern at a particular time.

Most existing methods are designed to return a single "best" animation, and hence may ignore other equally good, or even preferable solutions. The idea of an optimum is particularly ill suited to graphics because the optimality conditions may be derived from visual appeal, which is extremely difficult to codify in an objective function. The evolutionary computing solutions can exhibit variations within a population, which Auslander et. al. [3] refer to as different styles, but the number of examples is limited by the population size.

Multi-body constraint problems are good candidates for a *Design Galleries* [55] interface, in which a user browses through sample solutions to locate the one they prefer. Our work addresses the sampling aspect of a Design Galleries interface for multi-body constrained animations, but we consider other aspects of the interface only as future work.

2.2 Animation Distributions

The MCMC algorithm distinguishes itself from motion synthesis approaches by generating multiple, different, "good" animations that satisfy a set of constraints, but no "best" animation. To generate multiple plausible constrained animations, we must provide a model of the world defining:

- The objects in the world and their properties, including the sources of uncertainty.
- The simulator for generating animations in the world.
- The constraints to be satisfied by the animations.

For example, in a 2D animation of a ball bouncing on the table, we might have uncertainty in the normal vectors at the collision points, a constraint on the resting place of the ball, and a simulator that determines what happens when a 2D ball bounces on a table with arbitrary surface normals. We will use this example, from [7], throughout the next two sections.

A simulator used with our approach need not be physically accurate, or even physically based. Our 2D ball simulator is obviously non-physical, and the simulator described in chapter 3, used in other examples, has some problems with complex frictional behavior (section 2.4.2). In any case, we assume that if the simulator is given a plausible world as input it will produce a plausible animation, according to some definition of plausibility (see section 2.2.3).

2.2.1 Incorporating Uncertainty

We define a function, $p_w(A)$, representing the probability of any possible animation A that might arise in the world model. Intuitively, $p_w(A)$ should be large for animations that are likely in the world, and low for unlikely animations. For the 2D ball example, $p_{w,ball}(A)$ should be high if all the normal vectors used to generate the animation were close to vertical, and low if most of them were far from vertical. Let us further insist that $p_w(A)$ be non-negative and have finite integral over the domain implied by the random variables in the model, so that we can view $p_w(A)$ as an unnormalized joint probability density function defined on the space of animations.

Expanding on the 2D ball example, let us describe the direction of the normal vector for each collision i as an independent random variable, θ_i, distributed according to the (bell-shaped) Gaussian distribution with standard deviation of, say, 10.0 degrees. In that case we get:

$$p_{w,ball}(A) = p_{w,ball}(\theta_1, \theta_2, \ldots) \propto \prod_i e^{-\frac{1}{2}\left(\frac{\theta_i}{10.0}\right)^2}$$

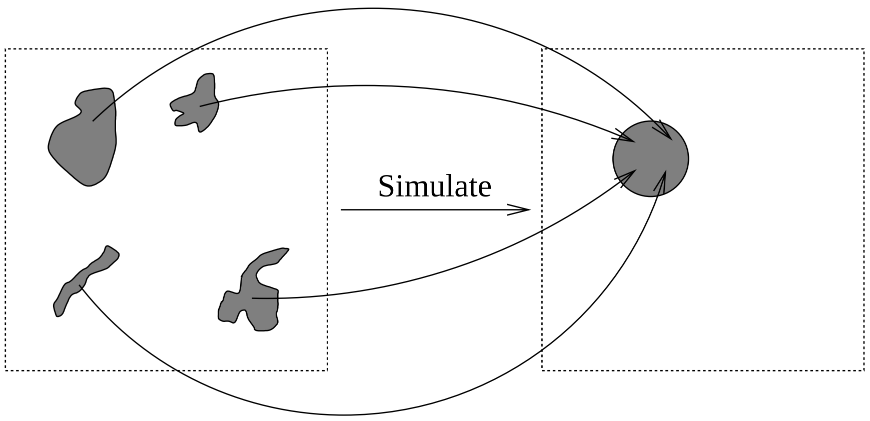


Figure 2.1: *Sampling from animation distributions conditioned on constraints is likely to be difficult, because a constraint may be simple in the output space while its image in the space of random variables may be very complex. For example, in the figure the constraint requires the output to fall within a circle, yet the random variables that lead to outcomes in that circle come from disjoint regions with complex structure. This is because the inverse simulation mapping may be neither smooth nor even continuous. Directly sampling animations that satisfy the constraints requires sampling points within the complex regions of the random variable state space, which is difficult to do efficiently.*

which is the product of density functions for each collision normal. Note that we are ignoring normalization constants, an omission we justify in section 2.3. Also, we could in principle measure a real table to infer the true joint distribution on surface shape, and use that instead.

2.2.2 Constraints

If we restrict our attention to animations that satisfy constraints, we are concerned with the distribution function $p_w(A|C)$, which is the conditional distribution of A given that it satisfies the constraints C. For the 2D ball example, if we want the ball to land in a particular place, we could generate samples from $p_{w,ball}(A|C)$ using an inverse approach: join the ball's start point to its end point using a sequence of parabolic hops and then infer which normal vectors were required to generate such a trajectory. However, using this approach we cannot directly ensure that the animation we generate is likely in the world, because it is difficult to know which hops to use to get a set of likely normal vectors.

Unfortunately, it is frequently impractical to sample directly from $p_w(A|C)$, because there is no way to find, without considerable effort, any reasonable animation in which the constraints are satisfied. Doing so would require directly specifying values for all the random variables such that the animation using those variables satisfies the constraints. For collision intensive examples in particular, animations in which the constraints are satisfied are likely to be spread around the space of random variables in ways that are difficult to describe, and hence sample from directly (figure 2.1). Sampling randomly in the hope of hitting a good animation is likely to fail because the good animations may be very sparse in the space. Attempting to invert the simulation to locate good samples also fails in general, because there may be constraints at multiple points in the simulation and some models may not be invertible, such as those for collisions with friction.

In such cases (like all the examples in this chapter), we expand $p_w(A)$ to include a term for the constraints, resulting in a density function $p(A)$. The new intuition is that $p(A)$ will be large for animations that are likely in the world *and* satisfy the constraints, and small for animations that are either implausible in the world or don't satisfy the constraints. We will refer to $p(A)$ as the *probability* of an animation. Note that now even animations that don't satisfy the constraints have non-zero probability, so if we sample from $p(A)$ we may get animations that don't satisfy the constraints, which we must discard.

For the examples in this chapter, we define:

$$p(A) \propto p_w(A) p_c(A) \tag{2.1}$$

where $p_c(A)$ depends only on how well the animation satisfies the constraints. If we want our 2D ball to land at a point whose distance, d, from the origin is small, we can define

$$p_{c,ball}(A) \propto e^{-\frac{1}{2}\left(\frac{d}{\sigma_d}\right)^2}$$

which is the Gaussian density function with standard deviation σ_d, suitable values for which we discuss in section 2.4.1. This function gives higher values for distances near zero, and lower values as distances increase. Hence, for the 2D ball example:

$$p_{ball}(A) \propto e^{-\frac{1}{2}\left(\frac{d}{\sigma_d}\right)^2} \prod_i e^{-\frac{1}{2}\left(\frac{\theta_i}{10.0}\right)^2}$$

The multiplicative form used on equation 2.1 can be viewed in the following way (figure 2.2). We begin with a distribution on animations implied by $p_w(A)$, which gives a density for every possible world according to how likely it is (higher density means more likely). We then multiply the density for every animation A by an amount $p_c(A)$ according to the desirability of A in terms of the constraints. This skews the density to favor those worlds that generate the desired outcome, or looked at the other way, skews those animations that satisfy the constraints to favor those that arise from likely worlds.

In the following section (2.3) we describe a technique for generating animations such that those with high probability will appear more frequently than those with lower probability, but even some low probability events will occur — as in the real world, unlikely things sometimes happen. In other words, we will sample according to the distribution defined by $p(A)$.

2.2.3 What does “Plausible” mean?

The restrictions on $p(A)$ are quite weak, so we can describe many types of uncertainty and a wide variety of constraints. By phrasing the problem as one involving probabilities, we can leverage a wide range of mathematical tools for talking about plausible motion, and make strong statements about the properties of the animations we generate (under certain conditions, see section 2.3). We can also outline what it means to be physically plausible:

> A model, including its simulator, is plausible if the important statistics gathered from samples distributed according to $p(A)$ are sufficiently close to the real world statistics we care about.

This is a very general definition of plausibility, because we say nothing about which statistics we might care about, or what it means to be sufficiently close. For example, to validate a pool table model we could run simulations of virtual balls on a table, and analyze video of real balls on a real table, then compare statistics such as how long a ball rolls before coming to rest. For entertainment applications, we would care less about the quality of the match than if we were trying to build a training simulator for young pool sharks.

Our measure extends the traditional graphics idea of plausibility — “if it looks right it is right” — by allowing for definitions of statistical similarity other than a user’s ability to detect a fake. However, for many applications, particularly involving motion, a viewer’s ability to distinguish real from artificial remains the primary concern [42].

2.3 MCMC for Animations

We use the Markov chain Monte Carlo (MCMC) method [32, 47] to sample animations from the distribution defined by $p(A)$. MCMC was originally developed to estimate

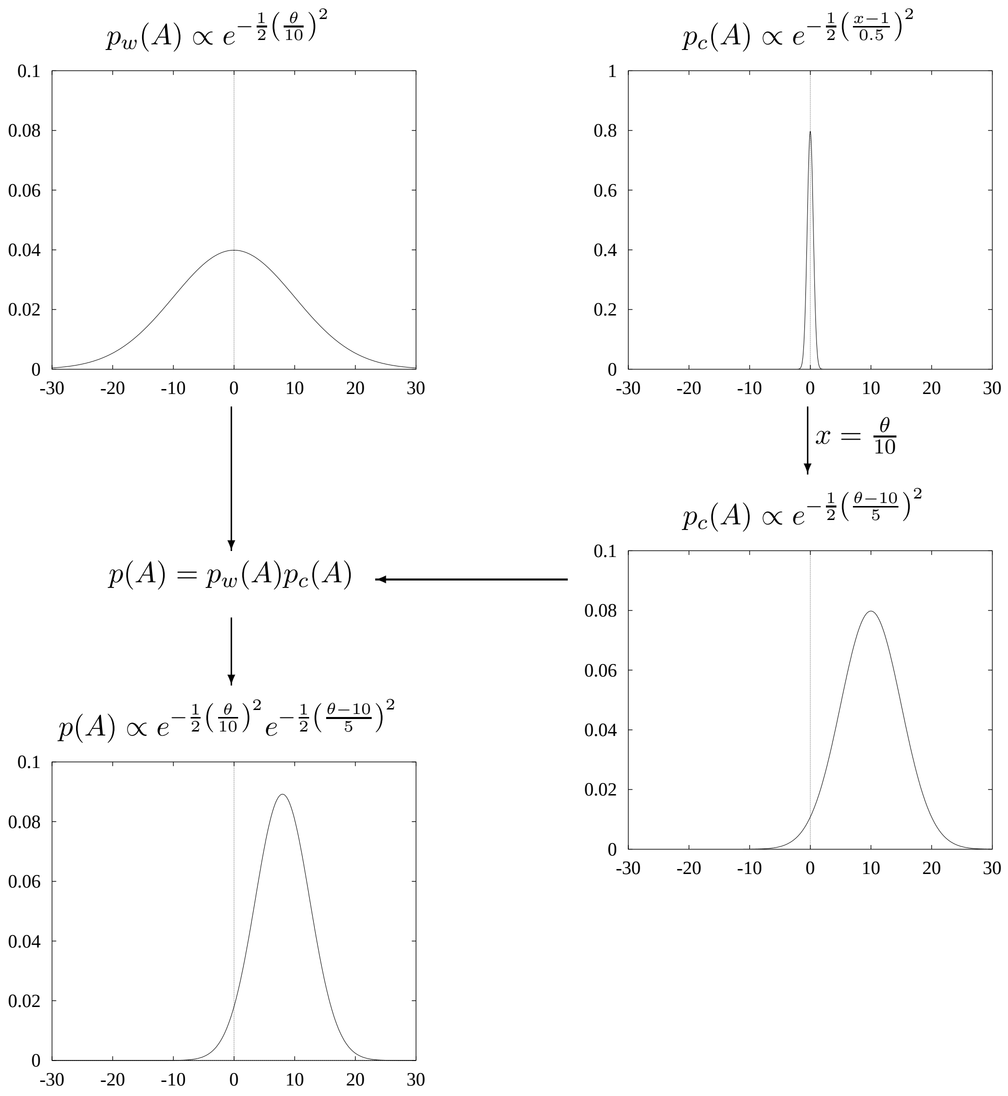


Figure 2.2: *Construction of an animation distribution for a single ball bounce, with a constraint on the landing position, x, at the second bounce. Assume that x is related to the collision normal inclination, θ, by $x = \frac{\theta}{10}$, and we would like the ball to land at $x = 1$. Top left is a plot of $p_w(A)$ for this scenario, using a Gaussian distribution that favors normals near vertical. Top right is a constraint Gaussian distribution expressed in terms of the landing position, x. Because we have an explicit relationship between x and θ, we can express the constraint distribution in terms of θ (center right). Combining $p_w(A)$ and $p_c(A)$ yields the animation distribution $p(A)$. Note that this distribution peaks at normal orientations closer to zero when compared to the constraint distribution, reflecting the influence of the world model over the outcome of the simulation. We could more strongly favor the desired outcome by reducing the "standard deviation" on the constraint distribution. Also, in most cases it is not possible to express the constraint probability in terms of the random variables directly. Instead, we simulate to determine the outcome and evaluate $p_c(A)$ and hence $p(A)$.*

values in statistical physics problems, such as the energy in a lattice. More recently it has been applied to many other research areas [32]. MCMC has several advantages for the task of sampling from distributions of animations:

- MCMC generates a sequence, or *chain*, of samples, $A_0, A_1, A_2, \ldots$, that are distributed according to a given distribution, in this case $p(A)$.
- Apart from the initial sample, each sample is derived from the previous sample, which allows the algorithm to find and move among animations that satisfy constraints.
- If available, domain specific information can be incorporated into the algorithm, making it more efficient for special cases. On the other hand, the algorithm does not rely on any specific features of a model or simulator, allowing its application in a variety of situations.

Our MCMC algorithm for generating animations begins with an initial animation then repeatedly proposes changes, which may be accepted or rejected. Explicitly:

Algorithm 2.1 MCMC algorithm for sampling constrained animations

```
1   initialize($A_0$)
2   simulate($A_0$)
3   repeat
4       propose($A_c, A_i$)
5       simulate($A_c$)
6       $u \leftarrow$ random$(0,1)$
7       if $u < \min\left(1, \frac{p(A_c)q(A_i|A_c)}{p(A_i)q(A_c|A_i)}\right)$
8           $A_{i+1} \leftarrow A_c$
9       else
10          $A_{i+1} \leftarrow A_i$
```

Line 1 gives initial values to all the random variables in the world model. On line 4, a new animation, A_c, is proposed by making a random change to the previous animation, A_i. The details of this change are application specific. For example, in the 2D ball model of section 2.2 it might involve, for each normal, choosing to change it with probability one half and, if it is to be changed, adding a random offset uniformly distributed on $(-5, 5)$ degrees (for reasons discussed in section 2.4.1). The probability of making changes is defined by the *transition* *probability*, $q(X|Y)$, which is the probability of proposing animation X if the current animation is Y. For the 2D ball, the transition probability is:

$$q_{ball}(X|Y) \propto \left(\frac{1}{2}\right)^n \cdot \left(\frac{1}{5-(-5)}\right)^k$$

where n is the total number of collisions (assumed fixed) and k is the number of collisions that were changed. The first factor is the probability of choosing the particular set of normals to change, and the second factor codes the probability of choosing a particular offset for each normal that is changed.

The transition probabilities, along with the probabilities of the animations, are used in computing the *acceptance probability*, which is the probability of accepting the proposed candidate (line 7):

$$P_{accept} = \min\left(1, \frac{p(A_c)q(A_i|A_c)}{p(A_i)q(A_c|A_i)}\right)$$

From the formula we can see that the algorithm will tend to accept a candidate animation with higher probability than the current sample, unless the transition probabilities are significantly out of balance. Often, as in the 2D ball example, the transition probabilities are symmetric — $q(X|Y) = q(Y|X)$ — and will cancel. Note also that only the ratios of probabilities appear, so we can use functions that are only proportional to true probability density functions (section 2.2.1).

The proposal mechanism is one of the key factors in how well the algorithm will perform in a particular application. In practice, proposals are designed through intuitive reasoning and experimentation, using past experience as a guide. In section 2.4 we describe proposal mechanisms for several situations.

The MCMC algorithm guarantees that the samples in the chain will be distributed according to $p(A)$, as the number of samples approaches infinity and provided certain technical conditions are met [32]. Intuitively, these conditions require that the algorithm have some probability of reaching every point in the domain that has non-zero probability density, and that there be no cycles as the algorithm moves among samples. Hence with a little care we can be certain that the samples our algorithm generates truly reflect the underlying model, and if this model is plausible (section 2.2.3), the collection of samples will be plausible. It is also the case that the samples in the chain will never satisfy the constraints if the underlying model says they cannot be satisfied. For instance, if a bowling simulator cannot capture complex frictional effects, animations that bowl the seven-ten split can never be found (see section 2.4.2).

MCMC has been used in graphics to generate fractal terrain that satisfies point constraints [82, 89]. The samples generated by an MCMC algorithm may also be used to estimate expectations, as in Veach's Metropolis algorithm for computing global illumination solutions [87]. For the task of sampling animations we are not concerned with expectations, so we can use short chains, as long as are necessary to satisfy a user with several different animations of high probability.

2.4 Examples

We are interested in four things when designing an MCMC algorithm for generating animations:

- Is the motion plausible? We assume that the simulator produces plausible motion, so we are left to ensure that the distributions we use for the model are reasonable.
- How long does it take to find a sample that satisfies the constraints?
- How rapidly does the chain move among significantly different samples, or *mix*? Chains that mix faster are desirable because they produce many different animations quickly.
- How many of the samples satisfy the constraints well enough to be useful?

The following examples discuss issues in building models, defining constraints and selecting proposal strategies, all of which influence the behavior of the algorithm.

2.4.1 A 2D Ball

In the 2D ball example of section 2.2 a ball bounces on a table, starting in a fixed location and undergoing, for simplicity, a fixed number of collisions. For each collision we specify a random normal vector. The aim is to sample these normal vectors such that the ball comes to rest close to a particular location. As a specific case, we will drop the ball from above the origin at a height of $4.5D$, where D is the diameter of the ball, use five collisions, and specify that it come to rest near $x = D$ on the sixth collision.

The simulation model is: the ball moves ballistically between each collision, when the velocity of the ball is reflected about the corresponding normal vector and the normal component of velocity is scaled by $\frac{1}{\sqrt{2}}$. This model is not physically plausible (for instance, we are ignoring rotation effects), but for this example we value simplicity.

Uncertainty and Constraints

The probability of an animation is described in section 2.2.1, but probabilities (the values of density functions) can be very large numbers, so in practice we work with their logarithm. In this case, with x the horizontal position of the sixth collision:

$$\log(p(A)) = -\frac{1}{2}\left(\frac{x-D}{\sigma_d}\right)^2 - \frac{1}{2}\sum_{1\leq i\leq 5}\left(\frac{\theta_i}{10.0}\right)^2 + C$$

for some constant C, which will cancel out when computing the acceptance probability.

The value of the constraint standard deviation, σ_d, has a major effect on the samples generated by the chain. Say we choose a small value for σ_d, corresponding to a very tight constraint because only values of x very close to D give high values for $p(A)$ and all other landing points have very low probability. From the initial animation, the chain will move to some high probability animation close to the constraint. But, once there, almost no new proposals are accepted (most candidates will be far from the constraint and have very low probability) and the user sees few different animations — an undesirable situation.

Alternatively, say we choose a large value for the standard deviation, corresponding to a weak constraint. Then $p(A)$ is relatively high for a wide range of landing positions. The result is undesirable: the chain will contain many high probability animations that are far from the constraints.

Hence we must choose a value for σ_d that is high enough to promote different samples but low enough to enforce the constraint. In this example we use a value of $0.1D$, where D is the diameter of the ball, which, as figure 2.4 shows, leads to the generation of very different samples that generally are close to the constraint. In this example, the algorithm is not very sensitive to the exact value for σ_d (anything within a factor of five works fine) and it is possible to experimentally evaluate a few values on short chains and choose the best, which in this case took only a few minutes.

In other applications there is no guarantee that we can achieve both good constraints and good mixing. In such cases the algorithm must run for many iterations to generate different samples, which may take prohibitively long. The tumbling dice example of section 2.4.4 is a borderline example in which we can satisfy constraints but mixing is poor. In such cases we can generate more samples by running multiple chains in parallel [54].

Proposals

The proposal mechanism, which specifies normal vectors for a candidate animation, A_c, given those for the current animation, A_i, provides a means of moving around the space of possible normal vectors:

Algorithm 2.2 Proposal algorithm for 2D Balls

```
for $j = 1$ to $5$
    $A_c.normal[j] \leftarrow A_i.normal[j]$
    if $\mathsf{random}(0,1) < 0.5$
        $A_c.normal[j] \leftarrow A_c.normal[j] + \mathsf{random}(-5,5)$
```

This proposal changes some of the normals by an amount between minus one half and half their standard deviation of 10.0 degrees. For good mixing it is important to allow more than one normal to be changed at once, because the effect of each change on the landing position (and hence the constraint) can then cancel. The alternative, changing only one normal, makes it very difficult to change the first collision normal, because any but the smallest change will move the ball far from the desired landing position, and hence be rejected. The size of the offset we add is chosen to allow both small changes and relatively large changes, but not so large as to shift the normals too far from their mean in one step, which would reduce their probabilities and result in rejection of the candidate animation.

An Example Chain

We ran the MCMC algorithm and generated a chain containing one thousand samples (many of these are repeats, arising when a candidate is rejected). Figure 2.4 plots the horizontal resting position of each sample. The first sample was initialized with randomly chosen normals, and came to rest a long way from the constraint. But within twenty iterations the chain moved toward a good location. The bumpiness of the graph indicates good mixing, because flat spots would indicate many repetitions of one sample as candidates were rejected. The majority of animations have the ball coming to rest within $0.1D$ of the desired position, indicating that σ_d is sufficiently small to enforce the constraint.

Three (randomly chosen) samples from the chain are shown in figure 2.3. They do not differ greatly from what one would expect: the ball tends to take an early bounce toward the constraint and keep moving in that direction, with later collisions adjusting it’s final position.

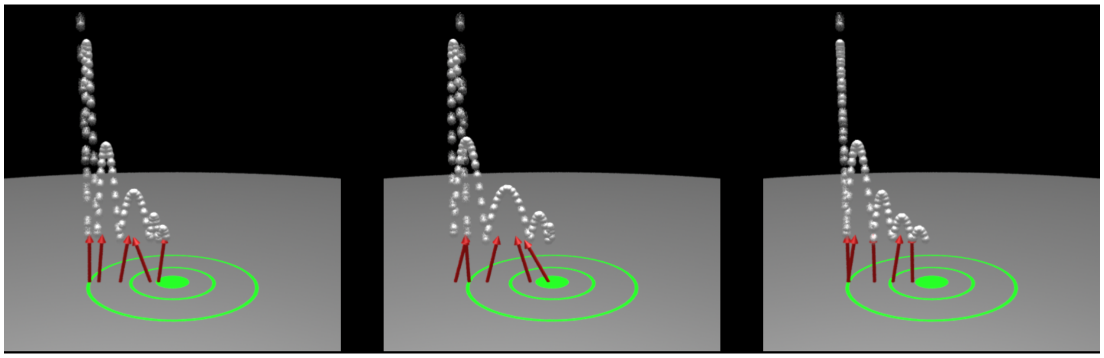


Figure 2.3: *Three sample paths from the 2D ball example, plotting the trajectory of the center of the ball (although the plot is 3D, the ball moves only in 2D). The green target is centered on the constraint. Each red arrow is located at a collision point and indicates the direction of the normal vector used at that point. Note that in each example one of the earlier normals pushes the ball toward the constraint, and later normals refine the final position. One ball bounces slightly away from the constraint before moving toward it, which is not implausible.*

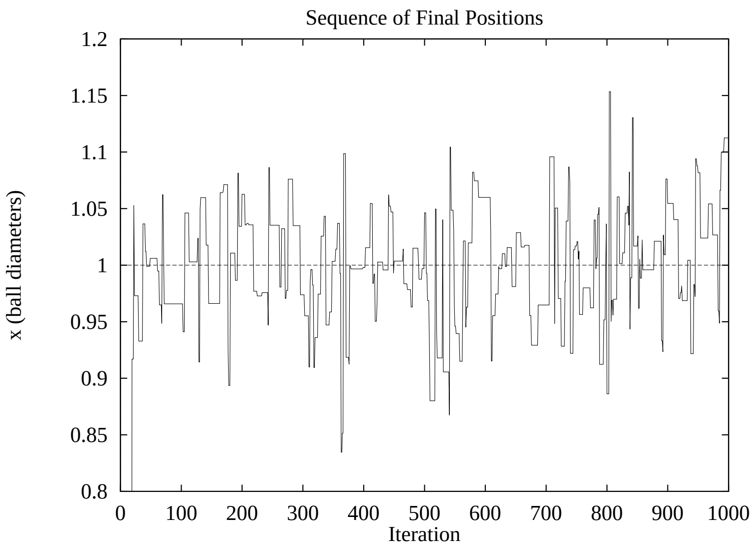


Figure 2.4: *The resting position of the first one thousand samples in a chain for the 2D ball example. The roughness of this graph indicates good mixing, and most samples are close to the constraint (the majority within $0.1D$). The position of the first few samples are far from the constraint (off the graph), but the chain moves to samples within twenty iterations.*

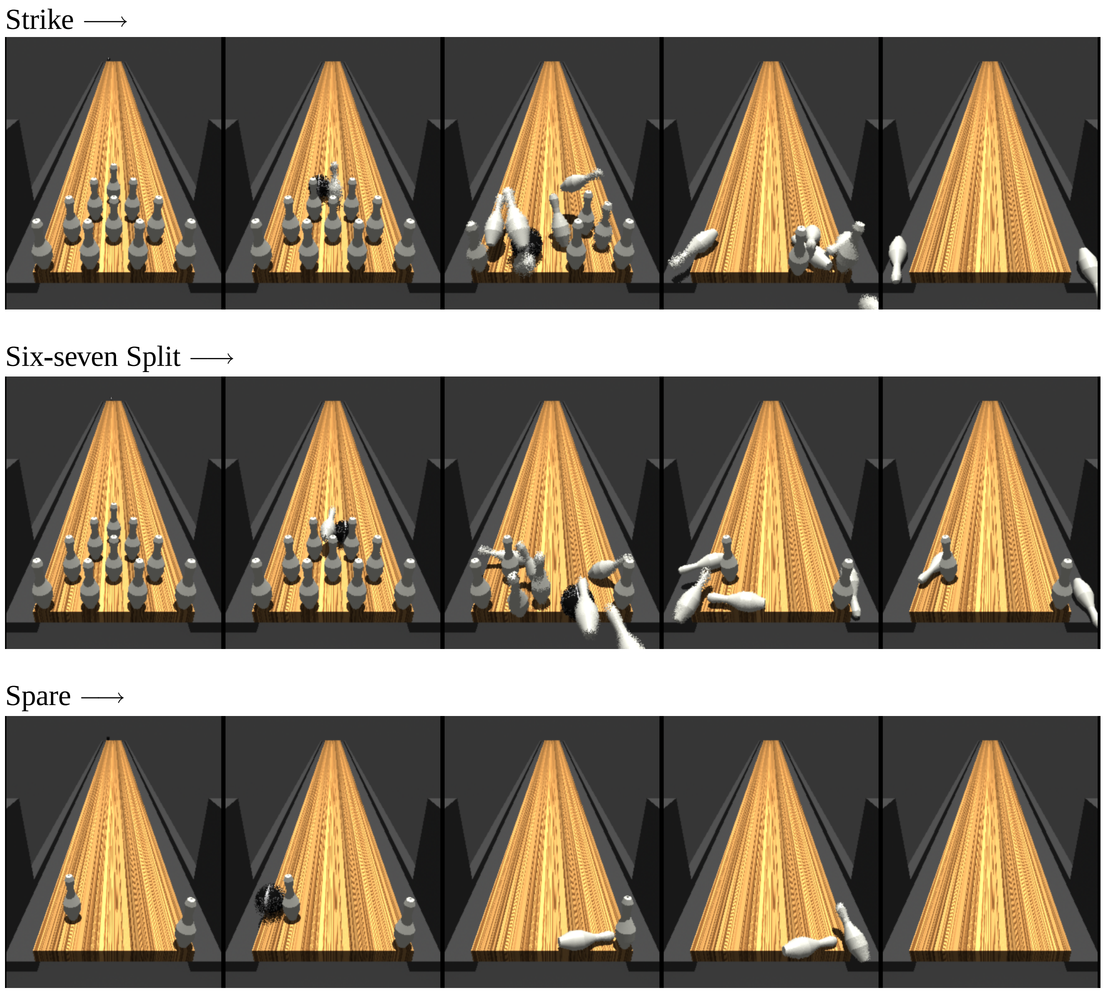


Figure 2.5: *Frames from three bowling examples. The initial conditions for the ball and the pin locations are random variables. Given an initial and final pin configuration, the MCMC algorithm samples particular values for the random variables that lead to the desired shot. In this case, we demanded a strike, a six-seven split and the corresponding spare.*

2.4.2 Bowling

In this scenario the aim is to animate any particular ten-pin bowling shot (a goal suggested by Tang, Ngo and Marks [83]). The physical model is implemented by an impulse-based rigid-body simulator, described in detail in chapter 3 of this thesis. We model the bowling ball, the lane with simplified gutters and side walls, and the pins. All the models are roughly based on the rules of bowling, including variations allowed by those rules (details are given in appendix A.1.1):

- The ball is simulated as a sphere, with variable radius, density, initial position, initial velocity and initial angular velocity.
- The lane is fixed with regulation length and width, and includes rectangular gutters and side walls starting in line with the front pin.
- Each pin, of fixed shape and mass, has its initial position on the lane perturbed by a small random amount.

The coefficients of friction and restitution between all the components are fixed. The probability $p_w(A)$ is proportional to the product of the distribution functions for each of the random variables in the model.

Constraints

The simulation begins with a subset of pins specified by the user, so we can specify the initial conditions for bowling spares. The user also sets the constraint by stating which pins should be knocked down and which should remain standing. We are unable to propose candidates for the MCMC algorithm that are certain to satisfy the constraints (section 2.2.2), so we assign non-zero probability to every possible outcome, but assign higher probability to those outcomes that are closer to the target, and the highest probability to outcomes matching the target. This is achieved with the Gibbs distribution function:

$$p_c(A) \propto \lambda^{k+m}$$

for some constant $\lambda > 1$ with k the number of pins that end up correctly standing or knocked down, and m the number of standing pins that have not moved far beyond their initial position. Animations that do not meet the goals will sometimes appear in the chain (they have non-zero probability), but these would not be shown to a user. The samples that remain are correctly distributed according to the conditional probability $p(A|C)$, the distribution of animations in which the constraints are fully satisfied. The constraint involves a term derived from the pins' final position because some simulations result in the pins being pushed but not knocked down — behavior we wish to discourage.

The value of λ affects the proportion of animations in the chain that must be discarded for not satisfying the constraints. High values for λ give animations satisfying the constraints much higher probability, making them more likely to appear in the chain. But the chain mixes better if some “bad” animations appear. Say only perfect animations appear, then getting to a significantly different animation requires making a big change that also happens to get all the pins correct, which is unlikely. If some pins are not correct, a big change only has to get the same number of pins correct, and they can be different pins. A low value for λ makes it easier to accept an animation with some incorrect pins, make big changes, and then move toward a different, fully correct state.

For this example, we used $\lambda = e^{2.5}$, which gives a wide variety of animations that satisfy the constraints. Animations that improve the constraints are favored enough to ensure that good animations come up often, but not so much as to inhibit mixing.

Our use of the Gibbs distribution was motivated by other applications of the MCMC algorithm, such as counting the number of perfect matchings in a graph. It is known [46] that there is an optimal λ that balances the concerns outlined above, but that the algorithm is relatively insensitive to its exact value. Experience suggests that many applications may exhibit similar behavior [78]: there exists a range of values for λ that give the chain good properties, and one such value may be found through experiment. Our results are consistent with this (also see section 2.4.3).

Proposals

Our proposal mechanism for bowling randomly chooses to do one of several things:

- Sample new values for all the random variables.
- Change the radius, density or initial conditions of the ball.
- Change the initial position of some pins.

The details are given in appendix A.1.2

The first proposal strategy, which changes every random variable in the simulation, serves to make very large changes in the simulation. These are desirable as a means of escaping low probability regions, which we discuss in more detail in the next example (section 2.4.3). The other transitions are based on ideas similar to those in section 2.4.1: we must move around among possible values for the random variables, and we wish to do so with both large and small steps, but not so large as to make the new value highly unlikely under the model.

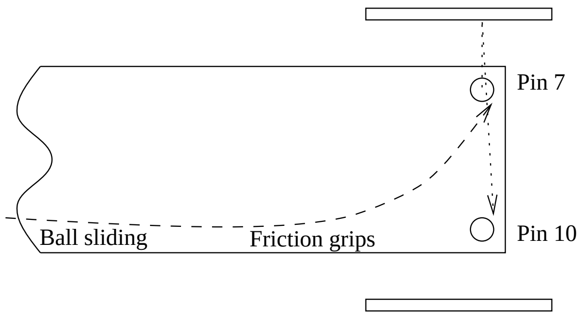


Figure 2.6: *The seven-ten split, in which the aim is to knock down both the seven and ten pins in one shot. The technique used by bowlers relies on the fact that a bowling ball will slide while spinning about an inclined axis, then, at some point, friction will cause the ball to grip, converting the angular momentum of the spin into linear momentum across the lane (dashed line). The seven pin must be struck behind its center of mass, so that it initially moves away from the ten pin (dotted line), bounces off the wall and moves back across the lane to hit the ten pin. Our simulator cannot model friction well enough to simulate this shot. In particular, an efficient, plausible model is required that takes into account both the normal force and the relative tangential velocity at the point of contact.*

Sample Animations

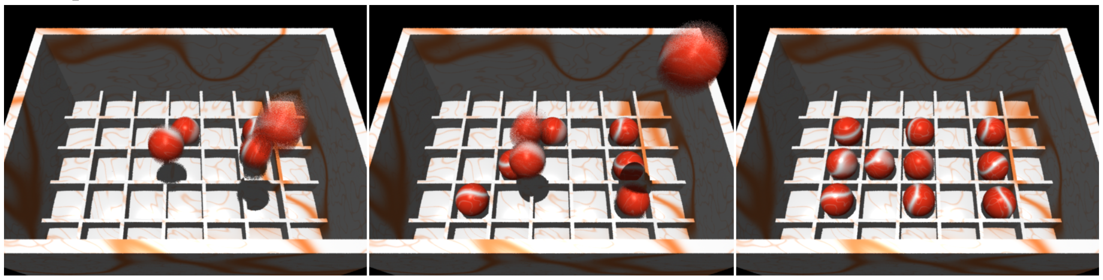
We tested this model with three sets of constraints:

- Bowl a strike.
- Bowl a ball that leaves a six-seven split.
- Bowl the spare that knocks down the six-seven split.

Frames from example animations appear in figure 2.5. The strike example is the easiest, because strikes are quite likely given our simulator. Bowling the six-seven spare is not difficult either, because the various solutions probably form a connected set in state space, so once a single solution is found, the others can be explored efficiently. Bowling the ball that leaves a six-seven split is the hardest example, intuitively because it is hard to knock down the pins behind the six pin while leaving it in place.

We also attempted to bowl the seven-ten split (figure 2.6). This shot depends on the precise frictional properties of the ball and lane. Our simulator's friction model could not capture the required effect, so we could not make the shot. This demonstrates that the MCMC algorithm only generates samples that are plausible according to the model (section 2.3). Our simulation model says that balls never take big hooks, so we never see animations involving big

Example 1 ⟶



Example 2 ⟶

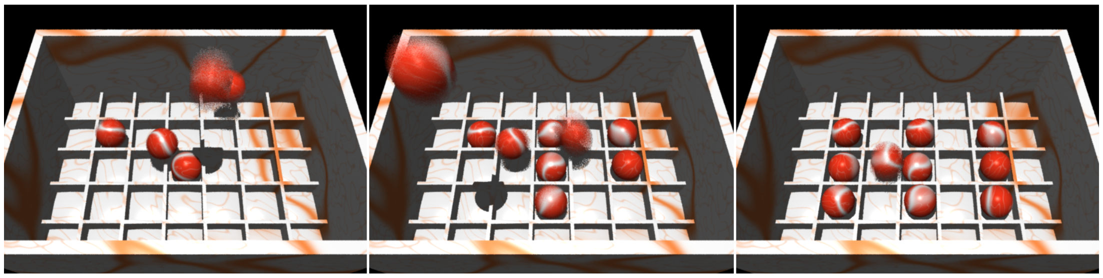


Figure 2.7: *Two examples of the spelling balls model, in this case spelling "HI" in a seven by five grid. The shape of the boxes is allowed to vary slightly, as are the initial conditions of each ball. Our algorithm chooses box shapes and ball initial conditions that lead to the formation of a specific word.*

hooks, regardless of the constraints. The shot could probably accomplished with the existing simulation model by allowing the coefficient of friction to vary over the lane. A lane would be required to be mostly slippery but have a sticky spot near where the ball must hook.

2.4.3 Balls that Spell

In these experiments we drop a stream of balls into a box partitioned into bins so that, when everything has come to rest, the balls form letters or symbols (figure 2.7). We don't care which ball ends up in which designated bin. We use an impulse-based rigid-body simulator, as in the bowling example.

The uncertainty in this world arises from the shape of the partitions and the location from which each ball is dropped. The top surface of the partitions depends on a set of *partition vertices*, each of which is randomly perturbed about a default position. Each ball is dropped from a random location. Details of the geometric layout and probability models are given in appendix A.2.1.

The constraint we impose is that, when all the balls have come to rest, each ball is in a designated bin. We fix the maximum number of balls, so if each ball falls into a designated bin there can be no ball in an undesignated bin. We face a situation in which we cannot propose animations that are certain to completely satisfy the constraints, so, as for the bowling example, we use the Gibbs distribution for the constraint probability $p_c(A) \propto \lambda^k$, where k is the number of balls in designated bins at the end of the animation.

To facilitate mixing we allow the number of balls in the simulation to vary between zero and the minimum number required to form the word, by flipping each ball between active and inactive states: inactive balls do not take part in the simulation. If all the designated bins are filled, removing a ball frees up a bin for another ball to move into, making a significant change to the animation. Removing the ball entirely, rather than just having it go into an undesignated bin, reduces the amount of interaction between the balls, possibly making it easier to make acceptable proposals. It also speeds the simulation when balls that aren't contributing anything are removed. Our initial experiments used a fixed number of balls, and the chain failed to mix well.

The probability of an animation depends on how many balls are participating, the initial locations of the balls and the offsets of each partition vertex. See appendix A.2.1 for details. The use of un-normalized probability density functions with the variable numbers of balls in this example means that we can no longer guarantee that the MCMC algorithm samples according to the correct distribution over the variable number of balls. It does however, still sample from the correct marginal distribution of animations with the full number of correct balls, which is the distribution we are actually concerned with.

Proposals

The proposal algorithm we use performs one of five actions:

- The *change-all* strategy: change all the partition vertices and change all the balls.
- Change a subset of partition vertices.
- Change an active ball.
- Activate some balls (possibly none).
- Deactivate some balls (possibly none).

Details appear in appendix A.2.2.

The change-all strategy appears as a means of escaping from low probability regions (figure 2.9). When an animation is found that satisfies the constraints, subsequent animations tend to also satisfy the constraints, but their probabilities degrade. This occurs because the reduction in probability for a partition vertex change may be quite small, and such proposals are likely to be accepted. The downward trend can continue, moving the chain into a region of low probability. Then, a change-all proposal can reset all the partition vertices to much higher probability values, and even though the constraints are no longer satisfied, the net change in probability will be positive and the proposal will be accepted. This *change-all effect* is good for mixing, because the next fully correct sample will generally be very different from the last.

The second and third proposals are designed to move around the state space by modifying balls or partitions, similar to proposals in previous examples. The proposals to activate or deactivate some balls let us change the number of balls in the simulation. The proposal strategy we use makes the probability of adding or deleting any given ball independent of the maximum number of balls. We first tried a proposal that chose a single ball and flipped its status, but if the maximum number of balls in the scenario is large, the probability of removing a ball goes up as more balls are activated while the probability of adding a ball goes down, making it difficult to get all the balls into the simulation.

The considerations in choosing a value for λ in this example are identical to those in the bowling example (a balance between good animations and good mixing), with an additional requirement due to the change-all effect: the constraint probability should be balanced against the model probability (in this case the probabilities of the partition vertices). If the constraint probability is too high, almost no change in partition vertices can overcome a well satisfied constraint. Good balance is achieved when a much better set of model values can overcome a constraint that is satisfied but uses poor model parameters.

As a specific example, we chose a bin designation that spells “HI” on a seven by five grid (figure 2.7). We used $\lambda = e^5$ for this word. A plot of k, the number of designated bins that are filled, for each iteration of an example chain is shown in figure 2.8. The important feature of this graph is that the chain tends to rapidly reach correct spellings, stays there for a short period, then drops back to incomplete spellings. The twenty thousand iterations shown here took a few hours to compute on a 200MHz Pentium Pro PC.

The change-all effect is evident in this chain. Figure 2.9 plots the probability of the sample for each iteration. Places are marked where there is a sharp reduction in the number of correct balls, and these correspond to sharp increases in probability. At each of these sharp

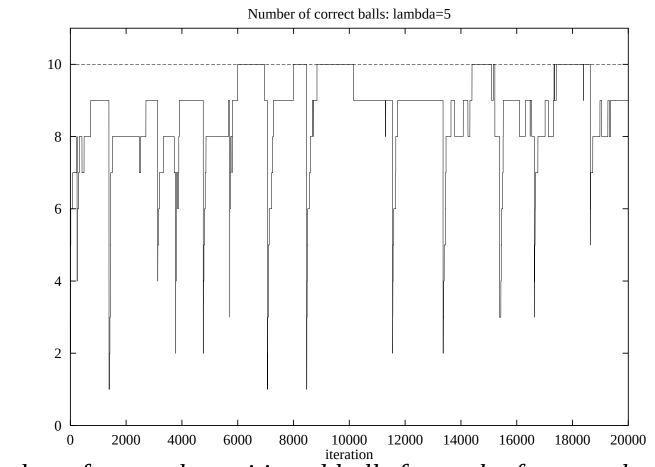


Figure 2.8: *The number of correctly positioned balls for each of twenty thousand iterations of the “HI” model, with $\lambda = e^5$. The maximum number of correct balls is ten. The chain finds its first good animation after around six thousand iterations (we have seen chains that find good animations within one thousand samples). This graph indicates good mixing because the chain spends only a short period of time near similar solutions, then makes significant changes before rapidly moving to a new good solution.*

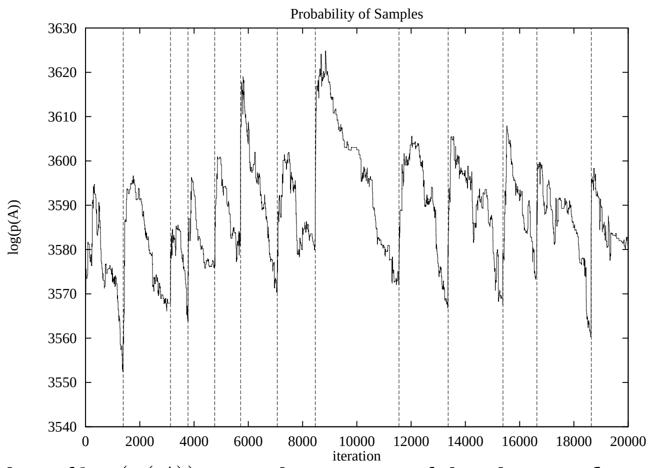


Figure 2.9: *The value of* $\log(p(A))$ *at each iteration of the chain in figure 2.8. The graph is quite bumpy, indicating good mixing. The dashed vertical lines correspond to all the iterations where the number of correct balls drops sharply (figure 2.8), yet all those iterations show a sharp rise in probability. This effect, due to the change-all proposal strategy, is discussed in the text.*

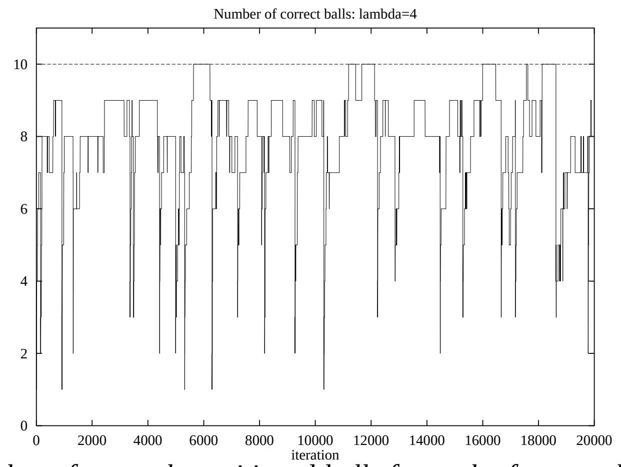


Figure 2.10: *The number of correctly positioned balls for each of twenty thousand iterations of the "HI" model, with $\lambda = e^4$. Compared to the chain with $\lambda = e^5$ in figure 2.8, this chain produces fewer samples with the desired ten correct balls. This is due to the weaker constraint, allowing the more frequent acceptance of samples with reasonable partition vertices but poor ball positions. However, even in this sub-optimal example, performance is not very bad.*

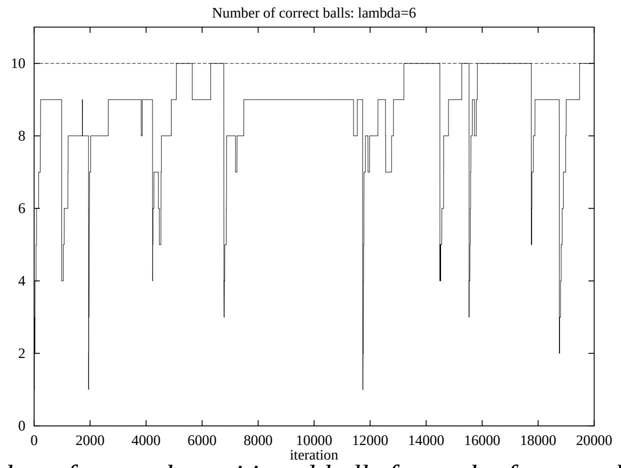
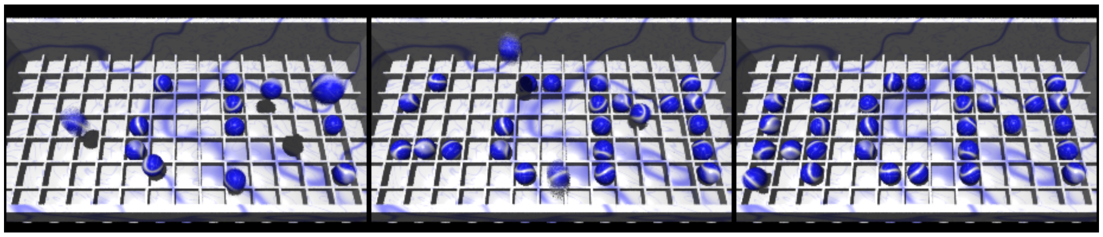


Figure 2.11: *The number of correctly positioned balls for each of twenty thousand iterations of the "HI" model, with $\lambda = e^6$. This chain shows a tendency to get stuck on samples that have many correct balls, resulting in poor mixing when compared to the chain in figure 2.8. Note that the tighter constraint does no guarantee more correct samples. This chain gets stuck for a long period with an almost perfect solution, unable to escape to try something different.*

changes, a change-all proposal has been accepted that replaces a poor set of partition vertex offsets with a much more likely set, even though this breaks the constraint.

We experimented with different values of λ, both higher and lower, but they lead to less satisfactory chains. Values of λ that are too low result in chains that have trouble finding correct animations, because the chance of accepting a poor proposal (from the point of view of

Example 1 $\longrightarrow$



Example 2 $\longrightarrow$

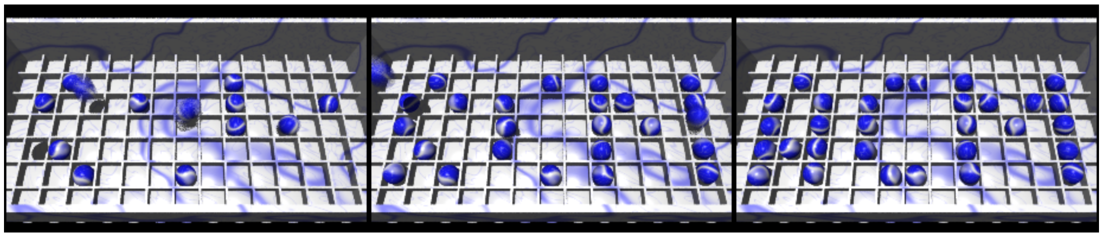


Figure 2.12: *Balls that spell ACM. The box contains 105 bins, of which 30 are designated to contain balls. We show two animations, one on each row, generated from a single chain. Each has the bins being filled in a different order, evidence that the chain produces a good mix of samples.*

the constraints) is too high. Values of λ that are too high make it less likely that a change-all proposal will be accepted, and also make it hard for the chain to abandon poor near-solutions. It takes only a few thousand iterations to see enough of the chain to know how lambda should be changed, and the range of acceptable values is reasonably large so little time is spent in tuning parameters. Our experiments show that chains with $\lambda = e^{5\pm1}$ are not much worse than those for $\lambda = e^5$ (figures 2.10 and 2.11).

We also performed a larger experiment, with 30 of the 105 bins on a fifteen by seven grid to be filled (figure 2.12). In this example we used a value of $\lambda = e^{7.25}$ after experimenting with other values of λ between six and eight. The higher value for λ is required because there are more partition vertices and more balls. The greater number of partition vertices allow the change-all proposal to remain effective at higher λ values, so we still see adequate mixing. In fact, higher λ values are required to make it harder for a change-all proposal to succeed, so that the chain has enough time between major changes to converge to good animations.

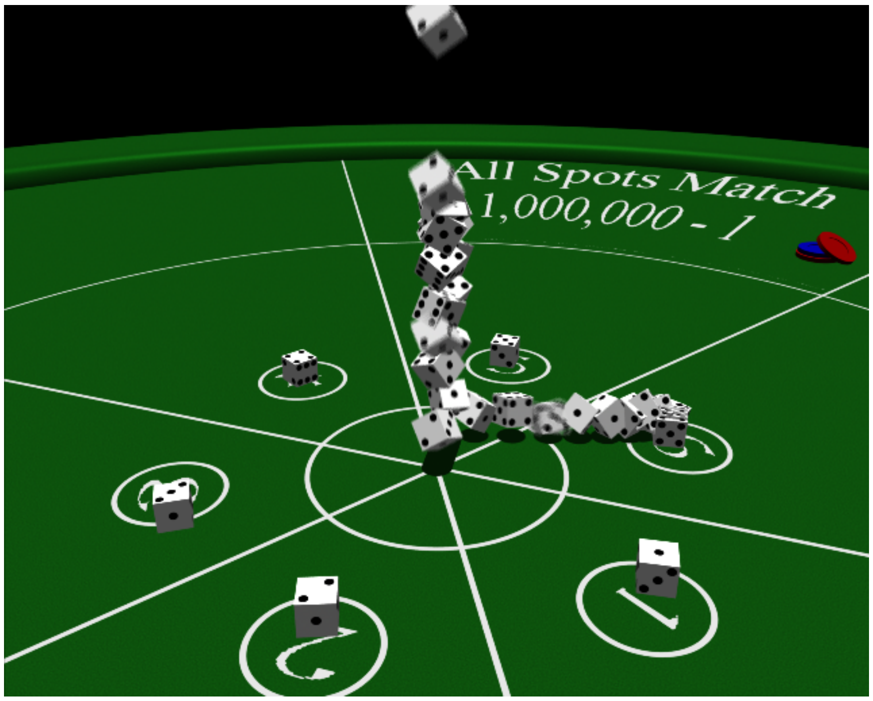


Figure 2.13: *A composite of six sample animations showing the control of a single bouncing die. Each die in the image was animated separately. Each had a different target location and desired side-up, but started with the same distribution on initial conditions.*

2.4.4 Random Tables with Dice

This summarized example demonstrates objects bouncing on a random table, coming to rest in constrained configurations. Dice are used as random number generators in the real world because they are exceptionally hard to control, yet our technique is capable of finding animations in which dice come to rest near a particular place with a particular face showing.

The 2D ball example (section 2.4.1) used a very simple table model, with two main drawbacks due to the use of independent normals at each collision:

- An object bouncing in place will appear to have the table change underneath it as a different normal vector is chosen for each collision.
- Nearby points on the table are not correlated, as points on a real, bumpy table would be, which reduces the plausibility of the animations.

In this example we use a continuous, bumpy surface for the table. Rather than describe random normals directly, we specify a random b-spline surface via control points on a grid with fixed spacing but random vertical offsets. We can also specify random restitution and friction values at the control points, to be interpolated by the spline, thus extending the model to include the concept of springy or sticky regions on the table (such as spilt beer). The b-splines defining the table shape and properties define random fields over the surface. In principle, we could measure real tables, model their particular random fields, and use those in our simulation.

The simulator used in this example simulates only one object at a time bouncing on the random b-spline surface. It uses special techniques to manage the large number of control points required for a table with fine bumps.

In this example, constraints can be defined for any aspect of the object's 3D state at any point in time. Initial conditions for the object are specified by constraining its state at the start of the simulation ($t = 0$). The probability of an animation in this world contains components for the control vertices defining the table's shape, friction and restitution, and a component for each constraint on the object.

An animation generated from this type of scenario is shown in figure 2.13. Each of six dice is dropped and told to land in a specific place showing a specific side up. The dice are treated individually and do not interact — the table is not the same for each die. It took an hour or so of processing time to find a good animation for each die (a few hours for the complete animation). However, the chain does not mix well, so it takes many hours to find significantly different animations. Better mixing might be achieved with annealing [31] or tempering [66, 67] approaches, both of which use a sequence of intermediate distributions to move from a tractable distribution to the distribution of interest. The idea is that mixing in the tractable distribution will carry over to the less tractable distribution of interest.

Proposals in the dice example were made by changing one control point at a time, or one initial condition component at a time, or everything at once, the choice being made according to user supplied relative probabilities. Changes were made by adding a random offset to the current value, resulting in symmetric transition probabilities.

The ability to make changes at any point in the simulation, through the surface control points, makes it easier to find good animations in this world. Control points near the first few collisions get the die somewhere close to the target, and later collisions refine the location. This is not an explicitly coded strategy, rather it emerges naturally from the chain. However, a better proposal strategy might make explicit use of the behavior.

2.5 Future Work

The models we use arise naturally in the real world, and we provide a means of verifying the plausibility of simulations. With further work it should be possible to experimentally obtain more accurate models, and test simulation algorithms on such models, to obtain results like those of Mirtich et. al. [63].

It is an open problem to determine the difficulty of a particular example without experimentation. Computation time can be adversely affected because the simulation itself is slower, or more iterations are required to find good animations, or both. For example, our bowling and spelling ball examples take comparable times to compute, the former due to slow simulation and the latter due to difficult constraints. Simulation time dominates the cost of each iteration, so it is reasonable to spend more time making better proposals to improve mixing and hence reduce the total number of iterations. For example, in the bowling simulation we might bias changes in the ball's initial conditions according to which pins were knocked down.

Constraints in our approach are specified as probability density functions, which allows almost any type of constraint. In particular, it might be possible to constrain collisions or other events to occur at specific times (or frames). This would allow physically-based animations to be choreographed to music, or collisions to occur at frame boundaries.

Popović et. al˙ [73] describe an interactive system for manipulating small numbers of colliding objects. They note that a sampling approach such as the one described here could be used as a preprocessing step within a larger system. Following the generation of a range of initial trajectories using an MCMC approach, users might then be able to interactively improve particular sub-parts of the simulation. This approach would require the interactive solution of fixed two point boundary problems, because the endpoints of the manipulated section must retain continuity within the larger animation. An alternative is a hierarchical constrained sampling approach, where a sub-part of the animation is refined within the whole.

The modeling for the examples in this chapter was all done by hand, particularly the specification of the various random variable distributions. User interfaces could be designed to assist in this task. The interfaces would require techniques for visualizing probability densities defined over multi-dimensional spaces (a volume visualization task). There are also interesting issues in designing randomized geometry, particularly related to ensuring that topological relationships are maintained when, for instance, vertices are allowed to move randomly.

The MCMC algorithm produces many animations for any given goal, and we would like interfaces that allow a user to browse through the results. A Design Galleries [55] type of approach seems the most applicable. Techniques must be developed to index the solutions in a reasonable way. In particular, it should be the case that animations that are visually similar should be grouped together, yet it is not clear how to identify similarly given either a description of how the objects move over time (which is what the simulator produces) or the actual rendered frames of the animation. It may be that the MCMC algorithm itself can provide insight into the problem, by looking at particular transitions in the chain. The spelling balls example supports this idea, because identifying places where the number of correct balls catastrophically drops (figure 2.8) may also indicate the beginning of visually distinct sets of animations.

We have only touched on the possibilities of plausible motion with constraints, focusing entirely on rigid body dynamics. Our techniques may also work in other domains that are hard to constrain, including group behaviors [10] and deformable objects [18]. In the conclusion to this thesis we discuss real-time control within the context of a continuously running virtual environment, which requires solutions to a new set of problems.

Chapter 3

Asynchronous, Adaptive, Rigid-Body Simulation

The MCMC algorithm for rigid-body animations requires the outcome of each candidate to be determined in order to evaluate its probability. Hence, fast simulation is vital to achieve good results in a reasonable time. This chapter describes the new rigid-body simulation algorithm used for many of the experiments in the preceding chapter.

All users of rigid-body simulator desire speed: animators prefer immediate feedback; designers wish to test many possibilities in a short period; and the authors of virtual environments require real-time performance for complex scenes. The algorithm described here is faster than traditional methods for rigid-body simulation and scales better with the number of simulated objects.

Simulation requires an underlying physical model, and we will describe our algorithm assuming the impulse-based model [39, 62]: bodies move with ballistic trajectories until they collide with another object, at which time an instantaneous collision impulse is applied. Only two objects are considered to be colliding at any moment, and transient constraints such as rolling are enforced by large numbers of high-frequency collisions. While this model leaves much to be desired, it is the best existing technique for situations with many transient, short-lived constraints.

The overall trajectory of a single object in an impulse-based rigid-body simulation consists of a sequence of ballistic trajectories separated by collisions. The physical model defines two things: the ballistic equations of motion for the objects and the equations for resolving a collision between two objects. Given this, the outcome of a simulation is determined by the physical properties of the objects in the simulation (geometry, mass, moments etc.) and their initial conditions. The primary purpose of a simulator is to find the correct collision sequence for the simulation, and hence compute the outcome. The design goal of our simulator is to minimize the amount of work done to identify each collision.

Simulators are frequently required to perform a large number of related simulations, such as the sample chain in a MCMC algorithm, or multiple simulations to numerically estimate some derivative. In such cases, it is highly desirable to exploit the results of previous simulations in computing a new simulation. For example, the parameters to the new simulation may differ only slightly from those of the previous one (as is the case with MCMC), and hence large portions of the simulation may be unchanged from one iteration to the next. Considerable speedups may be possible if these unchanged regions can be identified and re-used in the next iteration. While not an explicit design goal for the algorithm described here, the issue is discussed at the end of the chapter.

3.1 Existing Algorithms

Existing rigid-body simulation algorithms differ in their physical models, but all share a common need to identify a sequence of collisions. The majority of published algorithms have the following generic structure:

Algorithm 3.1 Generic rigid-body simulation algorithm

```
genericSimulate(objects: $O$)
    while $simulating$
        for all $o \in O$
            step($o$)
        $C \leftarrow$ potentialCollisions($O$)
        for all $c \in C$
            if colliding($c$)
                processCollision($c$)
```

Detailed descriptions of variations on this approach have appeared in [39, 40, 45, 35, 60] and the algorithm is implicitly assumed for many collision detection papers [19, 64, 88, 91, 93]. There are three basic optimizations described in the literature:

- Dynamically choose the time-step by attempting to predict when the next collision might occur [40, 45, 62].
- Identify potential collisions without checking every pair of objects. Several methods exist: Cohen et. al. use sorted lists of bounding volumes [19]; Mirtich uses spatial hashing of bounding volumes [60]; and several authors have used octrees or similar structures [88, 91, 93].
- Improve the speed of collision detection (see [51] for a survey).

The framework above imposes one significant constraint: all the objects are updated synchronously, and all the data structures to identify potential collisions must also be updated. Much of this work is wasted, because sufficient information exists from previous steps to know that many objects cannot collide — there is no need to compute their state again. Some schemes modify the global time-step [40, 45, 60], using predictions based on the current simulation state (such as velocities and object separations), but they do not modify the integration time-step for individual pairs. Erickson et. al.[26] also describe a predictive method, but only for a single pair of objects.

Kim et. al. [50] describe an event-driven algorithm for the simulation of ballistic spheres. In their asynchronous algorithm, collision times for all potential collisions are placed in an event queue. As the head of the queue is processed, only the two spheres involved in the collision are updated. To avoid unnecessary predictions, spheres are associated with cells in a uniform spatial subdivision, and only spheres that share a cell must be checked for collisions. To maintain the spatial data structure, additional events are used to track when spheres enter or leave a cell.

In order to be efficient, the number of boundary crossing events must be kept low, and the number of spheres in each cell must be controlled. Kim et. al. use techniques from molecular gas theory to choose a subdivision size based on the size of the objects and their expected density. However, in practical simulations the density and velocities of objects may vary significantly over time or space, a situation for which the gas theory model is poorly suited. A subdivision poorly tuned to the instantaneous distribution of objects will lead to large number of unnecessary events, and a loss of efficiency as the algorithm fails to adapt.

Some physical models, and hence algorithms, allow multiple interacting objects at the same moment in time [4, 5]. They still conform to the above framework, but with the definition of a collision generalized to any change in the topology of object interactions (who interacts with who and in what way). Our algorithm may be modified to handle such models.

This chapter describes an asynchronous event-driven algorithm that adapts to local changes in object behavior, both spatially and temporally. We begin with an overview, then describe the algorithm in detail and give experimental results demonstrating its performance. We conclude with some possible extensions and future research directions.

3.2 Adaptive, Asynchronous Simulation

Our algorithm is event-driven: it maintains a queue of interesting future events, sorted according to when they occur, and repeatedly pops the head of the queue and processes it by changing object state and adding or deleting events. The algorithm is efficient because we use a prediction scheme to schedule events only when something might happen, and the processing for each event changes only those objects affected by the event, rather than all the objects in the simulation.

As noted in [50], sweep algorithms in computational geometry behave in an identical way: maintain data structures and schedule events when they change [23].

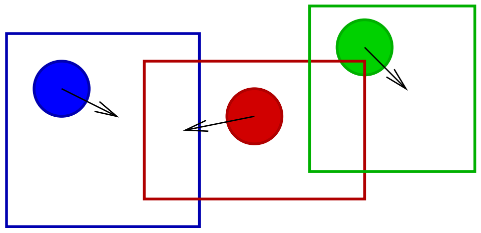
3.2.1 Prediction Scheme

The aim of a prediction scheme is to identify future times at which objects might collide. The basic component is a procedure, $\mathsf{predict}(o_1, o_2)$, that returns a conservative[1] estimate of the time (possibly infinite) at which two objects will collide on their current trajectories. We describe our version of the $\mathsf{predict}(o_1, o_2)$ procedure below, but for the moment assume it exists.

Note that a prediction remains valid until something happens to change the expected state of the predicted objects, regardless of *when* the prediction was made and regardless of which *other* objects are changed. In other words, predictions need not be updated until one of the objects involved in the prediction undergoes a collision.

It is clearly wasteful to make and maintain predictions for all $O(n^2)$ object pairs in the simulation, so a method is required to rapidly decide which pairs cannot collide any time

[1]A conservative estimate is one that is certain to be earlier than or equal to the actual collision time



Events:
Green bound expires at $t = 1.0s$
Blue-Red collide at $t = 2.0s$
Blue bound expires at $t = 3.0s$
Red bound expires at $t = 4.0s$
Red-Green collide at $t = \infty$

Figure 3.1: *A momentary snapshot of an example 2D simulation involving three balls. Each ball is shown in a different color with its temporal bounding volume shown in the same color. Arrows indicate the direction of travel. On the right are the events currently in the queue for this simulation. Note that there is an event for the expiration time of each bounding volume and a collision event for each pair of objects with overlapping bounds. We would like to choose bound sizes such that each bound expires infrequently yet few bounds intersect, as that will result in the fastest algorithm (with the fewest events to process).*

soon. Following [60], we define the *temporal bounding volume* for an object and an interval to be an axis-aligned box that is guaranteed to completely contain the object for the duration of the interval, assuming nothing happens to change the trajectory of the object. We say that the bound *covers* the interval, it *expires* at the end of the interval and that it is *invalidated* if the object is involved in a collision (which changes the object's trajectory).

Various data structures exist to store axis-aligned boxes and identify which boxes intersect [19, 60, 88, 91]. We use one of them [19], described below, to identify intersecting pairs of temporal bounding volumes. We make predictions for the objects whose bounds intersect, and do nothing for those that don't (figure 3.1).

Observe that at any moment it is permissible for the bounds of different objects to cover different intervals, provided that each bound is valid at that moment. For example, if the bound for one object covers the interval $I_a = [t_{a1}, t_{a2}]$ and the bound for another covers $I_b = [t_{b1}, t_{b2}]$ we will correctly identify a potential collision at time t if $t \in (I_a \cap I_b)$, regardless of the actual endpoints of the intervals. This frees us to adapt the bound size for different objects according to how we expect them to behave: objects with sparse interactions can have large bounds, and objects with dense interactions can have smaller bounds.

To ensure that the bound for a given object remains valid, we schedule an event each time the bound expires. When the event reaches the head of the queue, the bound is updated to cover a new interval. We select the new interval according to the time since the

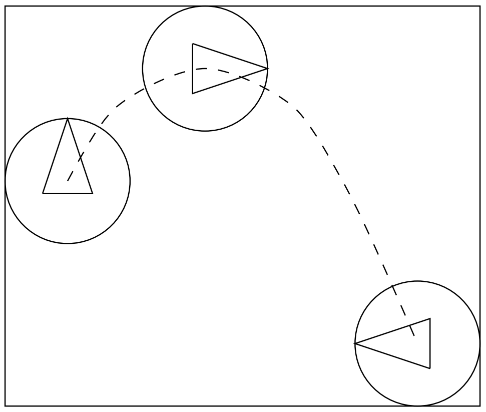


Figure 3.2: *A 2d temporal bounding volume, of the form described by Mirtich [60]. The object's bounding sphere, and its location at the beginning, end and possibly apex of the parabolic trajectory are used to define the box.*

object's most recent event: the length of the new interval is twice the elapsed time from the previous event. With this heuristic, objects experiencing frequent collisions will have relatively small bounds that cover small intervals, whereas objects undergoing no collisions will see their bounds grow exponentially until a collision is predicted. It is in this way that the algorithm adapts to changing object behavior.

Bounds are also recomputed when an object experiences a collision. The new bound interval is calculated using the same heuristic.

We now describe the algorithm in detail, starting with assumed procedures, describing our data structures, then discussing the processing of events.

3.2.2 Basic Procedures

We assume the existence of several basic procedures that operate on one or two objects:

integrate(o, t)**:** Given an object and a time, integrate the equations of motion for the object to find it's state at that time. We use a variable step-length Runge-Kutta method described in [75].

generateBound$(o, t1, t2)$**:** Given an object and the endpoints of an interval (computed with the heuristic in section 3.2.1 above), generate an axis aligned bounding volume that is guaranteed to contain the object throughout the interval. We use the method described in [60] for ballistic objects, which bounds the motion of a bounding sphere centered at the center of mass for the object (figure 3.2).

predict($o1, o2$): Given two objects, make a conservative prediction of their collision time, and insert an event into the queue for that time. Our prediction algorithm uses one of two estimates: if the objects' bounding spheres do not intersect at the time the prediction is made, we predict the objects will collide when their bounding spheres collide[2], otherwise we use the predictor in [60], which uses bounds on maximum angular velocity and hence maximum normal velocity. We use a two stage predictor because the prediction based on the maximum normal velocity is extremely pessimistic when the objects are a significant distance apart.

closestPoints($o1, o2$): Given two objects, find their closest points, which can be used to determine the distance separating them and the normal of a separating plane. The distance and normal information is used in the prediction phase, in addition to providing data for collision resolution. We have a strong preference for closest feature algorithms, because they allow for more robust separating plane computations as objects approach very closely. We use a modified version of the VClip algorithm [61]. The colliding($o1, o2$) predicate finds the closest points and tests the distance between them to see if they are colliding.

resolveCollision($o1, o2$): Given two objects and their closest points, modify their state to reflect the occurrence of a collision. We use Euler's method, as presented in [39], but other methods may be used [5, 15, 60].

3.2.3 Data Structures

The algorithm maintains data structures for each object's state, for scheduling events, and for tracking intersecting temporal bounding volumes. Each object stores its physical properties (geometry, coefficient of friction etc.), its dynamic state (position, orientation, velocity etc.), a temporal bounding volume, and pointers to various events for the object. Events are stored in a priority queue that supports event insertion, min-entry deletion and arbitrary deletion. A change-key operation is also useful. Events themselves store their time (the priority search key) and information about the objects involved.

[2]The bounding spheres are distinct from the temporal bounding volumes. They are defined by the object's center

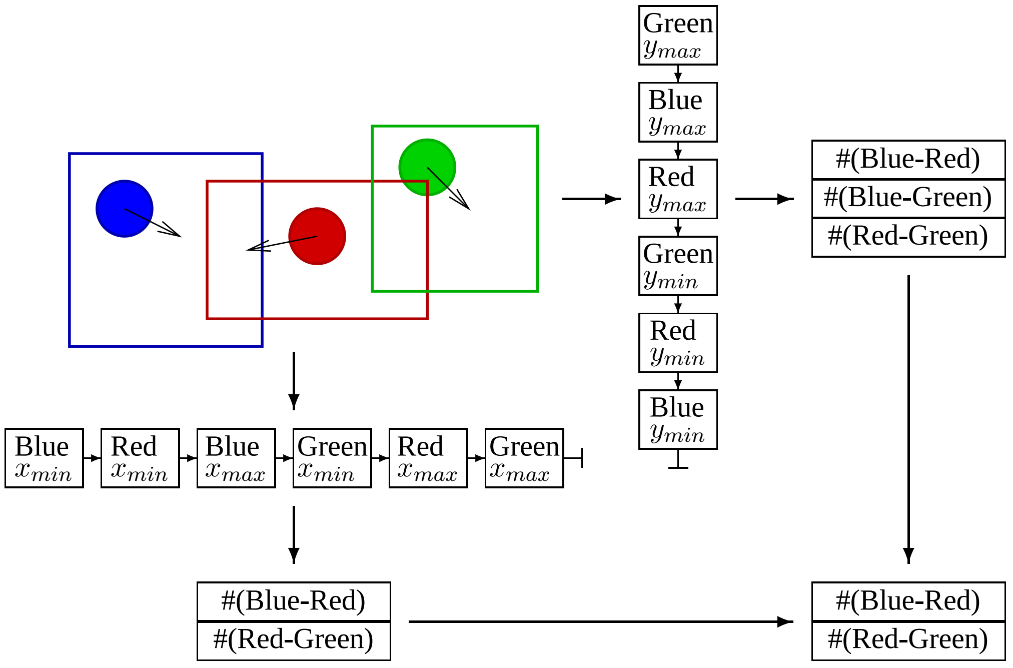


Figure 3.3: *The data structures used to detect bounding volume intersections, presented here for the 2D example of figure 3.1. The minimum and maximum extents in each dimension of all the bounding volumes are stored in sorted lists. Any overlaps in a single dimension are stored in a hash table for that dimension. Single dimension overlaps occur whenever either extent of one bound appears in the sorted list between the extents of another bound. In this example, one such overlap is between the blue and red bounds in the x dimension, because the minimum extent for the red bound lies between the extents of the blue bound. Overlaps that appear in all dimensions, in this case the blue-red and red-green overlaps, are stored in another hash table. Not shown are various pointers between bounds, hash table entries and objects.*

The data structures for storing temporal bounding volumes are designed to support a linear sort dimension reduction approach to finding intersecting boxes [19]. We maintain three sorted lists, one for each dimension (figure 3.3). Each list contains both the minimum and maximum extents in that dimension of every bounding volume, with pointers back to the bound object. In addition, we maintain four hash tables, one for each linear dimension and 3d space. Each hash table stores the intersections in that dimension, or 3d intersections. Finally, each object keeps a list of the other objects whose bounds its bound intersects in 3d. Together, we refer to these data structures as the *intersection data structures*.

The intersection data structures must be updated each time a temporal bounding volume changes. We describe the update procedure below, in the context of event processing.

3.2.4 Event Processing

Two types of events are required for the basic algorithm:

- An update event is scheduled for the time that an object's temporal bounding volume expires. A new bound is computed and the intersection data structures are updated.
- A collision event is scheduled for times when two objects might collide. An invariant is maintained that whenever the bounds for two objects overlap, and they are predicted to collide in finite time, there is an event in the queue for the predicted time.

As stated above, events are stored in a priority queue. The outer loop of the algorithm simply deletes the minimum-time event in the queue and processes it as follows.

Update event processing

An update event (figure 3.4) indicates that a temporal bounding volume has expired for some object, and hence a new bound must be computed, which is done using the generateBound procedure with a new interval of length twice that of the time since the last event for the object (as described in section 3.2.1).

Given the new bound, the intersection data structures must be updated, as depicted in figure 3.5. We begin by re-sorting the lists of extents in each dimension. Only one bound has changed, so we can re-sort by starting with the bound's old location in the list and searching

of mass and the point on the object furthest from the center.

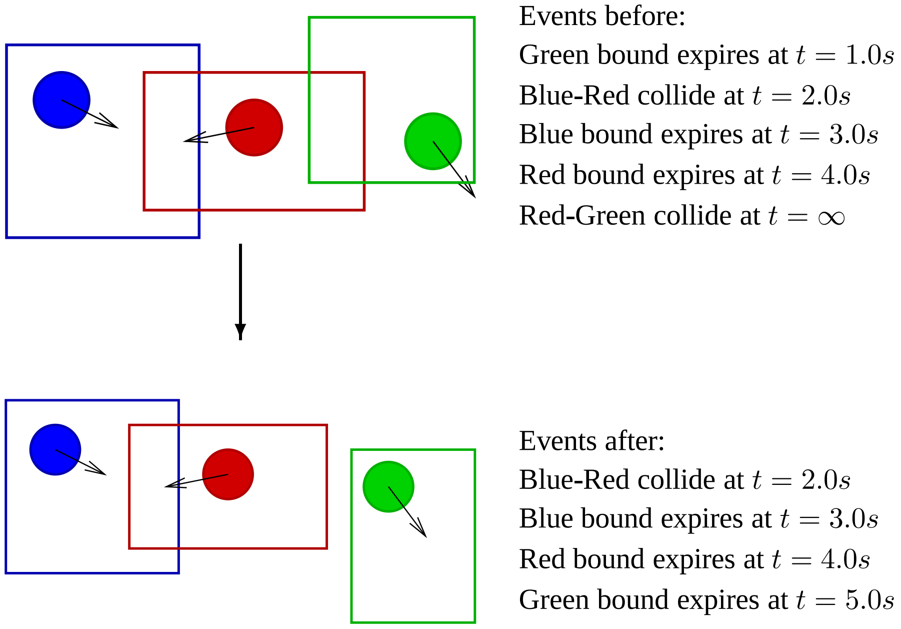

Events before:
Green bound expires at $t = 1.0s$
Blue-Red collide at $t = 2.0s$
Blue bound expires at $t = 3.0s$
Red bound expires at $t = 4.0s$
Red-Green collide at $t = \infty$
Events after:
Blue-Red collide at $t = 2.0s$
Blue bound expires at $t = 3.0s$
Red bound expires at $t = 4.0s$
Green bound expires at $t = 5.0s$

Figure 3.4: *Processing for an update event. In this 2D case, the green object's bound has expired. A new bound is generated, which means that the green bound no longer intersects that of the red object. The intersection data structures must updated to reflect the change (figure 3.5). The event queue also changes to reflect the new bound's expiration time.*

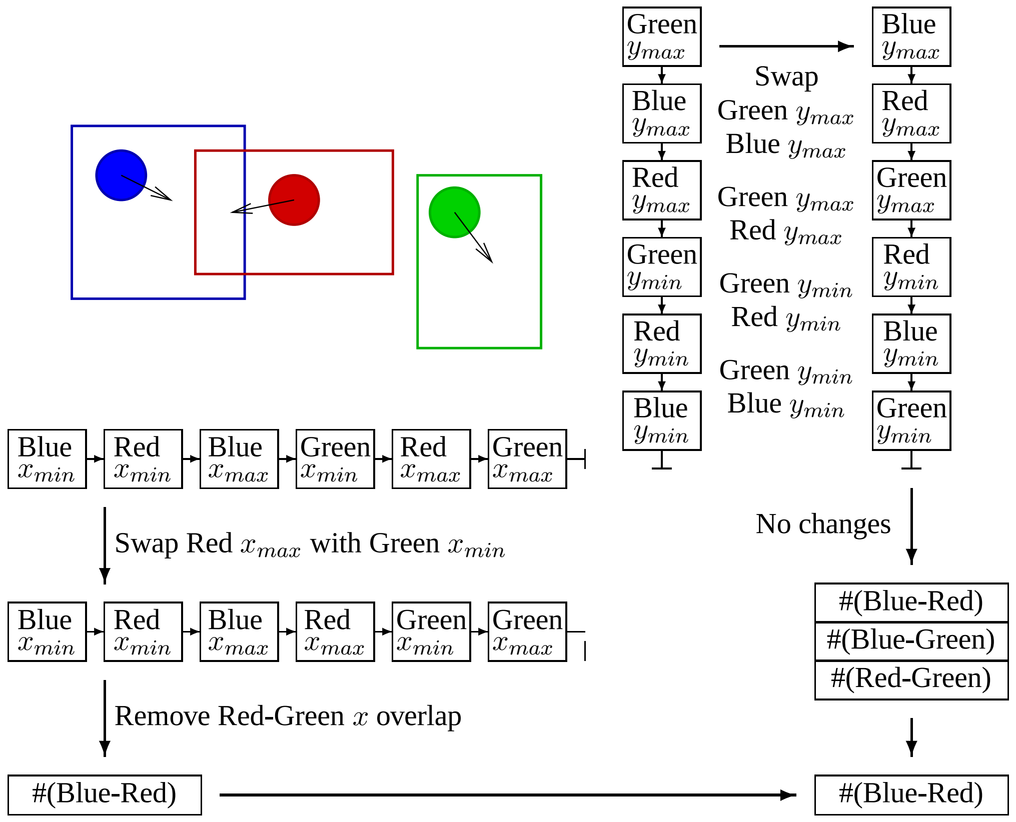


Figure 3.5: *Updating the bounding volume intersection data structures. The green bound has just changed. For each dimension, the list of extents is resorted by shifting the changed extents (the green ones) within the list. Each swap is tested to see whether it changes the overlap status in that dimension. Swaps of minimum extents past maximums, or vice versa, lead to changes in overlaps, but swaps between minimum and minimum or maximum and maximum do not. The changes in status are reflecting in the hash tables, and for each change in a given dimension the other dimensions are checked to see if the global intersection status should be changed. In this case, the removal of the red-green overlap in the x dimension means that the bounds no longer overlap in all dimensions and the global intersection should be removed. Note that in the y dimension, all the swaps are between minimum and minimum or maximum and maximum extents, so there is no change in overlap status in that dimension.*

forward or back to find its new location. As we search, we build lists of potential changes in intersection status, as indicated by changes in the sort order. After re-sorting the linear lists, we examine the changes in one-dimensional intersections that occurred and map these onto changes in three-dimensional intersections. Each potential change can be incorporated in constant time by looking for intersections in the various hash tables. If a new three-dimensional intersection arises, we make a prediction for the collision time for the relevant objects and insert an event into the queue if required. The entire process takes time linear in the number of position changes that occur in the one-dimensional lists. Note in particular that intersections that are unchanged by the new bound are not considered. This is desirable, because nothing about any object's state has changed to invalidate a prediction.

The update event processing is summarized as follows:

Algorithm 3.2 Update event processing

```
processUpdate(simulation: $S$, object: $o$, time: $t$)
    integrate($o, t$)
    $t\_new \leftarrow 2 * (t - o.t\_last)$
    generateBound($S, o, t, t\_new$)
    updateIntersections($S, o$)
    $o.t\_last = t$
```

Collision event processing

A collision event indicates that two objects *might* be colliding (with an exact prediction scheme we would know the objects are colliding, such as in [50]). The first task is to check whether or not the collision is actually occurring, which requires integrating the two objects to the event time and performing a collision test. If the objects are not colliding, we simply make a new prediction for these two objects and insert it back into the event queue. Nothing more needs to be done, because nothing about the objects' expected behavior has changed, nor have their bounding volumes changed.

When the objects are colliding (figure 3.6), the collision is resolved, which changes the trajectories of the objects involved, so bounding volumes and existing predictions must be updated for both objects. A new bound is computed and the intersection data structures are updated as described above, except that predictions for newly generated intersections are delayed, because they will be made by the next step in the procedure. Some care must be taken when updating the data structures for the first object, because the second is not in its correct sorted position. As a final step, we consider all of the objects whose bounds intersect that of a colliding object, and make new predictions for the pairs. The cost of this phase is linear in the number of intersections for the objects.

In pseudocode:

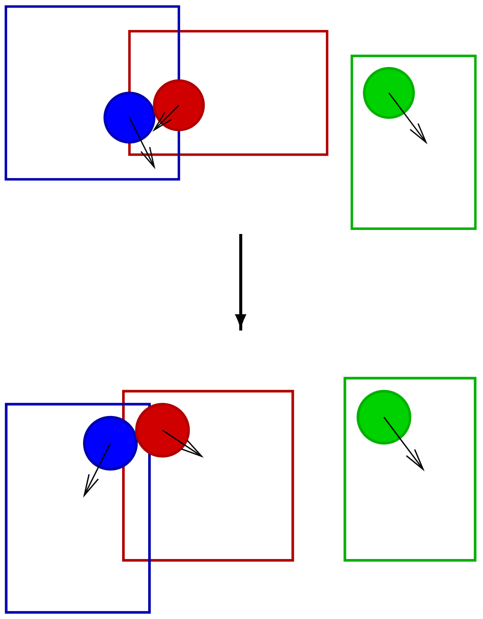
Algorithm 3.3 Collision event processing

```
processCollision(simulation: $S$, object: $o_1$, object: $o_2$, time: $t$)
    integrate($o_1, t$)
    integrate($o_2, t$)
    if colliding($o_1, o_2$) then
        resolveCollision($o_1, o_2$)
        $t\_new \leftarrow 2 * (t - o_1.t\_last)$
        generateBound($o_1, t, t\_new$)
        updateIntersections($o_1, false$)
        $t\_new \leftarrow 2 * (t - o_2.t\_last)$
        generateBound($o_2, t, t\_new$)
        updateIntersections($o_2, false$)
        for all $o$ s.t. $o.bound \cap o_1.bound$
            predict($o_1, o, t$)
        for all $o$ s.t. $o.bound \cap o_2.bound$
            predict($o_2, o, t$)
        $o_1.t\_last = t$
        $o_2.t\_last = t$
    else
        predict($o_1, o_2, t$)
```

Additional Events

Additional events are required to introduce new objects into the simulation, and to obtain trajectory information from the simulation. A start event is scheduled for every object for the time at which the object should enter the simulation. Start events are processed by inserting the extents of the object's bound into the intersection data structures, then immediately calling processUpdate with the object, giving it a last event time in the recent past.

It is relatively easy to extract trajectory information at arbitrary points in time if the collision sequence is known. The simulator can dump such information as part of the collision event processing, but the amount of information generated may be unwieldy. An alternative



Events before:
Blue-Red collide at $t = 2.0s$
Blue bound expires at $t = 3.0s$
Red bound expires at $t = 4.0s$
Green bound expires at $t = 5.0s$

Events after:
Green bound expires at $t = 5.0s$
Red bound expires at $t = 6.0s$
Blue bound expires at $t = 6.2s$
Blue-Red collide at $t = \infty$

Figure 3.6: *Processing for a collision event. In this 2D case, the red object is colliding with the blue object. The collision is resolved, which changes the velocity of the two colliding objects. The change in velocity means that new bounds must be computed for the blue and red objects, and the bounding volume intersection data structures must be updated. Note that the green object is unaffected by the collision, so its state is not changed in any way. The event queue changes to reflect the new bound expiration times and the new predicted collision time for the red and blue objects.*

is to sample the simulation state at various times, using special sampling events. Our current implementation can insert events at a constant rate to gather data for rendering. Other schemes could adapt the sampling rate depending on, for instance, how fast objects are moving, or even whether or not the objects are in view at any given time.

If desired, other events could be inserted for instantaneous control actions or user interactions.

3.2.5 Analysis

Describing the complexity of a simulation, and hence of algorithms that compute it, remains an open problem. Clearly there are several components to the inherent cost of a simulation: the geometry of the objects determines the worst case complexity of collision tests; how the objects move with respect to each other influences the cost of collision detection; the shape of the objects and their configuration influences prediction schemes; the global configuration determines the cost of broad phase collision detection; and so on. Recent work by Erickson et. al. [26] has provided the beginnings of a theoretical basis, by examining the complexity of collision detection as a function of the algebraic complexity of the objects' trajectories, yet this says nothing about the effects of the global configuration, nor is it clear that their algorithm, while open to analysis, is faster than other techniques in practice. There are also difficulties in analyzing how various modules (such as prediction schemes and collision detection schemes) interact. For instance, a higher predicted collision frequency probably makes individual collision tests faster.

Instead, we pursue a qualitative analysis, looking at the expected computational savings of our approach when compared to existing algorithms that seek to achieve the same thing: locate all the collisions in the simulation. We identify one event processing step in our algorithm with one global step in traditional algorithms, referring to both simply as events. Collision prediction technology can be applied equally well in most algorithms, so we can assume that the total number of pairwise collision events in a given simulation is likely to be similar across most algorithms. Our algorithm processes additional update events to maintain the temporal bounds, but our heuristic for computing expiration times tends to keep the total number of updates low with respect to the number of collisions. Update-like events may be required in existing algorithms, when there are no intersections between the bounding volumes and the algorithm must take an arbitrary step. Overall, we assume that our algorithm processes only a small percentage more events than traditional approaches.

The priority queue operations required to maintain the event heap theoretically cost $O(\log n)$ per event, but in practice the constants are so small compared to the cost of other computations that a single closest point computation is more expensive for any practical number of objects. We will ignore the priority queue cost.

We break the other costs of processing each event into four components:

Integration of rigid-body motion is an unavoidable cost regardless of the algorithm used: every object must be integrated for the duration of the simulation. Variable step-length integrators are designed to be most efficient when called for large total steps (even if they internally break the step down). Previous approaches call the integrator for every object at every event with the small global event interval, whereas our algorithm calls the integrator for fewer objects with their own longer event interval. Apart from the reduced overhead of integrator calls, some savings are possible due to the bigger steps, so we can expect a small reduction in the total integration cost.

Collision resolution is an unavoidable cost in both simulations, and if all algorithms process the same collisions, the cost is identical.

Closest Points computation costs are proportional to the number of predictions made (which includes the number of collision tests conducted, because there is one prediction corresponding to every test.) The cost is related to the number of bounding volume intersections that occur, as well as many other factors including the trajectories of the objects in the simulation. Our algorithm appears to make more predictions because the bounding volumes tend to be larger, but the effect is balanced by the fact that each prediction provides some information, which can reduce the number of future predictions. Experimental evidence suggests that the overall effect on computation time is small compared to other factors.

Bounding volumes change in existing algorithms for every object at every event, unlike our algorithm which changes at most two objects' bounds at any step. This is the most significant advantage of our algorithm. If the number of intersection changes in each dimension is a constant, we can perform this step in constant time, unlike other approaches which must always perform $O(n)$ operations. We do expect the number of intersection changes for each event to be small in many situations, and it cannot be worse than $O(n)$, so we must obtain some speedup.

There is a common event that our algorithm is guaranteed to process in constant time. When a conservative prediction is incorrect (the objects are not colliding), our algorithm performs a constant amount of work: two integrations, one collision check, one prediction and one event insertion (which we assume to have negligible cost compared to the other operations). Existing algorithms perform $O(n)$ operations for this case, because they integrate and update bounding volumes for every object. We achieve significant speedups by making this common case fast.

In summary, if the number of bounding volume intersections for a given object is roughly constant, we can potentially process each event in constant time (assuming the cost of integration is roughly constant for each event). Existing techniques must perform at least $O(n)$ computations, so the potential exists for an $O(n)$ speedup with our algorithm. The exact conditions under which this is achievable are difficult to fully enumerate, but simulations in which objects are relatively sparse and uniformly distributed in space are certainly one case. Importantly, our algorithm cannot perform worse than existing techniques, even in bad cases.

3.3 Experiments

We conducted two experiments to compare our algorithm with traditional approaches. One examined a scenario in which polyhedral approximations to a sphere move in a box, without energy loss or the influence of gravity. This is a common example in the literature [19, 50]. As a second example we simulated a stream of objects (sand) falling through a funnel (hourglass). We consider this to be more typical of real-world applications, because objects experience different conditions at different points in their trajectory, and the example demonstrates clustering as objects come to rest on the floor. This example is coincidentally similar to those in [93] and [88].

The traditional algorithm used was:

Algorithm 3.4 Traditional simulation algorithm used for experiments

```
traditionalSim(objects: $O$)
    **repeat**
        $e \leftarrow$ queueDeleteMin()
        **for all** $o \in O$
            integrate($o, e.t$)
        processEvent($e$)
        $t\_next \leftarrow$ queueMinTime()
        **for all** $o \in O$
            generateBound($o, e.t, t\_next$)
        updateIntersections()
```

Procedures integrate($o, e.t$) and generateBound($o, e.t, t$) are as described above. When this algorithm sees a collision event (the only type it sees), it checks for collision, and then either resolves the collision or makes a new prediction for the pair involved. updateIntersections() in this algorithm must do a complete re-sort of the extent lists in each dimension [19], and always makes predictions when new intersections are found. We also use event hysteresis: events are not deleted from the queue when an intersection ceases to exist, which improves the performance of the algorithm. Note that we do not update all predictions at every step, only those that may have potentially changed. We used the same sub-algorithms (closest points, prediction etc) in this traditional algorithm as in the implementation of our approach.

In comparing two algorithms for rigid body simulation, it is important to consider the general instability of simulation. In this particular instance, the algorithms use different bounding volumes, and hence will not make predictions at the same time. The inexact predictions produce small changes in collision time and state, and the outcome of the collision — changes that propagate until the simulation can become quite different. This is particularly true if there is nothing in the simulation that tends to force a certain outcome, as is the case in our examples. The instability of simulation algorithms has a significant impact on verification of their quality and how they should be used, but for the purposes of comparison we minimized the effects of instability by, for each data point, running multiple simulations with different initial conditions and averaging the results.

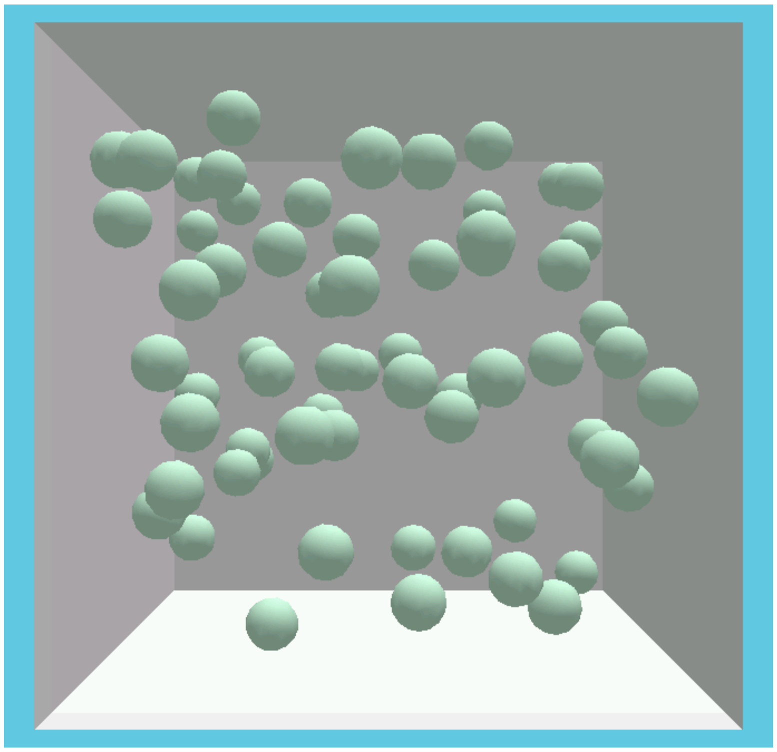


Figure 3.7: *A snapshot of a simulation of polyhedra in a box. One wall of the box is not rendered to allow a view inside, and the polyhedra are shaded to look like the spheres they approximate. There are 64 objects in this simulation. Note the uniformity of the spatial distribution of objects.*

3.3.1 Boxed polyhedra

This experiment simulated a variable number of identical polyhedra, each approximating a sphere, moving inside a box (figure 3.7). Gravity was turned off, and there was zero friction and no energy loss at each collision. The box was slightly larger than ten object diameters in each dimension. Each object was given a random initial position, orientation and velocity, and zero initial angular velocity. Each run simulated 1000 seconds of virtual time, and we tracked the number of events processed by each algorithm, and the total execution time. We conducted five runs for each data-point, each with a different pseudorandom seed. The average results are presented in table 3.1 and the speedups obtained by our algorithm are plotted in figure 3.8.

The results confirm our analysis in section 3.2.5. Our algorithm processes more events than the synchronous approach, due to update events. However, the processing time

n	Synchronous			Asynchronous		
	#events	t/event (ms)	t (s)	#events	t/event (ms)	t (s)
1	48192	0.084246	4.0600	45743	0.053079	2.428
2	119905	0.10013	12.006	119594	0.053598	6.410
3	163444	0.11469	18.746	165851	0.054832	9.094
4	236492	0.13033	30.822	233329	0.056461	13.174
6	378542	0.16059	60.792	388171	0.058206	22.594
8	508000	0.18938	96.204	509753	0.060057	30.614
11	741250	0.23281	172.57	773658	0.062345	48.234
16	1204244	0.30545	367.84	1241069	0.065629	81.450
23	1867831	0.40918	764.28	1974795	0.068919	136.10
32	3124585	0.54400	1699.8	3256984	0.072726	236.87
45	5162582	0.73846	3812.4	5411264	0.076436	413.62
64	8999377	1.0242	9217.3	9664098	0.078830	761.82

Table 3.1: *Results for the simulation of polyhedra moving inside a box, without gravity or loss of energy in collisions. The number of events, time per event, and total simulation time for our asynchronous algorithm are compared to data for a traditional synchronous algorithm for a range of n, the number of moving objects in the simulation. Our algorithm processes more events due to* update *events, but the cost of each event is lower, resulting in an overall speedup approximately linear in n. Times were obtained on a 200MHz Pentium Pro.*

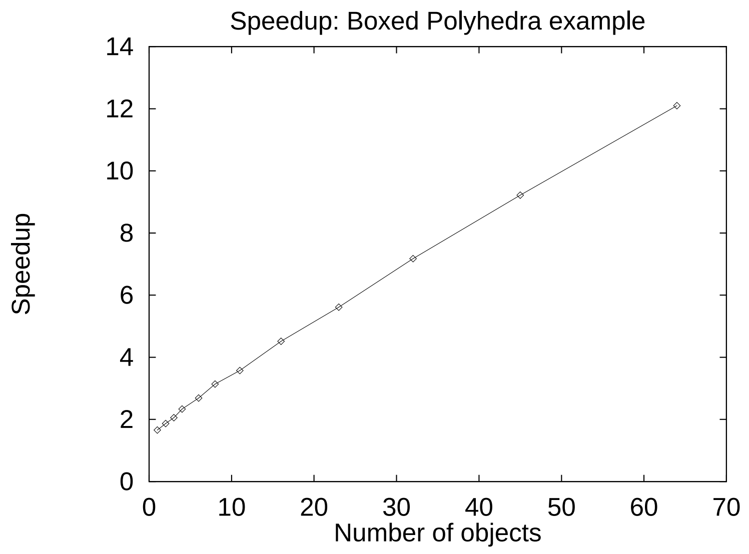


Figure 3.8: *The speedups obtained by the algorithm described in this chapter over a traditional approach (algorithm 3.4) for the simulation of spherical polyhedra in a box. Our algorithm demonstrates a speedup approximately linear in the number of moving objects. We stopped gathering data at 64 objects due to the prohibitive run-times for the synchronous algorithm.*

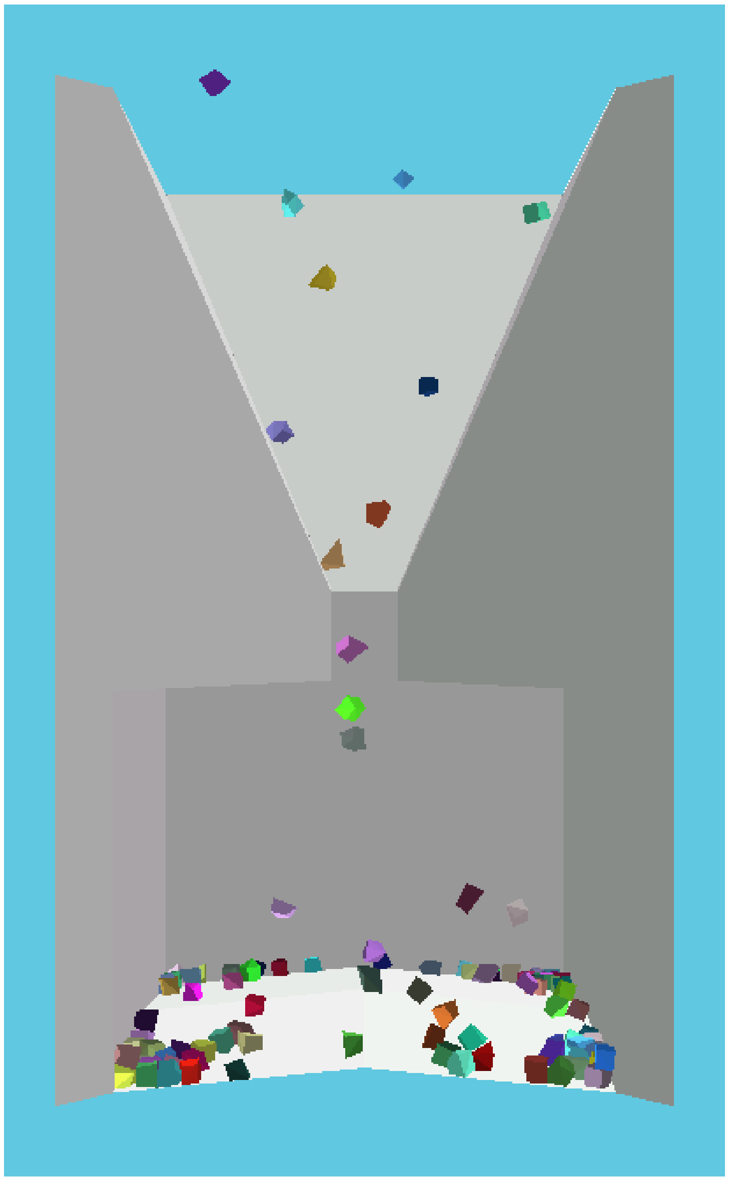


Figure 3.9: *A snapshot of a simulation of grains falling through a funnel. One wall of the box and funnel is not rendered to allow a view inside. There are 100 objects in this simulation. The objects undergo sparse interactions at the top of the funnel, but very dense interactions at the base, as they come to rest in clumps. There is major clustering in the corners of the box at the bottom, which results in the extents of the bounding volumes for each object lying very close together in each spatial dimension. This leads to poor performance of the bounding volume intersection algorithm that was used in the simulator implementation.*

for each event is significantly lower, resulting in speedups that are nearly linear in the number of objects in the simulation. There is some increase in processing time per event as the number of objects increases, due to the increased density of the simulation which generates more bounding volume intersections. Furthermore, for near spherical objects the prediction scheme we use works relatively poorly (the normal velocity component is very pessimistic), so the algorithms process a large number of missed predictions, favoring our approach.

3.3.2 Sand through the hourglass

In this example, grains of sand (cubes with perturbed corners) are dropped through a funnel, coming to rest on a slightly peaked surface below the funnel (figure 3.9). This scenario exhibits highly variable collision frequencies as objects fall through the funnel with relatively few collisions, and then come to rest with a very high collision frequency. It also exhibits clustering in all spatial dimensions as the objects come to rest on the almost flat surface.

The grains were dropped at a constant rate from uniformly random positions and orientations at a fixed height above the funnel, starting with zero velocity. Each data-point is the average of ten simulations with different initial conditions.

Both simulators used in this experiment tracked the velocity of the objects over time and decided that the object was at rest if its velocity remained near zero over the course of many collisions. Resting objects were treated as stationary until another object hit them, at which point they could again begin moving. Detecting stability significantly improves the speed of simulations, for both asynchronous and synchronous algorithms, although at the cost of additional implementation complexity. It is a simple example of detecting changes in an object's behavior and shifting simulation modes (from dynamic to fixed object) to deal with it.

Experimental results are summarized in table 3.2 and figure 3.10. Of particular note is the leveling off in speedup as the number of objects reaches some threshold. This is due to several things:

- The clustering of objects makes each intersection data structure update a potentially linear operation. Better data structures could solve this problem.
- The temporal bounding volumes for our algorithm will be larger than those for the synchronous algorithm, which generates more intersections without providing additional useful information.

	Synchronous			Asynchronous		
n	#events	t/event (ms)	t (s)	#events	t/event (ms)	t (s)
10	117009	0.22715	26.579	147358	0.055090	8.118
20	316778	0.36251	114.83	351714	0.059932	21.079
30	496172	0.47717	236.76	584747	0.063898	37.364
40	756671	0.56904	430.58	872613	0.069489	60.637
50	1010548	0.64935	656.20	1107481	0.075613	83.740
60	1383517	0.72677	1005.5	1578943	0.084519	133.45
70	1697377	0.78985	1340.7	1961114	0.088551	173.66
80	2149780	0.87392	1878.7	2497338	0.10317	257.64
90	2630125	0.93414	2456.9	3126652	0.10712	334.94
100	3156533	1.0239	3231.9	3614242	0.11155	403.15

Table 3.2: *Results for the simulation of sand falling through a funnel to come to rest below. The number of events, time per event, and total simulation time for our algorithm are compared to data for a traditional algorithm for a range of n, the number of moving objects in the simulation. This example processes slightly more* update *events then the polyhedra in a box example, which is expected considering its heterogeneous nature. Furthermore, processing time per event for the asynchronous algorithm grows faster in this example, due to additional time to process the intersection data structures, and fewer cheap missed prediction events. Times were obtained on a 200MHz Pentium Pro.*

- The prediction scheme works well in this case, so there are fewer very cheap missed predictions.

In summary, our algorithm shows a linear speedup until the point where objects begin to cluster significantly, after which it is a constant order of magnitude faster. The reduction in performance is a direct result of simulation conditions that make it difficult to track potential collisions efficiently — no published impulse-based algorithm handles such cases well.

3.4 Discussion

The experimental results indicate that our algorithm is significantly faster than existing techniques, and for at least one example, this speedup scales as the number of objects increases. However, the results also show that, in cluttered environments with good prediction, our algorithm provides only a constant speedup for some range of data. Improving our algorithm requires addressing the causes of this reduced speedup.

The intersection data structures we use, based on linear sorts, take time proportional to Δk, the number of changes in one-dimensional intersection status. These could be replaced

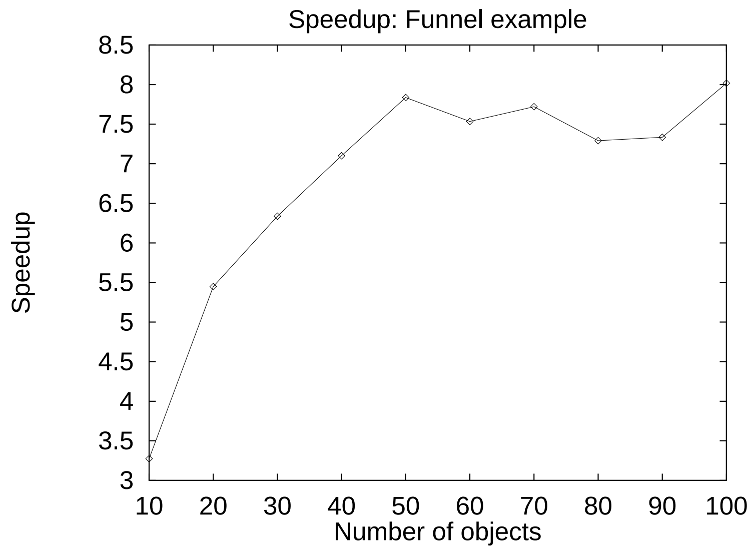


Figure 3.10: *The speedups obtained by the algorithm described in this chapter over a traditional approach (algorithm 3.4) for the simulation of sand in a funnel. For lower numbers of objects, our algorithm demonstrates an increasing speedup as the number of objects increases, but this speedup becomes constant as the number increases beyond 50. The lower speedups for more objects is due primarily to increased clustering as the objects come to rest, and hence poorer performance of the intersection data structures. Better data structures could improve the algorithms performance in such cases, but simulating cluttered objects that are essentially at rest is a case best handled by constrained-contact algorithms.*

by a hierarchical data structure that always took $\max(\log n, k)$ time [91], where k is the number of intersections reported. Alternatively, the spatial hashing scheme described in [60] would always take time proportional to k in our situation.

The heuristic for choosing the expiration time of temporal bounding volumes looks only at the time since the last event for the object. Alternatives could be explored, particularly those that consider the number of intersections generated by the new bound (based on a guess from the last bound). Ultimately, the number of intersections for a given bound is a property of the simulation itself, which may be impossible to reduce in cluttered environments.

The algorithm, as stated, assumes a conservative prediction scheme. Lifting this restriction may result in interpenetration, requiring that the simulation be backed-up, or unwound, to a time before the offending collision. This can be implemented by storing a small amount of recent state to unwind to, or even by selectively unwinding only those parts of the simulation that could have been invalidated. The recent event history, in combination with the intersection data structures, provides sufficient information to perform selective unwinding, although the implementation details are complex. The most difficult aspects of an unwinding strategy are deciding how much previous state to maintain and the design of efficient data-structures.

Note that most prediction schemes, even if not conservative, can be designed to be right most of the time, so we expect to be wrong relatively infrequently and hence it doesn't matter if the cost of an incorrect prediction is quite high. This is particularly the case when interactive user's lead to incorrect predictions: the user's events are likely to be very rare compared to simulator events.

The event sequence and temporal bounding volumes also provide information for re-using data from one simulation in another simulation — a potentially large saving given that simulations are commonly performed in large numbers with only small changes between each run.

In a prototype system, we break the simulation into discrete blocks of time. For each block, which covers a distinct interval of time, we record the maximum spatial extent of every object over the time interval, and the state of each object at the start of the interval. When the next simulation is run, objects are marked as changed or unchanged. Changed objects are simulated as normal, but each unchanged object is only considered when its bounding volume is intersected by a changed object, at which point the previously unchanged object is considered changed and the simulation is backed up to the start of the last recorded time block.

Other objects may need to be simulated to get the correct behavior for the previously unchanged object, but with some care we can guarantee that the simulation that results is correct.

Experimental results for re-using simulation data suggest that the scenario being simulated has a significant effect on how much data is re-usable. The most important property is *mixing*: how quickly changes at one point propagate throughout the system. If changes propagate widely, then there tend to be few opportunities for re-use, because most of the simulation will be different between successive runs. If changes tend to remain local, then re-use is easier to exploit, because little changes between runs. We are currently working to improve our re-use algorithm to ensure that it does as well as mixing allows. Tentative results with a simulator that does re-use for the spelling balls simulations of chapter 2 show speedups by a factor of two over a simulator that does not re-use any data. Most of the speedup results from cases where the change from one run to another is to a ball that enters the simulation late, so most of the earlier frames of the simulation can be re-used.

Finally, we note the compatibility of this algorithm with that of Sudarsky and Gotsman [81] for occlusion culling in dynamic scenes. Their technique requires temporal bounding volumes with “expiration times”, as provided by our algorithm. In an interactive rendering setting, our algorithm could avoid computing frame-by-frame state information for objects out of view, which would provide a simple means of reducing the computational cost of out of view motion, but not as efficient as the techniques described in the following chapters.

There are many important avenues for future work in the area of basic simulation models, including models for collision resolution and models that combine the best aspects of different paradigms to achieve superior performance — referred to a *hybrid simulation*.

As discussed briefly in chapter 2, collision friction models are not adequate for capturing some physical phenomenon, such as the seven-ten split in bowling. Restitution models are also inaccurate when compared to experimental data [12]. It may not be possible to define any model that captures the world in an completely accurate way, because the real world behavior may be sensitive to dust or oil that is difficult to model analytically. A more achievable goal may be to look for stability in the results of a simulation, in the sense that the outcome predicted by the simulation should be robust to variations in the parameters or the model itself. In this way, even though the parameters to the model are likely to be imprecise, at least the outcome may still be at least approximately correct.

Hybrid simulation is the term given to simulators that use different simulation modes depending on the behavior of the objects. For example, a hybrid simulator would simulate the sand through the funnel as an impulse system while the sand was falling, but a multi-point contact force system [5] when the grains come to rest. The difficult part of hybrid simulation is detecting when to change modes, but the payoff is large. In particular, it lets the underlying complexity of the simulation drive the algorithm, rather than the restrictions imposed by an inappropriate (and often inaccurate) model.

Chapter 4

Simulation Culling

The animation task for a simulator in virtual environments is to generate any time-varying state in the environment. More precisely, the simulator must compute the mapping $S_i = M(S_{i-1})$ to evaluate the state, S_i, at frame i, given a mapping function which encodes the simulation model, M, and the world state on the previous frame, S_{i-1}. Assuming a minimum acceptable frame rate for rendering the environment, the new state must be computed within the time allotted for the simulation for that frame.

In most environments, most of the world is not visible most of the time. While for virtual environments this may be by design, the real world clearly has the same property from most viewpoints, although on a larger scale. Visibility culling schemes [1, 22, 25, 53, 65, 72, 84, 96] exploit this fact to render efficiently, and simulation culling, described in the following chapters, exploits it to achieve scalable simulation.

Simulation culling provides scalable simulation by computing only what is necessary for the current view, and ignoring non-visible motion. With culling, it should be possible to make the total environment arbitrarily large, provided only a small amount is visible at any given time. For instance, we should be able to model an entire virtual city, but only pay to simulate the few blocks we can see at any moment. We will refer to a simulation model for the entire world without culling as the *complete model*, and a model that performs simulation culling as the *culling model*.

The primary concern when culling is that the simulation should still appear satisfactory to the viewer, or meet certain *quality criteria* defined by the user. The nature of the quality criteria will depend on the particular application. For instance, the quality of a training simulation might be defined by comparing a culling model to the real world: culling is successful

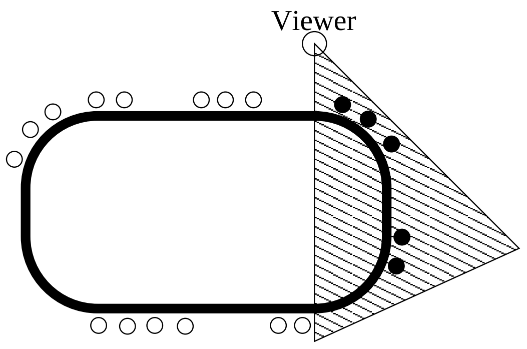


Figure 4.1: *A hypothetical racetrack simulation to demonstrate the problems that arise from culling. If the viewer can only see the shaded region, we wish to simulate only the filled cars, ignoring as much as possible the outlined cars that are out of view. This introduces two problems: how do we know when cars that have left the view should re-enter it again having passed around the track; and when a car does re-enter the view, how do we know what state it should be in, given that we haven't been solving its equations of motion for the time out of view.*

only when a viewer can't tell it apart from reality for the cases of interest. In a computer game application, the criteria might be much weaker: culling is successful if a viewer does not detect any obvious mistakes, such as a monster behaving strangely as it leaves and re-enters the view. Computer games already perform some culling, such as, for instance, not starting a monster's simulation until it is visible for the first time. Players learn the way the game behaves and expect no more.

Within the constraints imposed by the quality criteria, a culling strategy is perfect if it does work only for motion that influences the view. We define the efficiency, η, of a culling model as the ratio of the amount of work done for motion in view to the total amount of simulation work. The aim of a culling strategy is to be as efficient as possible. The perfect algorithm has efficiency one, with lower efficiency implying more work done for motion that is never explicitly seen. Actual culling models will in most cases have efficiencies below one, because they must perform some work to meet the quality criteria.

4.1 Problems Arising from Culling

Say we wish to simulate a virtual car race on an oval track. For the purposes of discussion, assume the viewer can only see one turn (figure 4.1), and that we wish the viewer to believe that the race is realistic, given what they can see of it. We aim to build an efficient culling strategy for this scenario.

To ensure realism, we might impose the following quality criteria:

1. Each car should take a reasonable time to do each lap, given the presence of the other cars which may cause it to go slower or faster.

2. Whenever a car is in view, it should have a speed and orientation typical of a real racecar. Cars should not (in general) be touching each other.

If these criteria are met, we might reasonably assume that the viewer is experiencing the right thing.

At a high level, for each frame of our virtual race track animation the simulation module must perform two tasks:

- Visible cars must be identified, a task we refer to as the *completeness* problem. Identifying visible cars requires a way to determine where each car is while doing the least amount of work, because if a car is not visible we don't want to waste time determining its location. In particular, in order to satisfy criteria 1 above for the racetrack, we must ensure that when a car leaves the view it re-enters after an appropriate time, to ensure the lap time is reasonable. The completeness problem may be defined another way: it is necessary to ensure that everything that should happen in the world does happen, at an appropriate moment and with the least amount of effort.

- Reasonable state must be generated for those cars that are visible. This is the *consistency* problem, because we must ensure that this state is consistent with the quality criteria and a viewer's past experience of the world. The consistency problem is simple when the car was in view on the last frame – just update its position from there. But the problem is difficult to solve for cars that were previously out of view. The state of those cars should be determined by solving a continuous set of equations for the time out of view, but that would introduce unacceptable lag. We must use a faster method, but one that still produces reasonable state given what the viewer knows of the car's history, such as how fast it was going when it left the view.

In practice, the extent to which the completeness and consistency problems manifest themselves is highly dependent on the simulation being culled and the quality criteria for the application. If a simulation is intended to give a very accurate impression of the behavior 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, as might be the case for computer games or other entertainment applications. Films, for instance, frequently contain inconsistencies yet the experience remains acceptable to most viewers.

4.2 Related Work

Sudarsky and Gotsman [80] have demonstrated algorithms for incorporating dynamic
models into a BSP tree visibility scheme. They assume that the state of each object can be expressed as a closed form function of time. They do not explicitly discuss the problem of simulation culling, but in effect they address the completeness problem by bounding the maximal extent of an object for some period of time (much as we will do), and solve the consistency problem by evaluating the object's equations of motion directly. The restriction to closed-form equations of motion excludes most interesting simulations. The work described here lifts that restriction: it is not necessary to have closed form solutions for the objects' future motion.

Setas et. al. [76] demonstrated a forest environment in which simulation level-of-detail techniques were used to reduce the cost of simulating systems that were distant from the viewer or out of view. While not culling, they did achieve significant speedups while maintaining a visually satisfactory world. Carlson and Hodgins [13] also applied level-of-detail techniques, in their case to hopping robots playing a game. Three different models were used for the robots: dynamic, kinematic and point mass. Significant speedups were achieved, and the authors describe some validation experiments that demonstrate the quality of the resulting simulation. While not presented as a level-of-detail strategy, the NeuroAnimator approach by Grzeszczuk et. al. [37] can be used to reduce the cost of systems that are distant or out of view. However, they provide little evidence of the quality of the resulting motion, particularly over extended periods of time.

The synthetic motion capture approach of Yu and Terzopoulos [95] is an explicit level-of-detail scheme in which the true motion of systems out of view is approximated by data captured from the correct simulation. While resulting in some computation savings, not all of the motion is approximated, so there appear to be additional savings available. Furthermore, the authors note that the approximations can significantly affect the quality of the simulation. In particular, the approximate simulation exhibits significantly different behaviors when compared to the real one, which is unacceptable for many situations.

The work described in the following chapters improves upon existing work in two ways:

- It achieves higher efficiencies by performing less, in some cases zero, work for out-of-view motion.
- It looks more carefully at quality criteria, and motivates the choice of particular models by the quality criteria they meet.

4.3 Quality Assurance Through Probabilistic Comparisons

Let us return to the definition of the simulation task: to compute the dynamic state of the world at each frame, given the previous frame. When culling, we wish to restrict our attention to those components of the full state that are visible to the viewer on a given frame. The aim of modeling for culling is to generate state for the visible components while performing as little work as possible, subject to the need to satisfy any quality criteria.

We choose to view simulation as a statistical exercise, where the aim of a simulation model is to compute samples of how the world might behave given some parameters (such as the initial conditions for the world). If the world is completely deterministic, there is only one acceptable sample for a given set of parameters. On the other hand, many environments contain random components, such as virtual creature models [9], models of natural phenomenon [76], or other random effects (chapter 2). In those environments each particular choice for the random variables represents a different sample world, and we can talk of a distribution over worlds, and compute expectations and other statistic according to that distribution (chapter 2 discusses this idea in some detail in the context of direction).

In the context of culling, we assume that the complete model (involving all the objects in the world all the time) defines how the ideal world should behave. We can capture expectations for the behavior of the model in the form of *reference distributions* based on the output of the complete model. For instance, in the race track simulation our expectations of reasonable behavior might be based on the distribution of lap times when all the cars are being fully simulated.

Quality criteria can be viewed as statements about the desired similarity between the complete model reference distribution and the culling model's output distribution. We might only require similarity in some aspects of the distribution (for instance, we might care about relative lap times but not absolute lap times for the race track). We might allow the distributions to differ in their tails, which means that the likelihood of rare events would be different. For example, we might be willing to accept fewer crashes in a culled race track simulation (and hence no infinite lap times), but more willing to accept higher variance in lap times, making very slow or fast laps more likely when culling.

It is reasonable then to view computing some of the visible components of the state as a sampling problem, particularly those components that have not been seen in recent frames. The complete model along with states the viewer has previously seen together imply the reference distribution. A culling model is free to sample states in any way, provided that the resulting sample distribution on the visible components is sufficiently close to the reference distribution, where sufficiently close is defined by the quality criteria. Note in particular that with culling, we assume that the quality criteria are only applied to subsets of the simulation that are visible to the viewer. Hence, we only care about the similarity of the appropriate marginal distributions on visible states.

The completeness and consistency problems may now be re-defined:

Completeness is the task of determining which marginal distribution to sample from. We are only concerned with motion a viewer can see, so we only care about the marginal distribution on visible state. Yet we must ensure we get the right marginal, to make sure the viewer sees everything they should.

Consistency is the task of sampling from the appropriate marginal. If the sample distribution is similar to the reference distribution according to the quality criteria, then we can be sure of presenting a valid state to the viewer.

4.3.1 Approaches to the Culling Problems

The above analysis suggests that direction and some aspects of culling are related problems: both require sampling from a distribution implied by a physical model and constraints on that model. In the case of culling, the constraints are on the state of previously invisible objects, and arise due to things the viewer has experienced in the past. For direction, the constraints are outcomes we would like to ensure.

However, it is not necessarily the case that the same technology should be applied to both problems. For direction, the conditioning is frequently on events that must happen in the future, so we must solve an inverse type of problem: determine which values to use at the start in order to make the right things happen at the end. On the other hand, for culling we are always solving a forward problem: given what has already happened, rapidly sample reasonable values from the distribution of final state.

Two approaches to sampling for culling will be discussed in the following chapters:

- We can model the target distribution on previously unseen state directly, and sample from that model. This approach is useful if the distribution can be modelled and we can sample from the model in real time. It focuses on the effects of the simulation as a whole, rather than attempting to track the evolution of individual components.

- We can avoid modeling the distribution explicitly, and instead model the expected evolution of each state variable over time. It is assumed that evaluating each individual variable results in samples that are reasonably distributed, without explicitly solving a sampling problem.

Chapter 5 describes technology for addressing the consistency problem, including tools that for some cases automatically generate simulation models suitable for culling. The work presented there assumes that the global spatial extent of objects in the simulation can be bound over all time, and hence the completeness problem does not arise. Chapter 6 removes that restriction, discussing an algorithm to solve the completeness problem. We present an implementation of a virtual city in which cars not in view are culled, and examine the effectiveness of the completeness algorithm before concluding with a range of open issues.

Chapter 5

Consistency when Culling

The consistency problem arises in isolation when culling objects that have limited range, because, while it is easy to determine roughly where such objects should be in the world, it remains difficult to know what state they should be in when they re-enter the view after being culled. Previous work [17] described hand-coded models that used various approximations and statistical models to solve the consistency problem. This chapter is primarily focussed on automated tools to perform the same task. We restrict our attention to simulations that can be broken into independent systems each with a low-degree state space. In addition, we describe a modeler for authoring dynamical systems and run-time algorithms for incorporating them into an environment.

To solve the consistency problem, it is necessary to generate simulation state for an object when it re-enters the view in such a way that the new state is plausible given what has occurred previously in the world. For entertainment applications, the state may be considered plausible if it does not offend the viewer's expectations as to what should happen in the world. For a training environment, there may be the additional requirement that the viewer be presented with a state that could reasonably exist in the real world, given what is known to have occurred so far.

As discussed in chapter 4, we view the simulation model and past events as implying a distribution on the state of objects that were not visible on the last frame. For a deterministic world, and complete, accurate knowledge of past state, all the probability density for each future state is isolated in a single set of state values, presumably determinable only by simulating the equations of motion. Fortunately, however, a viewer can never have perfect knowledge of the past, nor perfect knowledge of the model. Similarly, even training simulations may have some scope for approximating the real behavior of the system, given that the model being simulated is probably already an approximation to reality.

To see how imperfect knowledge can be used to arrive at quality criteria for culling, consider a probability distribution defined over some part of the simulation state that has left the view, and how this distribution evolves over time. Our quality criteria will be satisfied if we can sample according to the distribution corresponding to the moment the object re-enters the view. With imperfect knowledge, the distribution initially has all the density concentrated around the particular state the viewer saw, but not isolated at a point. The spread arises from the viewer's imperfect knowledge of the world, due to, for instance, the difficulties of determining the exact velocity of a moving object.

As the system evolves out of view, the density will be shifted around the state space according to the equations of motion for the simulation. We might also expect the density to be smeared over the space, depending on how chaotic the motion is [38], and how much randomness is present in the model itself. We can use the smeared distribution to define our quality criteria: provided that the state we present to the viewer has reasonable probability according to the appropriate density at the time it re-enters the view, the viewer will find the state plausible, and our culling strategy will be acceptable.

5.1 Qualitative Analysis of Conditioning

We qualitatively identify three ways to generate samples from the distribution of reasonable states for objects re-entering the view, characterized by how long the object has been invisible. This discussion is predicated on independence between the visible and invisible parts of the simulation, so we can consider culled systems in isolation, where a system is a set of related state variables, unrelated to any other state variables. In particular, the only thing that can influence the state of a system when it re-enters the view is its own last known state, rather than anything the viewer may see while it is out of view.

5.1.1 Short periods out of view

Over short periods of time, knowledge of past events has a significant influence on what is considered plausible state. Consider the case of a falling rock, where the viewer sees the rock go over the edge (figure 5.1). While the rock is in flight, but before it hits the ground,

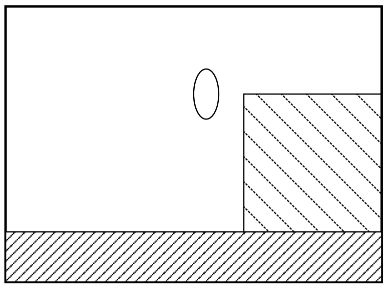


Figure 5.1: *An example of short term prediction: A falling rock may be accurately predicted until it hits the ground.*

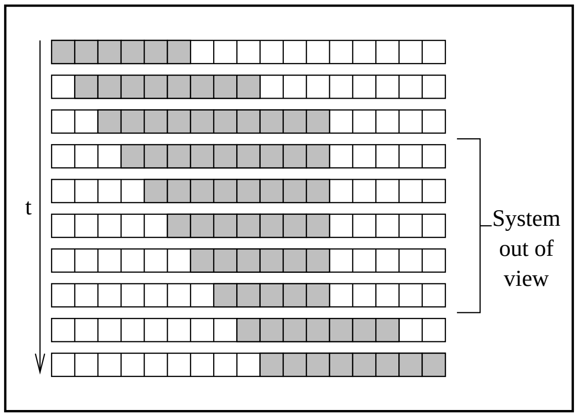


Figure 5.2: *Dynamic state may be buffered ahead of the rendering, as indicated by the shaded regions. When the object is in view, the buffer is filled faster than the renderer consumes values - three times faster in this case. When the system is not in view, no new values are computed, but the old values 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. State may also be interpolated between values in the buffer to make the frame rate independent of the simulation time steps.*

a viewer can readily extrapolate the expected state of the rock, because they have good experience of how falling objects behave, and too little time has passed for uncertainty to propagate. To summarize, uncertainty doesn't grow very much over short periods of time for smooth dynamics.

To ensure consistency in this situation the most accurate simulation model must be used, saving no computation. To avoid lag, the simulation may be run ahead of the rendering and state values buffered (figure 5.2). When an object goes out of view, filling stops, but if the object re-enters the view soon after, a value is ready in the buffer without incurring lag.

While using buffers to reduce lag while culling is a novel idea, it is not atypical to

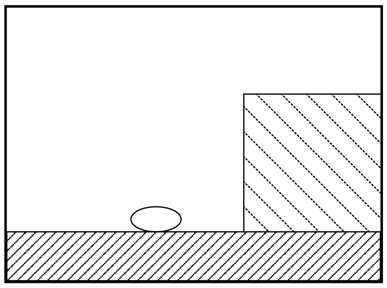


Figure 5.3: *An example of medium term prediction: Once the falling rock hits the ground, its position and orientation are uncertain. However, a viewer does know for certain that the rock is on the ground at the bottom of the hill.*

buffer the values of state variables between a simulator and a renderer. The practice arose because the rate of generation of state by a simulator may differ significantly from the rate of consumption by a renderer, and a buffer can smooth the flow. User interaction complicates the issue slightly by requiring buffer flushes when the user's actions invalidate state. Buffers also allow a slightly more relaxed definition of real-time performance. The tight definition is that a system is real time if the new state, $S_i = M(S_{i-1})$, can be computed for each frame in the time available for that frame, while the loose definition allowed by buffers is that a simulation is real time if the required state variable is always ready in the buffer when required to render a frame. In other words, the simulation can compute some frames at a slower pace than real-time, provided it can compute others faster.

5.1.2 Medium periods out of view

Over the medium term, the range of plausible state is in a transient zone. While the range of plausible states tends to increase, leading to some values being essentially free, there may still be significant constraints. Following the rock example (figure 5.3), 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 different expected behaviors.

Looking at the rock example within a probabilistic framework, the likely positions for the rock fall within a small set before the rock hits the ground. After several impacts, the likely positions are spread over the ground plane, but are very localized in the vertical dimension. However, it is difficult to explicitly state what the distribution should be, because it depends on where the rock fell from, how it bounces, and the nature of the ground. Any of

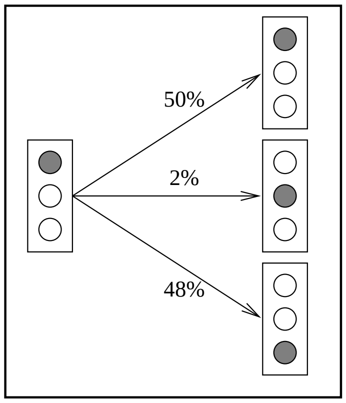


Figure 5.4: *A traffic light is an example of long term prediction. If the light has been out of view for more than a few minutes, a viewer cannot say exactly which state is plausible. However, a viewer does expect the light to be red or green more often than it is yellow.*

these things may change from fall to fall, so the distribution would have to be parameterized in some way that was simple to estimate and sample from.

Instead of capturing the distribution, approximations can be used to provide a state for the system. The motivation for approximations is that they may be relatively easy to construct and evaluate, and they are applicable as long as the state computed from them is reasonable (has high probability) given the viewer's knowledge and the model. The approximation must generate a new state for the system given the old state and the time out of view. The errors in the approximation should reflect the range of plausible states available. As long as the error is within this range, the viewer will consider the outcome plausible. Many methods exist for approximation; neural networks are used here.

While not discussed further here, in some situations it could be preferable to build the distribution explicitly and sample from it. One such case may be systems described by Markov processes, where the distribution after some number of frames is defined by matrix multiplication.

5.1.3 Long periods out of view

In the long term, for many simulations previous knowledge can only weakly condition the range of plausible states. Yet some states are still more plausible than others. Traffic lights provide a good demonstration (figure 5.4). A viewer's expectations reflect the general behavior of a traffic light, which is to be red more often than green, with yellow least likely. To exploit this, a new state can be sampled from the simulation's stationary distribution, which will capture reasonable long term behavior for the system, independent of any previous knowledge. We can use standard density estimation techniques from various sources to model stationary distributions [8, 56].

The use of a stationary distribution reflects the fact that, for many simulations, 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. It is possible to explicitly use the distribution in this case because the distribution is invariant over time, so can be computed and stored as a pre-processing phase.

5.2 Simulation Models for Culling

The tools described in this chapter take as input a basic description of the simulation and produce an alternate description suitable for culling. The input description is assumed to be the most accurate required for simulation: the *complete model*. We will call the output description the *culling model*. The tools work for dynamical systems that are free from external influence, and have a state space of low dimension. They are only useful 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.

The tools were implemented in Java [2] and VRML [90]. The accurate model consists of a Java class file which implements a function for evaluating the system at some given time when given an initial state and time. The function should be able to generate 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.

Systems are classified according to whether they are periodic or not. A system is periodic if its state, S_t, at time 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.

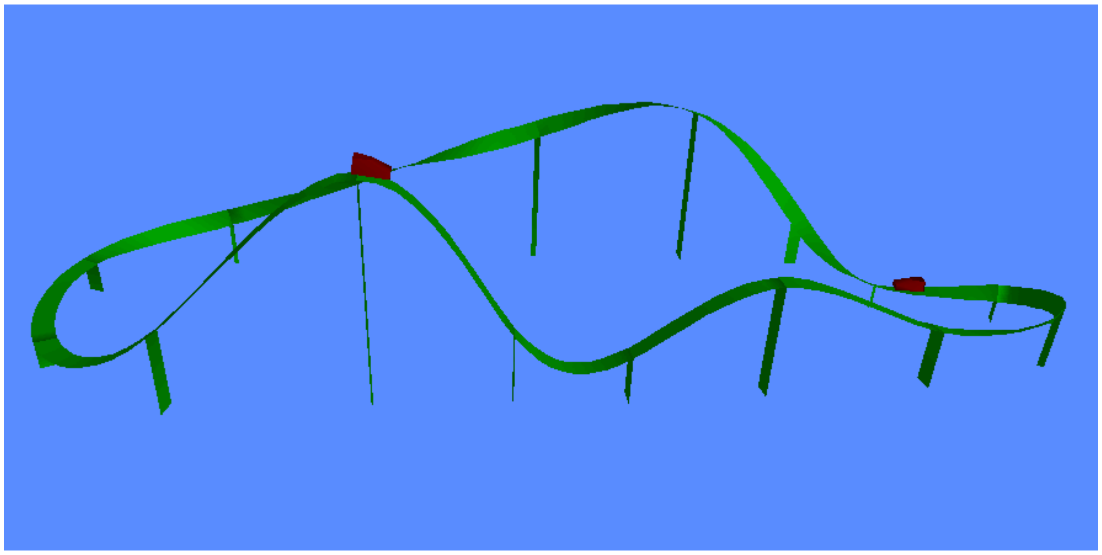


Figure 5.5: *The roller coaster model. To cull this model, we build approximations to each car's position and velocity as a function of the fraction of a period completed. When the ride has been out of view for more than a short period of time, the approximations are evaluated instead of solving the equations of motion for the time out of view.*

5.2.1 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 (figure 5.5), under the influence of gravity but without friction. The "upright" direction of the car 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, $\dot{u}$.

Consider what behavior a viewer might expect from a system like the roller coaster. Over very short periods, on the order of a couple of 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 is reasonable. Our task is to find functions that are quick to evaluate and whose output is close enough to the true dynamics to remain plausible.

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 $\hat{S}_{t'} = \hat{\phi}(t' \bmod T)$. Provided the error in $\hat{\phi}$ with respect to ϕ is not large, a viewer will not detect the approximation error.

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 culling model.

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

Given the period, it is possible to determine $\hat{\phi}$, the approximation function, which we represent as a neural network [8]. Neural networks have the advantage of near constant evaluation time for a given network, and are able to approximate well the wide variety of functions the tools may encounter. Specifically, a standard feed-forward neural network is used with two hidden layers and a fixed number of nodes. The network has one input node, corresponding to the time we wish to evaluate at, $t \in [0, T)$, and as many output nodes as there are state variables for the system. We use networks with ten 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 $(0, 1)$. In order to re-scale the state variables for a dynamical system to this range, the minimum and maximum possible values for each component must be found. This is done 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 [8] with momentum algorithm on these samples. Each sample is used $N - 1$ times, after which we replace it with a new

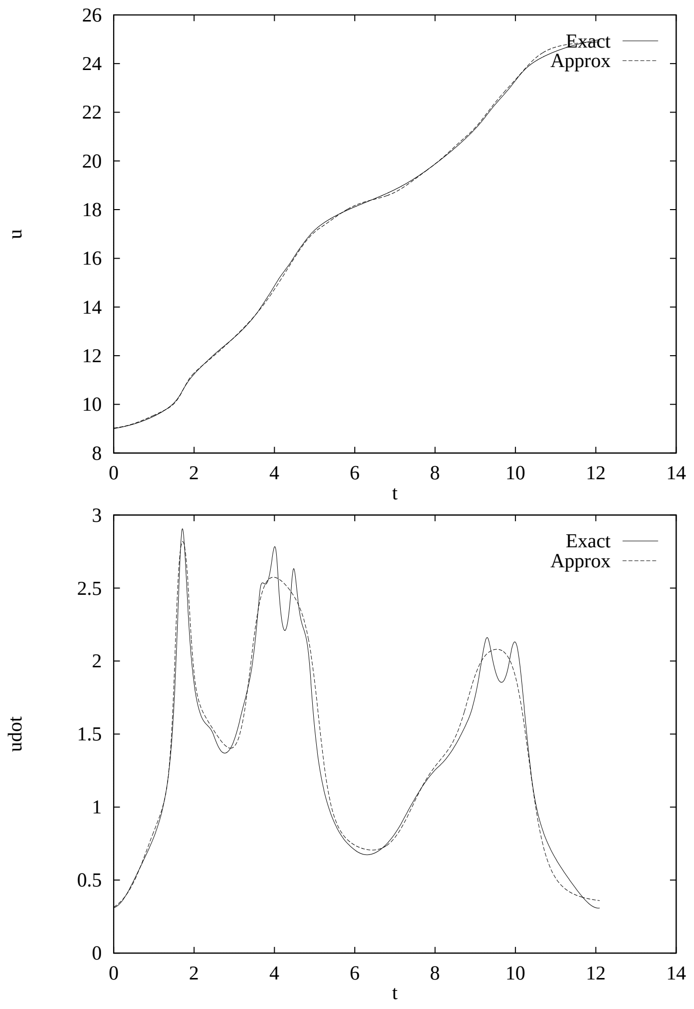


Figure 5.6: *The exact functions for the parametric position of a roller coaster car on its track,* $u(t)$*, and its derivative,* $\dot{u}(t)$*, and the neural network approximation to them, plotted for one complete period (12.13 seconds). The neural network was trained for around 30 minutes to achieve this result. The network approximates* u *with great accuracy, but does less well with the local maxima and minima of* $\dot{u}$*. This is not of concern in this example, because the underlying dynamical system can correct any error based on energy constraints. Currently, our tools use one network with multiple output nodes 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.*

sample. We do this to reduce the number of dynamical system evaluations, since they are 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. A plot of u and $\dot{u}$ as a function of time for the roller coaster model, and the neural network approximation learned, is shown in figure 5.6.

The final step is code generation, in which a new Java class is created to evaluate the state of a periodic system efficiently regardless of the time between evaluations. This new system can choose to either use the complete 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% of the period, otherwise it applies the latter. The threshold for using the approximation is somewhat arbitrary is this case. It could be specified by the user, or determined based on the error in the neural network as a function of time.

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

Many simulations, such as the roller coaster, have energy constraints which may be violated by the neural network approximation. In such cases it may be possible to set some parameters based on the energy constraint. For the roller coaster the speed could be explicitly set once the position was known, based on the total energy of the system.

5.2.2 Non-Periodic Systems

The Tilt-A-Whirl is an example of a non-periodic system (figure 5.7). It 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. 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.

Each car is independent, evolving according to the same equations of motion but out of phase to reflect their respective positions on the track. The state variables for each car 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, common to all cars, 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

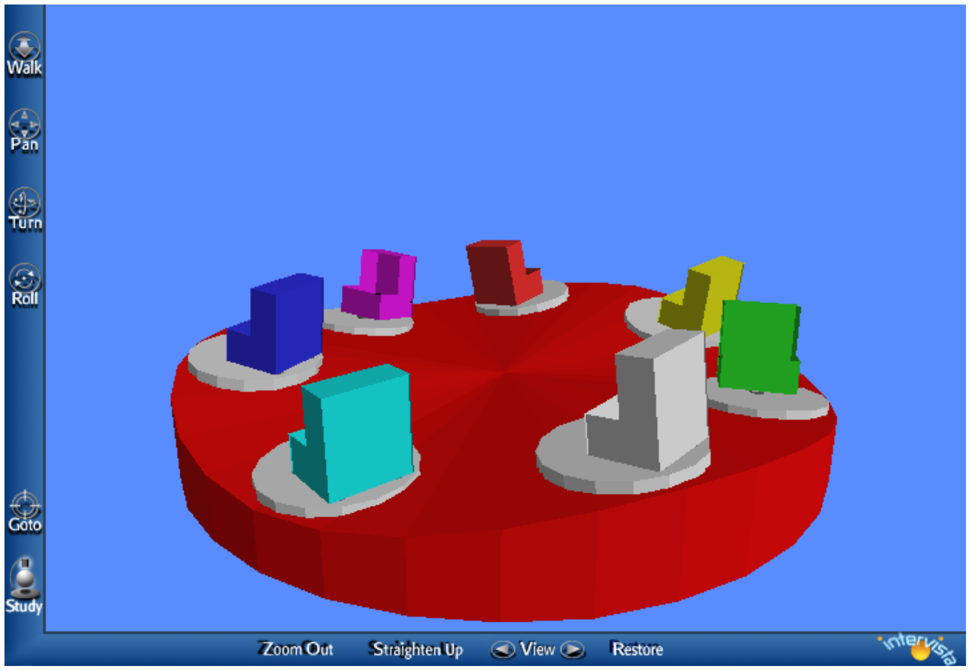


Figure 5.7: *A Tilt-A-Whirl amusement park ride. This ride is very difficult to predict over anything but short periods of time. Two stages of approximation are used: neural networks if the ride has been out of view for a relatively short period, and a statistical approximation if it has been out of view longer.*

hills; $\dot{\theta}$, the platform's initial angular position and velocity around the track; and ρ, a damping constant. Each car its own parameter, θ_0, specifying the initial position of its platform around the track.

The governing equation of motion for each car, derived from Lagrange's equation and making use of small angle approximations (following [49]) is:

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

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

Consider a Tilt-A-Whirl that moves out of the view, and then re-enters. If it is hidden for only a short period of time, a viewer can simply extrapolate from the state they saw when it left the 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 is out of view for longer periods, a viewer has increasingly weaker expectations for the new state, and the system can make larger errors in generating the new state without contradicting the viewer. In other words, approximations to the true system may be used, and the approximation error can grow as the time interval out of view grows.

After a Tilt-A-Whirl has been out of view for a long time, a viewer can no longer make use of information from the previous sighting to predict new state. However, a viewer can use their general knowledge of how a Tilt-A-Whirl behaves. To satisfy the 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 more often seen by a viewer will have a higher probability than states seen infrequently. To generate new state, we simply sample according to the distribution, which is essentially the stationary distribution for the system.

While the preceding discussion is phrased in terms of a Tilt-A-Whirl's behavior, the observations made are typical for any non-periodic 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 a complete 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 state space. 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 from the distribution.

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 complete 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 state space we must concern ourselves with, allowing discrete cell structures to be built on 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 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 harm the analysis. Also, some variables may be bounded by the user in the input description, particularly angular variables (which lie in $(-\pi, \pi]$ radians).

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 (user specified) size cells, and a probability,

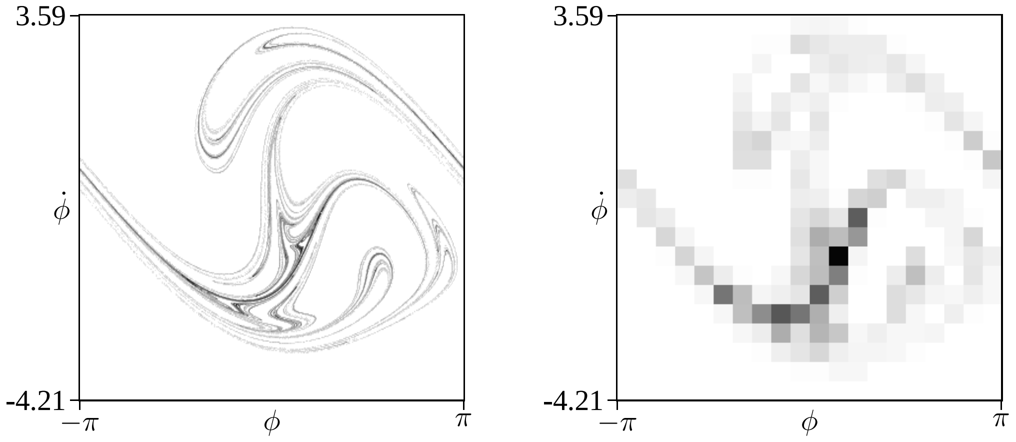


Figure 5.8: *The stationary distribution for the Tilt-A-Whirl system. The left image shows a high resolution image of the distribution, which assigns a probability density to every point $(\phi, \dot{\phi})$ in the state space, for the particular position on the track corresponding to $\theta = 0$. If we require state at a different θ, we sample one from this distribution and integrate to the required point. Darker points correspond to higher density, 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.*

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. 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, 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 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 5.8 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.

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 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 a viewer 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 state space, then integrate these paths for fixed time-steps (figure 5.9). 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}. This procedure is then repeated for other starting regions, until enough of the state space has been sampled from. The actual value used is the maximum t_{long} found for any region. Our methods directly examine the propagation of uncertainty, but other approaches may be applicable:

- Analytical methods based on differential analysis [38].
- Discretization methods that replace the continuous state space with a discrete one and the continuous differential equations with a discrete mapping between cells, and then apply the theory of Markov processes [44].

Building Approximations

The approximation functions built in this step will be used to generate new state quickly, with some error allowed. After some 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 more slowly than the time period over which they are evaluating. Neural networks are used, similar to those for approximating periodic functions.

In this step we generate several neural networks, each of which evaluates over its

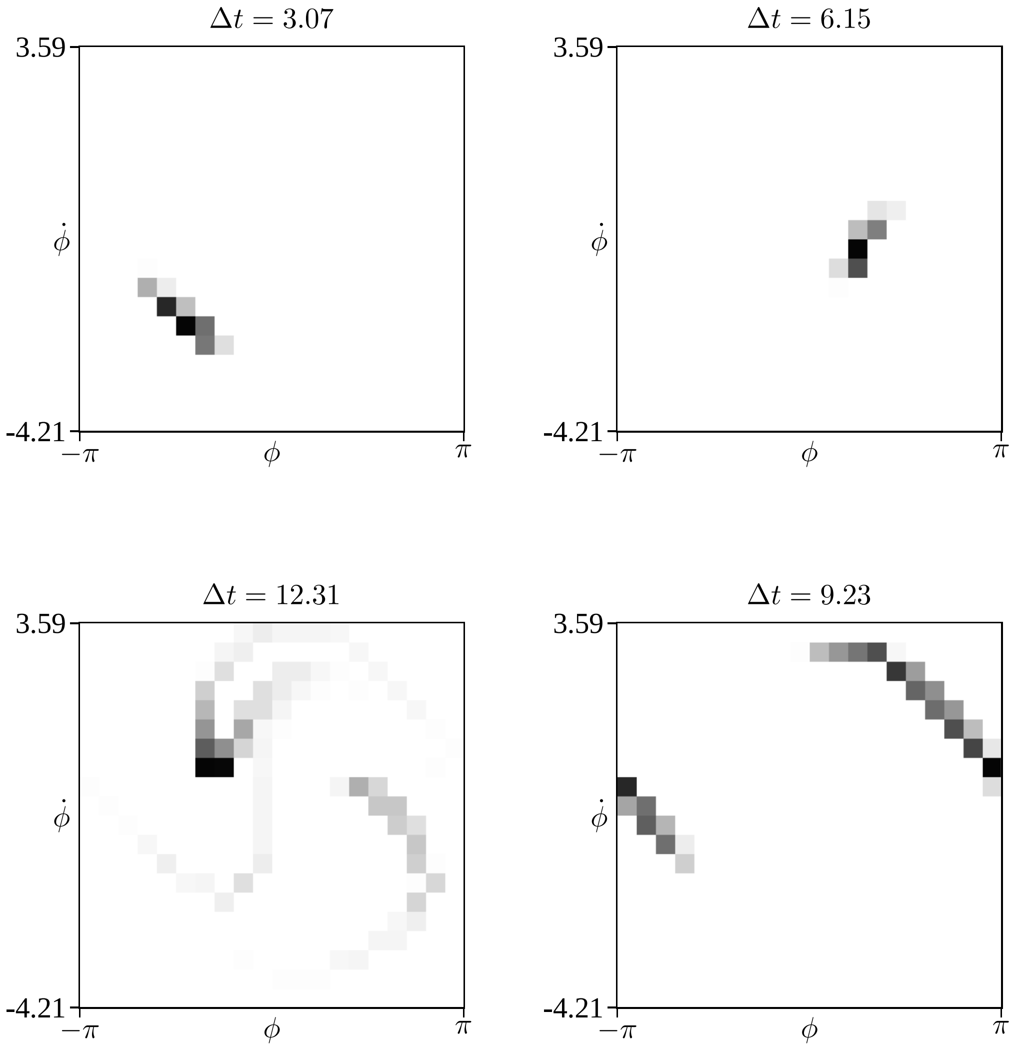


Figure 5.9: *The convergence of one cell to the stationary distribution, for the Tilt-A-Whirl model. Starting top left and moving clockwise, the plots show the distribution of 5000 paths after 3.07 seconds, 6.15 seconds, 9.23 seconds and 12.31 seconds. The distribution in the lower left is sufficiently close to the stationary distribution to stop testing for this cell. Other cells take up to 24.6 seconds to converge.*

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, $t_{long}/2$. 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 complete 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.
- We can learn networks concurrently, with significant improvements in training time and efficiency.

The samples the neural networks are trained on are distributed according to the stationary system (due to the method we use to generate them). This will result in the networks having lower error rates in regions of high probability, and higher rates in regions of lower probability (figure 5.10). This is 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.

Grzeszczuk et.al. [37] replace the equations of motion for a system with a neural network mapping function between successive states, with the aim of reducing the cost of frame to frame evaluations. They demonstrate the approach on more complex systems than

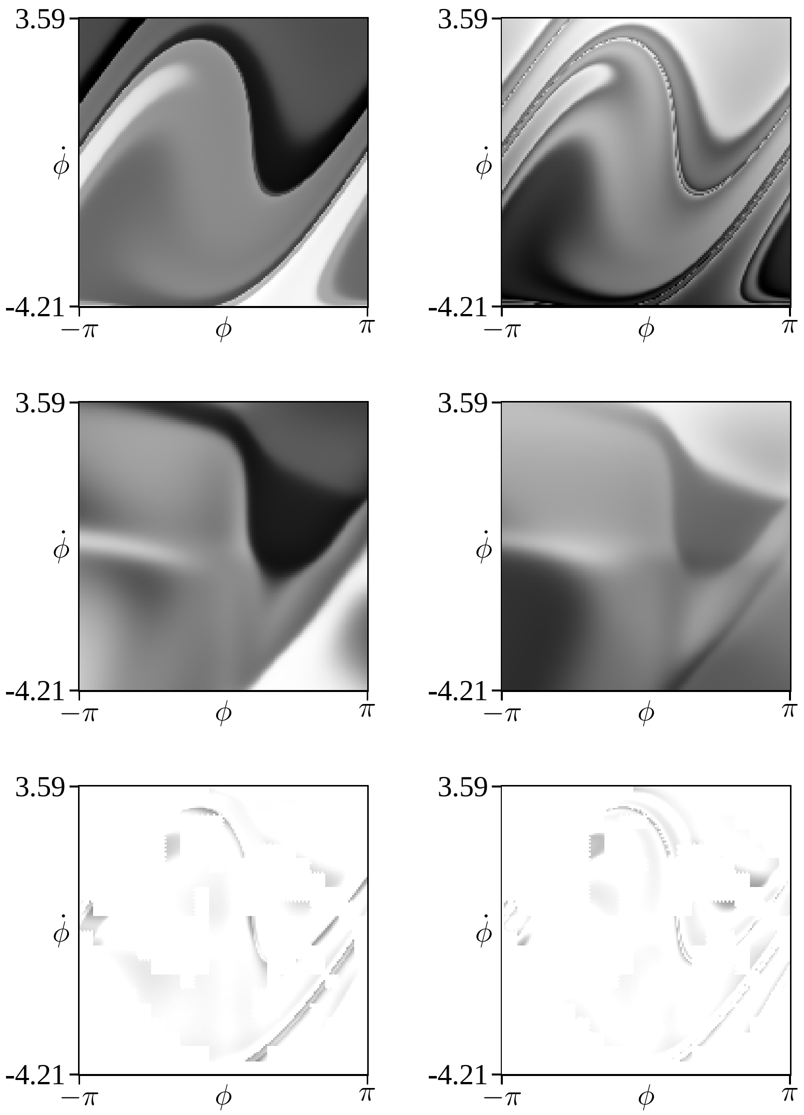


Figure 5.10: *An example of the Tilt-A-Whirl equations of motion approximated by a neural network. Intensity indicates function value: lighter is higher valued. The network shown is attempting to learn the change in orientation and change in velocity of a Tilt-A-Whirl car over a 6.15 second interval. The left column plots the change in orientation 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 5.8. 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.*

those handled by our system, providing additional evidence that neural networks are a useful technique for capturing dynamics.

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 how long before the approximation may be used instead 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, and we assume that neural networks are always cheaper to evaluate than the complete system, so we simply set t_{medium} to the smallest evaluation time of the networks we have learned.

Code Generation

The code generated for non-periodic functions allows efficient evaluation of new dynamic state 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 system was seen by the viewer. If the difference is greater than t_{long} it samples new values. Otherwise, if the difference is greater than t_{medium}, it uses a neural network approximation. Otherwise, it uses the complete model to generate state.

5.3 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 aim of a runtime layer is to provide an easy to use interface between code to evaluate the dynamics and the rendering system. In the specific implementation described here, the dynamics are evaluated by a Java class, and the renderer is a VRML browser. 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 renderer indicates a new frame[1], 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.

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 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 re-enters very soon after.

When the runtime system receives a request for new state, there are two possibilities, 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 repeatedly evaluated to fill the buffer until it signals that enough work has been done for one frame.

[1]In VRML, this is achieved through TimeSensor events, which we assume a browser sends at least once per frame

For the systems described in this chapter, a static bound can be computed that is certain to contain the object over all time. The restriction is imposed so that we need not solve the completeness problem when culling, and is lifted in the next chapter. The task of tracking visibility then reverts to the problem of determining whether a bounding volume is visible, a task that may be addressed in many ways.

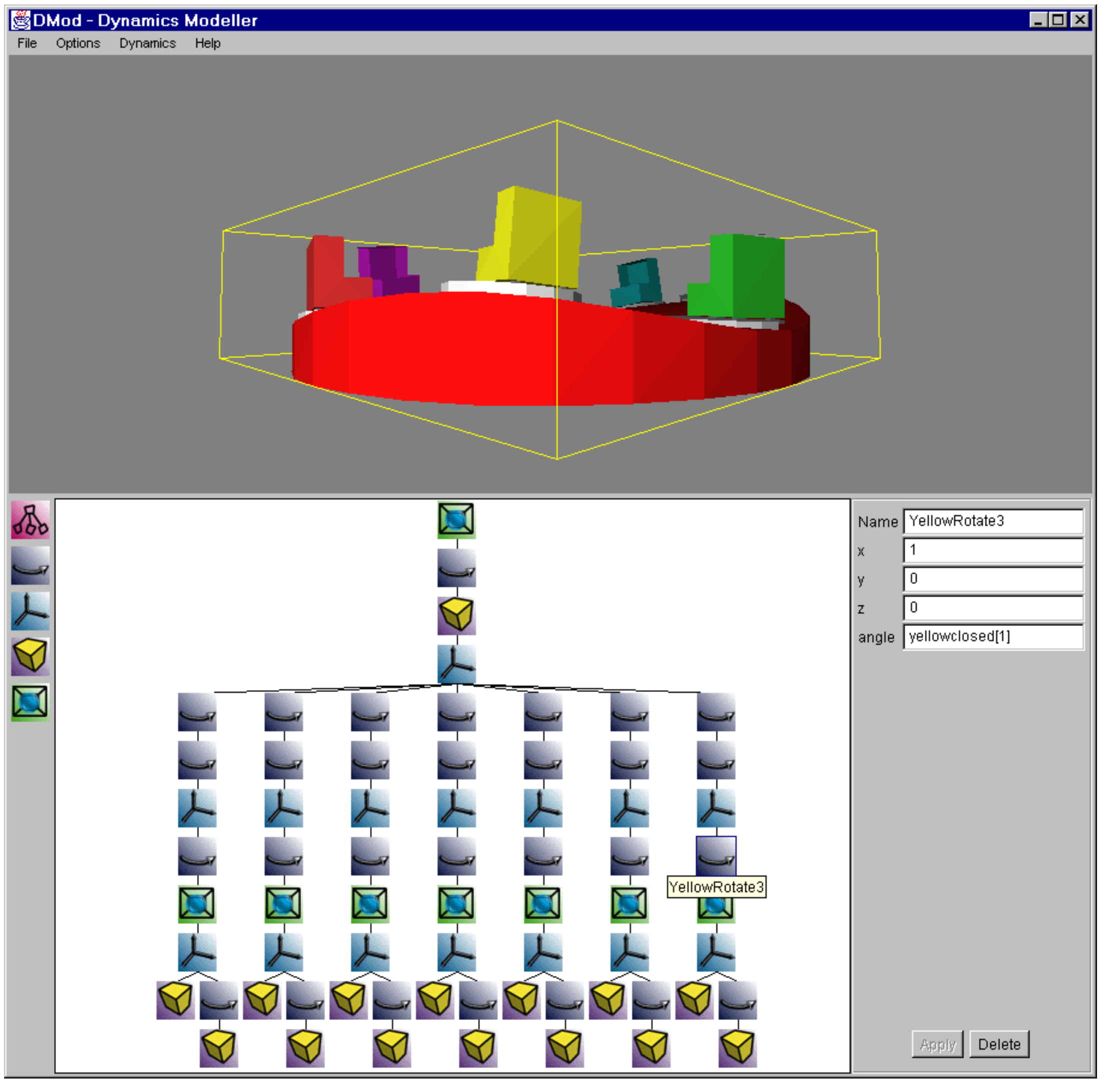
For the VRML implementation, we use VisibilitySensors defined by the VRML language. These sensors send events each time a bounding volume enters or leaves the user's view. In turn, these events cause the Java class to mark individual dynamical systems that control the object as visible or invisible. Invisible objects will not be evaluated for 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. The exact method by which visibility is determined depends on the VRML browser implementation.

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

5.4 Attaching Dynamics to Geometry

The rigid-body modeler allows a user to load geometric objects, 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 as described above, or it can save an intermediate file format. The interface is shown in figure 5.11.

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



Name YellowRotate3
x 1
y 0
z 0
angle yellowclosed[1]

Figure 5.11: *The interface for our dynamic transformation modeler consists of three regions. In the top panel the current geometric arrangement of the system is displayed, including its bounding volume. The transformation hierarchy for the system is displayed 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 a variable to animate the field. The user is shown here editing a rotation node, specifying a variable as the angle through which to rotate.*

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, in a VRML/Java implementation, 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 are also free to 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 provides a small, isolated environment in which to test dynamical systems, with reasonably strong error detection and debugging technology.

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. 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 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.

5.5 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 are shown in table 5.1. We began with one instance of each in the world, except for the Pendulum, of which their were two variations. We then added additional objects up to a maximum of 35 (five of each example). In each test, the average frame time was measured

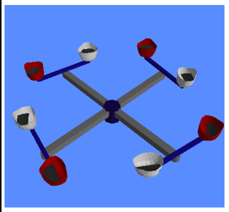
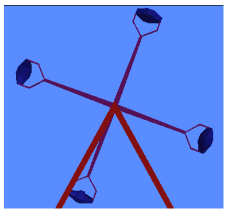
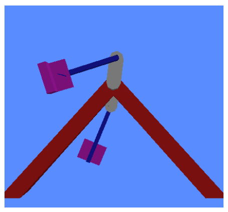
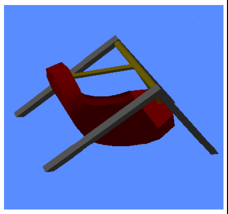
Name	Octopus	DiveBomber	Pendulum	Ship
#variables	10	5	3	1
#dimensions	2	2	2	2
period	none	none	none	3.07s
t_{medium}	15.0s	8.0s	1.92s	N/A
t_{long}	30.0s	16.0s	23.1s	N/A
#networks	1	1	4	1
time	11h53m	10h47m	6h46m	N/A
				

Table 5.1: *The fairground rides modelled with our tools and used in the experiments, along with various statistics. #variables is the total number of animated variables in the system. #dimensions is the dimension of the dynamical system that is optimized. It is not the same as vars because each ride consists of several independent systems, some variables are modeled as closed-form functions of time, and the rate of change of some variables may add additional dimensions. If the system is periodic, the period is given. 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.*

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 simulation for a ride was 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 complete model for the system. The geometric rendering was not affected by the culling – we assume the browser was culling geometry against the view volume.

Table 5.2 presents the timing values recorded in the experiments. The 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 5.12 plots the speedups obtained by culling over a world that does no culling. The speedups obtained are around 2.6, and remain

Total	Cull On	Cull Off	# In View	% in View
7	0.048s	0.126s	1.42	20%
14	0.090s	0.249s	2.94	21%
21	0.142s	0.394s	4.75	23%
28	0.208	0.550s	7.27	26%
35	0.280	0.727	9.12	26%

Table 5.2: *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 (figure 5.12), as expected with our set up.*

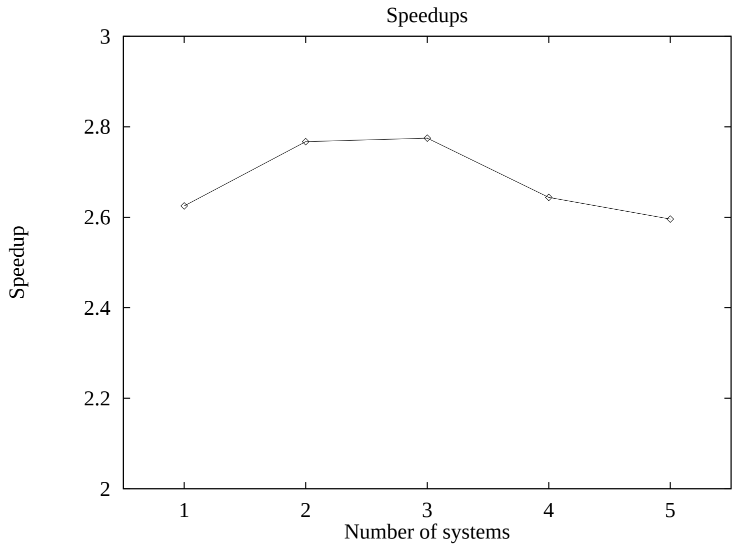


Figure 5.12: *The speedups achieved for the fairground rides with a culling strategy compared to the complete simulation, plotted as a function of the total number of models in the world. We expect to see a constant speedup in the constant volume, increasing density environments with which we experimented. The variation is probably due to extraneous factors including frame rate dependent differences and variations in the set of visible objects over time between different environments. Note that the vertical axis does not start at zero.*

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 $4\times$ speedup one might expect (only around $1/4$ of the systems were in view at any time). This is due in part to the fact that our underlying system must perform some computation to check whether each system is in view, as well as the overhead of Java scripts.

5.6 Discussion

One significant extension to our approximation 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 run 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 complete 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.

The work described in this chapter makes two significant assumptions about the simulation of interest:

- The entire simulation can be broken into independent systems each of low dimension. For instance, the entire fun park is broken into individual rides. The independence assumption implies that the behavior of a system cannot be influenced by what a viewer sees while it is out of view, and we need only consider the previous seen state of the system when generating models for culling.

- It is possible to bound the global extent of each sub-system for all time. The Tilt-A-Whirl's motion, for example, is confined to within the region covered by its base. This restriction meant that we did not have to address the completeness problem.

The next chapter describes a world in which these assumptions no longer hold, but in which a successful culling strategy is still possible.

Chapter 6

Completeness

The completeness problem is that of identifying all the potentially visible objects for a given frame, which in turn ensures that the viewer sees everything they should see. As discussed in chapter 4, one example arises on a virtual race track where the viewer can only see one turn. If a racer is seen crossing the line, and subsequently goes out of view, after some period of time the viewer expects to see the car re-enter the view from the other side, having gone around the track. To address the completeness problem we require a way to make sure the car re-enters the view, after the right time, but without doing a significant amount of work while it is out of view. Completeness is not just concerned with objects the viewer has seen before: in a virtual city the viewer expects to see cars entering the view from side streets, even if they have never seen those cars before.

As a specific case study we concentrate on a series of large scale environments populated by cars that move through the maze-like streets of old cities (designed before grid layouts became popular). Such cities were chosen as an example because:

- Visibility is relatively easy to determine for the cluttered streets of a city. The cities we use lack the regular, long line-of-sight plan that modern cities exhibit, which means that on any given frame we expect a relatively small but highly variable number of roads to be potentially visible.

- The cars in the city are examples of wide ranging dynamic objects with significant group interactions. The culling algorithm must capture the expected interactions between cars while they are out of view.

- The city models may be automatically generated, making it easy to generate multiple cities of different sizes but with similar statistical properties, such as the expected number of roads visible from a given location. We use a variety of city sizes in our experiments.

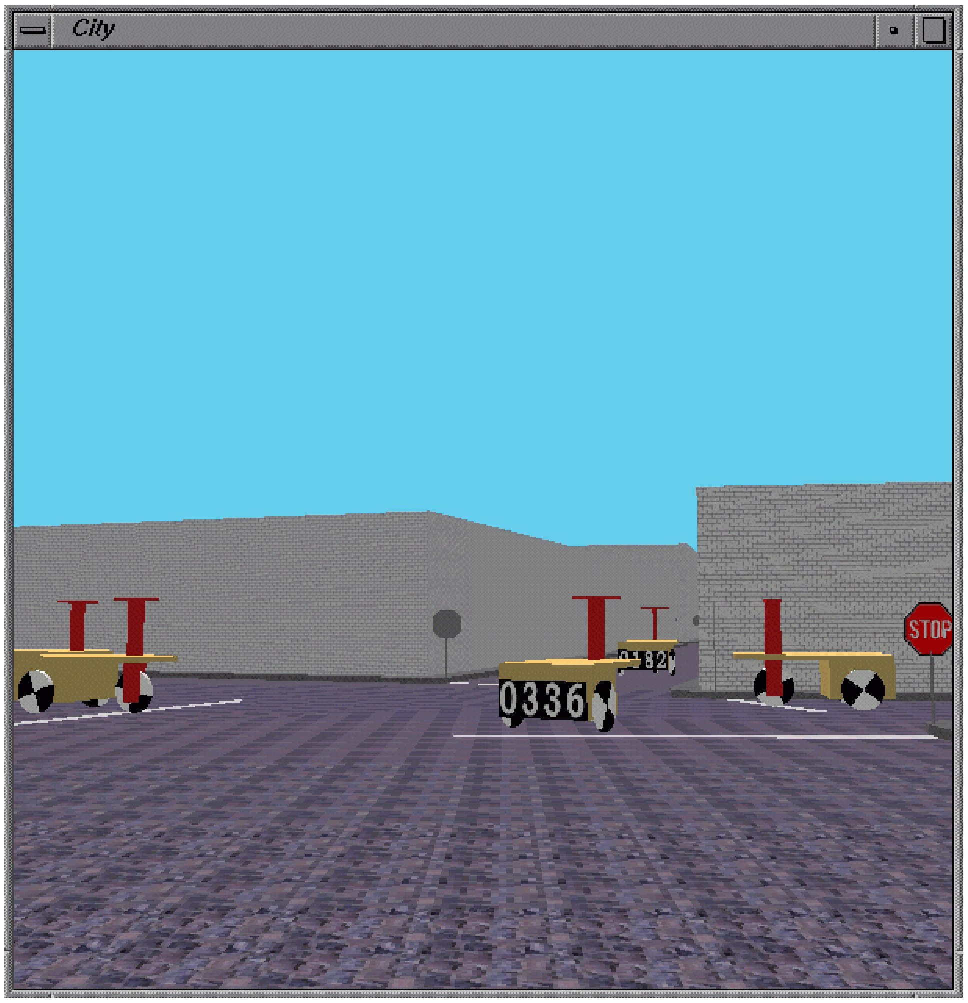


Figure 6.1: *A screen snapshot showing the intersection of several streets in a city simulation. Each car, labeled with a unique identifier, stops when it reaches the intersection and waits its turn before passing through.*

An screen snapshot for an example simulation is shown in figure 6.1. Complete details of the model are given in appendix B. In summary, the important features of the city are:

- The road network is a directed graph, with nodes in the graph representing intersections and edges representing lanes along which cars may drive.

- Cars perform random walk on the graph, traveling along each lane below a maximum speed and making random choices as to their next lane each time an intersection is reached.
- Cars avoid collisions by two means:
  - Cars slow down to maintain a safe braking distance behind cars in front.
  - Cars wait in turn to pass through each intersection, with the invariant that only one car is in any intersection at any time.

  The collision avoidance mechanisms introduce invariants into the model that a culling strategy should attempt to maintain, and it leads to uncertainty in the time taken by a given car to traverse a lane.
- Visibility is relatively easy to compute in the city, and we can efficiently track car visibility over time.

For any general large environment, the objects in the world may be qualitatively divided into three sets, depending on how recently they have been seen by the viewer, and how memorable or predictable each object is. We are interested in how difficult it is to determine whether or not these objects are visible on the next frame:

- **Visible objects** for this frame. The cost of determining the location of these objects for the next frame is small – their equations of motion must be solved for one frame interval, and even if an object leaves the visible region on this frame and its state is not required, we have only done one extra frame's worth of work. In the city model, these cars are identified by the visibility system using standard technology.
- **Previously seen** objects that weren't visible on this frame. These objects may be visible on the next frame, but it is unnecessary to compute the location of them all, because most of them are unlikely to be visible, and the work will be wasted. Instead, the aim is to infrequently update the potential locations of these objects, and use data structures that can report the potentially visible objects for the next frame without examining all the objects. In the city model, we include all the non-visible cars in this category.
- **Never before seen** objects that may be visible on the next frame. The experiments described in this chapter address such objects only for the first frame, when every object is

assigned an initial position and we assume that the viewer is aware of all those positions. Section 6.3 discusses ways to more efficiently manage these objects.

This chapter describes an algorithm that efficiently determines which objects are within the visible set for each frame, and we discuss some specific experiments with the city environment. The chapter concludes with a discussion of the benefits and inadequacies of the current algorithm, and discusses how it may be improved.

6.1 A Completeness Algorithm

As suggested above, the difficult aspect of a completeness algorithm is knowing when a previously seen object should re-enter the view. The solution presented here exploits spatio-temporal bounds, in principle similar to those described in chapter 3. In both cases, the bounds provide a means of excluding objects from consideration over some period of time.

The simulation maintains the set of objects potentially visible to the viewer, the *visible set*. Objects in the visible set are updated on each frame using the simulation equations of motion, restricted to objects in the set (objects outside the set are assumed not to exist). If on any frame a visible object leaves the visible region it is removed from the visible set. In the city model, the equations of motion for the visible cars relate their position, speed and orientation to the path they are currently traveling and their accelerations (section B.2).

Every object not in the visible set bounds the possible extent of its motion for some future time interval, after which the bound expires. If, on any frame, the viewer sees one of the possible locations for the object (sees its bound), the object is forced to decide whether or not it is actually in a visible location. If it is, it enters the visible set and has its state updated using some consistent method (once we know there the object is, it is a consistency problem to decide its state). The system also maintains a priority queue of expiration times for bounds, and on each frame any expired bounds are recomputed.

In the city model, bounds are computed by performing a breadth first search to locate roads the car might reach within a given time. When a certain number of roads have been found, the search terminates. The roads found form the bound, and the earliest time that the car can leave the bound is it's expiration time (see section B.3.2 for details).

If the bound for a non-visible car is seen by the viewer, the car determines its actual location using a sampling approach. In a preprocessing step, the distribution of travel times for each road is determined. When choosing a location, each car begins at its last known position, and performs a random walk using sampled travel times as the time taken to traverse each road in the walk. When the total travel time used in the walk reaches the current time, the car's location is known. Rejection sampling is used to ensure that the invariants of the simulation are not violated. There are some additional details which are discussed in section B.3.3 of appendix B.

The sampling technique used in the culling city model attempts to capture the influence of out of view car-to-car interactions without explicitly modeling them. In the full simulation, the actual time taken for a car to reach any point depends on how frequently it is slowed by other cars, and how much time it spends waiting at intersections. By independently sampling the location of each car, we are not tracking these interaction explicitly. However, the distributions from which we sample travel times to model the expected waiting time due to the interactions, so the viewer ultimately experiences correct travel times over many sightings of the car.

Define O as the set of all objects, V as the set of visible objects, and o as an object. Define $\mathsf{minkey}(Q)$ to return the key of minimum entry in the priority queue of expiration times, Q, and $\mathsf{min}(Q)$ to delete from the queue and return the object whose bound expires at that time. Finally, assume that S is the visible region of space. Overall, the following operations are performed for every frame:

Algorithm 6.1 An algorithm for completeness: per frame operations

```
simulateFrame(carset: $O$, carset: $V$, queue: $Q$, region: $S$)
    update($V$)
    for all $o \in V$
        if $\neg$visible($o$)
            $V \leftarrow V \setminus o$
            newBound($o, Q$)
    while minkey($Q$) $< t_{frame}$
        $o \leftarrow$ min($Q$)
        newBound($o, Q$)
    for all $o \in$ visibleBounds($O \setminus V, S$)
        delete($o, Q$)
        setPosition($o$)
        if visible($o$)
            setState($o$)
            $V \leftarrow V \cup \{o\}$
        else
            newBound($o, Q$)
```

The simulation is initialized as follows:

Algorithm 6.2 An algorithm for completeness: initialization

```
cullingInitialize(carset: $O$, queue: $Q$)
    $V = \emptyset$
    for all $o \in O$
        sampleLocation($o$)
        newBound($o, Q$)
```

In addition to unordered set and priority queue implementations, the above code requires several application dependent components to perform the following functions:

update(V): Update the state of the visible objects using the complete model, assuming that non-visible objects do not exist.

visible(o): Return true if an object may be visible, given its position. See sections B.1.1 and B.2.1 for details in the city model.

newBound(o, Q): Bound an object's future locations, given its last known position, compute when that bound expires, and insert the object into the queue of expiration times keyed on its expiration time. For the city model, see section B.3.2.

visibleBounds$(O \setminus V, S)$**:** Find the set of objects with visible bounds (section B.3.1).

setPosition(o)**:** Find the actual location of an object, given its last known position (section B.3.3).

setState(o)**:** Set the state of an object after some period out of view (solve the consistency problem). See section B.3.4 for the city model.

sampleLocation(o)**:** Sample an initial position for an object (section B.3.5).

Section B.3 of appendix B describes each component in detail for the city model.

6.2 Experiments on the City

To investigate the efficiency of the city models we ran two sets of experiments, one involving a constant city size and increasing numbers of cars, and another with a constant density of cars in variable sized cities. We begin with a look at estimating the efficiency of a culling model.

6.2.1 Analyzing Culling Efficiency

We define the efficiency, η, of a culling algorithm as the ratio of the amount of work done for state in view, C_v, to the total amount of work done for the simulation C_t. The ideal culling algorithm has an efficiency of one, because it only does work for systems in view. We can determine the efficiency of the city models for a given city size and number of cars as follows:

$$\eta = \frac{C_v}{C_t} \tag{6.1}$$

$$= \frac{\alpha n_v}{\alpha n_v + \beta (n_t - n_v)} \tag{6.2}$$

where α and β are constants reflecting the complexity of simulating a single car while it is in and out of view respectively, n_t is the total number of cars and n_v is the number of cars in view. The speedup, s, obtained by a culling algorithm with cost $C_{culling}$ over a complete simulation with cost $C_{complete}$ is defined as:

$$\begin{aligned} s &= \frac{C_{complete}}{C_{culling}} && (6.3)\\ &= \frac{\alpha n_t}{\alpha n_v + \beta\left(n_t - n_v\right)} && (6.4) \end{aligned}$$

and hence

$$\frac{\alpha n_t}{s} = \alpha n_v + \beta\left(n_t - n_v\right) \tag{6.5}$$

Combining equations 6.2 and 6.5, the efficiency when culling is:

$$\eta = \frac{n_v}{n_t} s \tag{6.6}$$

The efficiency of the complete algorithm is:

$$\begin{aligned} \eta_{complete} &= \frac{C_v}{C_t}\\ &= \frac{\alpha n_v}{\alpha n_t}\\ &= \frac{n_v}{n_t} \end{aligned}$$

6.2.2 Common City, Variable Density

The city used for this experiment contains 199 roads and 1764 paths. Its layout is pictured in figure 6.2. We built simulation models for this city containing from 50 to 200 cars in 25 car increments. To gather data, a fixed viewer trajectory was recorded for fifteen minutes of simulation time, and then played back at a constant ten frames per second for each model.

The efficiency of our algorithm for a common city and increasing density of cars is shown in figure 6.3. The graph is based on data in table 6.1.

The efficiency varies from 86% to 96%, rising as the density of cars increases. For the largest city then, 96% of the work is done for cars the viewer potentially can see, and only 4% goes to cars that aren't seen. The rise in efficiency as the density of cars increases is due to the fact that accurate simulation of denser traffic is more expensive compared to light traffic, while the tracking of cars out of view is unaffected by the density. So as the density increases the culling algorithm spends proportionally more time working on the cars in view, leading to higher efficiencies.

Note that for the particular computer used here, an HP J9000, the non-culling simulation cannot simulate more than 150 cars in real time (even without spending any time on

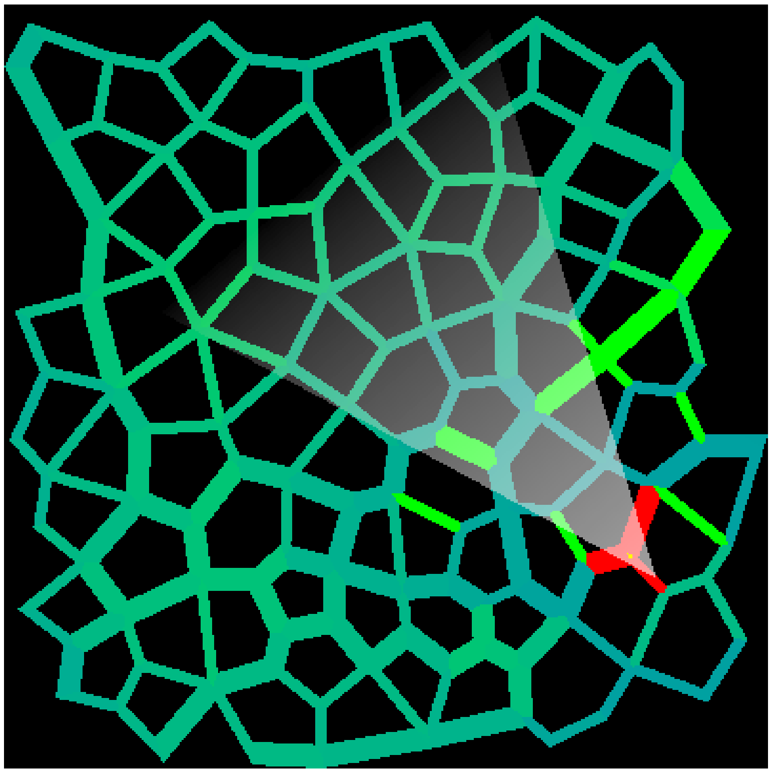


Figure 6.2: *The common city model used for the experiments with a variable number of cars. The viewer is located at the apex of the viewing frustum (the transparent triangle), and the red cells are potentially visible from that location. The other roads are shaded according to how many cars include each road in their bounds, weighted by the size of each car's bound. Bright green roads may have many cars on them, while cyan cells probably have no cars.*

n_t	n_v	$t_{complete}$ (s)	t_{cull} (s)	s	η
50	1.88	0.037	0.0016	22.94	0.86
75	3.17	0.055	0.0026	21.09	0.89
100	4.42	0.073	0.0036	20.50	0.91
125	5.87	0.090	0.0046	19.53	0.92
150	5.96	0.108	0.0048	22.82	0.91
175	7.40	0.126	0.0058	21.68	0.92
200	8.75	0.144	0.0066	21.87	0.96

Table 6.1: *Data for culling in a common city with a variable number of cars. n_t is the total number of cars in the city, while n_v is the number of cars in the visible set, averaged over all the frames. $t_{complete}$ and t_{cull} are the times spent performing dynamics calculations per frame, for non-culling and culling respectively, averaged over all the frames. Note that the simulation was running at ten frames per second, so 0.1 seconds per frame is the real-time limit. s is the speedup achieved by culling, and η is the efficiency of the culling algorithm, calculated using formula 6.6. These results were obtained on a HP J9000 with an Evans and Sutherland Freedom graphics engine.*

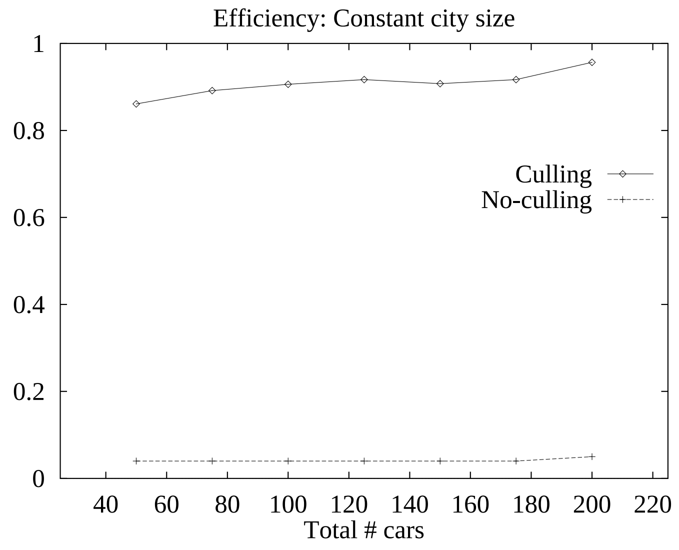


Figure 6.3: *The efficiencies of both the culling and non-culling simulations for a common city (figure 6.2) and variable number of cars. The efficiency of the culling algorithm rises as the number of cars, and hence density, increases. This is because the full simulation of denser traffic is more expensive, so the culling algorithm performs more work fully simulating the cars in view, while the workload for cars out of view is unaffected. Hence the efficiency rises. The non-culling version sees roughly constant efficiency, which is expected because the ratio of cars in view to cars out of view stays constant. In the 200 car city, 96% of all the work done by the culling version is for potentially visible cars. Only 4% of the work is toward potentially visible cars in the non-culling model.*

n_t	n_v	$t_{complete}$ (s)	t_{cull} (s)	s	η
150	6.13	0.109	0.00506	21.46	0.88
300	6.00	0.217*	0.00540	40.17	0.80
600	6.09	0.434*	0.00639	67.94	0.69
1200	6.31	0.868*	0.00890	97.46	0.51
2400	5.96	1.736*	0.01223	141.92	0.35

$*$: extrapolated

Table 6.2: *Data for culling with a constant density of cars in various city sizes. n_t is the total number of cars in the city, while n_v is the number of cars in the visible set, averaged over all the frames. $t_{complete}$ is the estimated simulation time per frame for a complete simulation running at ten frames per second. The data for $n_t = 150$ was obtained experimentally in the constant city size experiments. It would take prohibitively long to run complete simulations for the other cities, so we extrapolated the cost assuming a constant cost per car. t_{cull} is the time spent performing dynamics calculations per frame for the culling simulation. Note that the culling simulation is real-time at ten frames per second up to around 2000 cars in the city. s is the estimated speedup achieved by culling, and η is the efficiency of the culling algorithm.*

rendering), while the culling algorithm poses no difficulty even with 200 cars. In other words, none of the cities in the following example can be simulated on the computer in question in real time without culling. It is unlikely that any current desktop machine can simulate cities with thousands of cars, as we achieve with culling.

6.2.3 Constant Density, Variable Cities

In this experiment we examine simulations containing from 150 to 2400 cars, in cities sized to keep the density of cars roughly constant. All the cities used are shown in figure 6.4. The smallest is that used above, with 199 roads, while the largest contains 3820 roads and 29200 paths.

The efficiency of culling in these cities is plotted in figure 6.5, and the data is tabulated in table 6.2.

Note that it would take prohibitively long to simulate a complete version of these models. So we estimate the cost of the complete cities by assuming the per-car cost of the simulation is the same as that for the 150 car simulation (for which we have complete data), and multiply by the total number of cars in the other cities to estimate the cost of the complete simulations.

The culling simulation's efficiency falls as the number of cars and size of the city increases. This is to be expected, because while the number of cars in view stays roughly

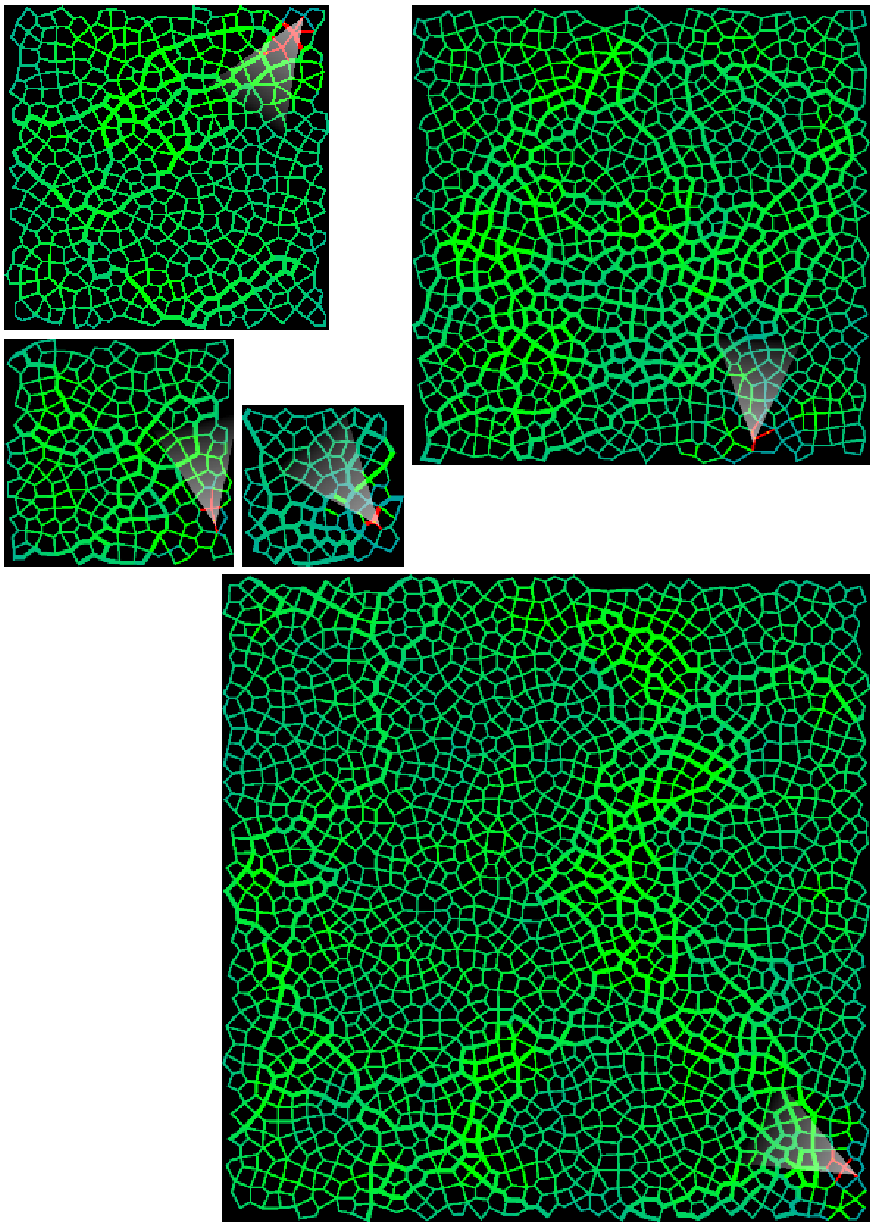


Figure 6.4: *The five city maps used for the constant density experiments. Each city has an area twice that of the previous, and we double the number of cars in each experiment to maintain a constant density. The smallest city is the same as that used for the constant city size experiments. The color semantics are the same as those of figure 6.2.*

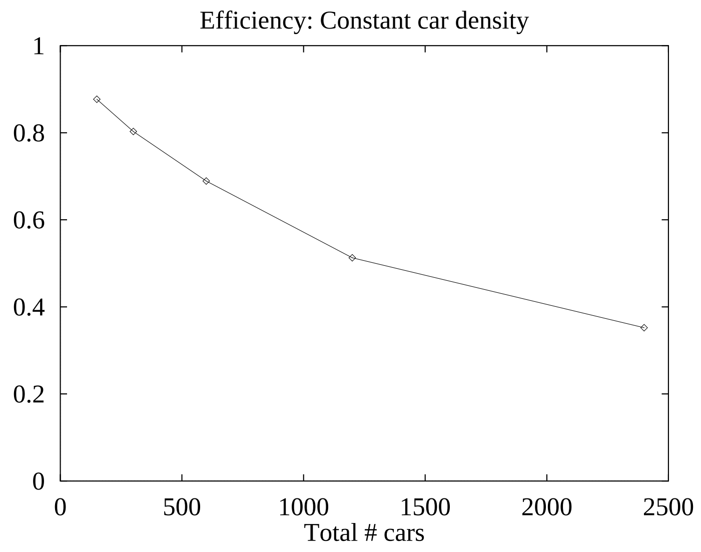


Figure 6.5: *The efficiency of our culling algorithm for various city sizes with a constant car density. The cities are shown in figure 6.4. The efficiency falls as the city size grows larger, because the number of cars in view stays roughly the same, but the number out of view grows. Hence the simulation does the same amount of work for cars in view, but does more work to track the greater number of cars out of the view. Different models are required to achieve greater scalability. In particular, it must be possible to do no work at all for most of the objects not in view.*

constant, and hence requires the same amount of work to simulate, the number of cars out of view grows, requiring more time to be spent tracking out of view cars. Despite the reduced efficiency, we can simulate ten times as many cars in real time with culling than we could without. The next section discusses how high efficiency might be achieved for larger models, allowing the number of moving objects to increase arbitrarily.

6.3 Discussion

The city model described in this chapter and appendix B offers significant speedups over the complete simulation and does reasonably well in capturing most of the behaviors exhibited by cars in the city. Empirical observations suggest that, among other things, car densities and travel times are comparable to those in the complete model, and only infrequently does it become apparent that the model is performing culling. Such lapses in quality are just one of many areas of potential improvement that we discuss here. The following section looks at improvements in efficiency, both at run-time and preprocessing, and potential improvements in quality. Finally, we briefly discuss how culling may extend to different simulation models, such as less random, more planned motion.

6.3.1 Optimal Bounds

The bounding algorithm for the city model uses breadth first search and a fixed maximum size for the bound, chosen largely arbitrarily for our experiments. We would prefer to analytically or experimentally determine an optimal bound size based on properties of the model. We might also consider entirely different motion bounding strategies.

A bounding volume strategy affects the efficiency of the culling simulation in three potentially conflicting ways:

- The size of the bound influences how likely it is that the viewer will see the bound, and hence how likely it is that a moving object will be updated. Smaller bounds are better in this respect, because they lead to fewer updates due to visibility and greater efficiency.

- The lifetime of the bound influences the number of updates. Long bound lifetimes (and hence larger bounds) are preferable, because they need to be updated due to expiration less frequently.

- The cost of determining the location of an object within its bound, and the cost of computing the bound itself determine how expensive it is to update the object when it is potentially visible. Cheaper bounds are better.

Any technique to optimize the generation of bounding volumes must balance the tradeoffs between the above concerns.

Analytic optimization requires some model for each of the above effects. The most difficult thing to capture appears to be the likelihood of a viewer seeing a bound, because it depends on the viewer's motion, the relationship between bounds and visibility, and inter-frame coherence. An analytic approach would also require models for the cost of computing bounds and the expected lifetime of the bounds, which appear to be easier to develop. Hence, while possible, a good analytic model seems difficult to obtain.

Experimental optimization appears a more promising approach at this point. The simulation may be run with different bounding strategies and the best selected. The individual experiments may not be particularly expensive, because they use the culling model and do not necessarily require new preprocessing steps for each candidate approach.

6.3.2 Probabilistic Bounds

The bounds used for the city model are conservative — the car is certain never to get out of the bound before the expiration time. However, conservative bounds are not strictly necessary for culling. Non-conservative bounds may be appropriate if, for instance, only half of the conservative bound contains 99% of the probable locations for the car. It is more efficient to use only half the bound, because it may be faster to compute and a viewer may be less likely to see it. But if we do so, there is a 1% chance that a car position we sample when the bound does become visible will turn out to be somewhere we said it could not be. We must reject this sample, because it may cause the viewer to see a car appearing in the middle of nowhere. While rejecting the sample introduces an error (cars will never appear in 1% of the places the really could be in), we know the error only occurs 1% of the time, so may be acceptable.

This is a clear tradeoff between simulation efficiency and quality. We see better efficiency with non-conservative bounds, but they lead to differences between the culling and complete models. Ideally, we would like to quantify such tradeoffs and use them in some way. For instance, we might be able to provide the best quality simulation given a fixed computational budget.

6.3.3 Reducing Storage Requirements

Each path in the current model has associated with it various approximations and distributions for culling. This is a very large amount of data for big environments, and clearly doesn't scale indefinitely (we are trading off space for speed as we scale the environment). A parametric model is required that describes each path by a few parameters, such as length and the number of other paths into and out of it. The various approximations and distributions could then also be parameterized, resulting in a more compact city model.

Devising good parameterizations has the flavor of a machine learning problem. We wish to devise models that are cheap to store and evaluate yet provide sufficient flexibility to capture the required effects, such as the expected travel time along a road. Given the model, we would then apply fitting or learning techniques to determine the optimal model parameters for each individual road.

6.3.4 Improving Scalability

The current city models still track all the objects that may ever appear in the simulation. This is equivalent to saying that the viewer always remembers the last place they saw a car, and can extrapolate that information into the future. That is clearly not reasonable. In real cities the vast majority of cars a viewer sees are immediately forgotten, and they will never be seen again. This fact should be exploited to gain higher simulation efficiency.

One way to exploit this is to cease tracking objects that have not been seen in a long time, and add them to a general pool of objects that might be anywhere in space. To add objects out of the pool back into the simulation, the model would include distributions for the expected density of cars in various places. If a viewer can see a region, and the current density of cars is unlikely, then cars would be added from the pool to make the world more plausible. Several extension to this model are possible, including:

- Models for the density of objects that take into account what the viewer has already seen in neighboring regions.
- Spatially local pools of cars so that cars could be relegated to the pool earlier.
- Generic cars with state that is created and deleted as needed, rather than stored over time in a pool.

The idea of pools of cars introduces a hierarchy of simulation models. Some objects are simulated fully, some are tracked statistically, and some simply belong to pools.

6.3.5 Placing cars

The task of placing cars is equivalent to that of sampling from the joint distribution on the locations of the cars that must be placed conditioned by the locations of the cars that the viewer saw on the last frame. The joint distribution encodes the various simulation invariants by assigning zero probability to states which violated the invariants. In chapter 2 we use the Markov chain Monte Carlo algorithm to sample from such distributions. In this case there are two obstacles to using MCMC: the sampling must be very fast, and it is difficult to formulate a distribution from which we wish to sample. In particular, we would require at least a way to evaluate the probability that a car is in a particular position given its previous position and how long it has been traveling. These distributions are difficult to compute rapidly due to their dependence on the previous position of the car and all the possible paths for the car to the position of interest.

There remain significant problems with the alternative approach that we use, placing cars individually and rejecting locations that violate the invariants. The greatest problem is that sometimes cars cannot be placed, and we must manipulate time to put them down somewhere. In these cases there is clearly the possibility that a car will end up somewhere it really should not be. The root of this problem is that cars positioned earlier may take the only reasonable spots for a later car, leaving it nowhere to go. Rather than adjusting time for the car we cannot place, a better solution is to move cars that were placed earlier. One possible approach is to place all the cars regardless of the invariants, and then apply a randomized relaxation procedure to shift cars into better locations.

6.3.6 Other Quality Criteria

It may be possible to improve the quality of the culled simulation by capturing more effects in the statistical models, or using different models entirely. For instance, the current simulation does not attempt to capture possible correlations between the travel times of different cars. For example, if a sampled travel time for one car implies it took a long time to traverse an intersection, then other cars that must traverse the same intersection should also take longer. However, it is not clear that a viewer could ever detect the correlation, or lack of it in the culled model. Hence we require a greater understanding of which behaviors are important before implementing more complex models.

Finding important behaviors is related to the problem of locating patterns in large amounts of data, or data mining. Here, the data come from running the complete simulation model, and we are looking for patterns in the behavior of the objects that may be important to a viewer's experience of the virtual world.

The city simulation sometimes exhibits extreme behaviors, such as gridlock. When this occurs, cars cannot make any progress, and in fact the deadlocked region grows over time as more cars get stuck waiting for intersections to clear. When gridlock arises in the fully simulated model there is no way to resolve it (other models could be designed to resolve deadlock). However, in the culled model it is possible to resolve deadlock by walking away from it for a while. The statistical models for out of view travel times do not explicitly account for the other cars, and so cars can jump out of the gridlock. Hence, the resolution of gridlock is one apparent difference between the complete and culled simulations, yet is an extreme event.

The failure to capture extreme events is a modeling issue. If an author was concerned with such effects, they would need to use models for out of view motion that captured the desired behaviors. This highlights the broad point of culling: efficient culling strategies are designed to capture those behaviors an author cares about and to ignore unimportant effects.

6.3.7 Culling Other Simulations

The cars in the city model perform purely random walk, which is not a realistic model for most situations. The randomness has two primary consequences:

- Bounds are cheap to compute, but inefficient, in that the car will never actually visit most of the places it could visit.

- All the interactions between cars are direct and localized.

Other simulations will not exhibit these properties, such as a simulation in which each car has a specific destination and looks for an optimal path given the expected traffic conditions. In such a simulation it is harder to bound the future motion of the cars, because it depends on all the other cars. Also, interactions are not localized because the car may look ahead to determine a good path, which may lead to stronger correlations between the behavior of various objects in the simulation.

Such a simulation will most likely require a more complex model for motion out of view, at least for cars that were recently seen. On the other hand, the bounds for future motion in such a model can probably be made tighter, because regardless of how bad the traffic is, there are still only a few good, and hence likely, paths. Tighter bounds may offset the additional cost of computing them, because they are less likely to be seen by a viewer.

Currently, there is no model for analyzing the tradeoffs between quality and culling. Clearly, we are trading off some quality in the city model for very large computational savings (by, for instance, ignoring correlations between cars and using independent car sampling models). Ideally, for a given complete simulation model and a smooth range of quality criteria, we would like to be able to choose a cost and determine what quality could be achieved, or describe a way to optimize the tradeoff. The overall approach might be similar to Funkhouser's level-of-detail techniques [30] for large static environments.

Chapter 7

Conclusion

The work in this thesis represents a step toward directable and scalable dynamic virtual environments, such as the virtual city discussed in the introduction. From the direction side of the problem, we have described a Markov chain Monte Carlo sampling approach to generate multiple constrained animations, each of which is consistent with a physical model. The MCMC algorithm can manage discontinuous systems and those with extreme sensitivity to parameters, such a the collision intensive multi-body systems that we demonstrate. A new asynchronous rigid-body simulation algorithm was also developed. The algorithm offers large speedups over previous approaches and begins to explore issues surrounding simulation re-use.

On the scalability front, we have introduced the basic ideas of simulation culling, and demonstrated its application to a range of models. Culling has enabled the design of large scale, highly dynamic virtual environments that can be explored in real-time.

Open problems and areas for future development have been described throughout this thesis on the topics of direction, basic simulation and scalability. Yet to create large controllable and scalable environments, several integration and modeling issues must be addressed, as we discuss next.

7.1 Integrating Control and Culling

The work in this thesis describes direction, and it describes scalability, but it does not integrate both into a single system. In principle, the algorithms we have described fit very well together, because both may be treated as the same problem of sampling from complex conditional distributions.

Consider our approach to culling, where the state of objects is modeled probabilistically if the viewer has not seen the object for some time. As discussed in chapter 4, the problem of presenting a plausible and consistent image to the viewer can be posed as a sampling problem: sample, from the marginal distribution of objects in view, new states for all the objects not recently seen, subject to the viewer's prior knowledge and the constraints of the simulation.

To integrate direction into this scheme, we may add additional conditions to the sampling problem, encoding the outcomes we wish to see. We then aim to sample new states for previously unseen cars that are not only consistent with the viewer's previous knowledge, but also lead to the desired outcome. Sampling problems of that form are exactly the type addressed by the MCMC direction algorithm of chapter 2.

Three open problems must be solved in order to integrate direction and scalable simulation in a real-time system:

- The MCMC algorithm as it stands is not real-time on anything but the simplest problems. A different sampling strategy may be required, or it may be possible to design the constraints and simulation to make the sampling problem easier.

- Directing the simulation in an interactive system must account for the unpredictable motion of the viewer. A good direction strategy for staging specific events must be robust to uncertainty in a viewer's behavior over time. On the other hand, it also suggests that the system will not wish to look very far ahead, which may make the sampling problem easier.

- It is not clear how best to state the desired outcomes in a real-time system. Ideally the outcome would be expressed in terms of a few simulation parameters or generic objects. For instance, the goal for a red light runner is to find some car to run the red light in front of the viewer at some time in the near future. While this gives the system more opportunities to plan events, it also requires a method for choosing the best time, place and participants for the situation.

Solutions to these problems will likely draw on several sources, particularly work on robot control and planning.

The work on simulation culling in this thesis enables dynamic virtual environments that were beyond the scope of existing technology. Adding direction into these worlds will open up an array of important applications in the fields of training and entertainment. Simpler authoring tools will further speed the development of such environments and make them accessible to a wider user community. We might see virtual driver training on inexpensive systems for every teenager, or computer games capable of presenting complex virtual spaces in which to entertain. Yet, even if we are capable of producing research examples of such worlds, modeling issues may still stand in the way of widespread adoption, an issue we turn to next.

7.2 Modeling Large Environments

Much of the work in this thesis is aimed at very large scale environments. In the virtual city models we applied automated techniques to construct the geometry of the world according to a particular model. We then designed by hand a vehicle simulation for that world, and constructed a suitable culling model. However, this approach is not acceptable to a broader audience.

Many models will be required to capture real-world environments. Work is proceeding on the geometric aspects of this problem, attacking the problem either by constructing parametric models from acquired data [24, 21] or by recording images and re-rendering them as required [58]. Motion capture [33, 74] could be seen as achieving the same goal for motion, but the technology does not capture higher level behaviors, nor is it clear how to parameterize the motion in a useful way. Some work is proceeding on technology for modifying motion capture data in order to achieve specific goals [33, 34, 74], which may be a first step in automated modeling. However, it leaves open the problem of capturing higher level interaction and decision making behaviors.

Regardless of how a realistic simualtion model is acquired, we would like incorporate a culling strategy into that model. For instance, given a city with models for vehicles, people and trees, we might provide a partly interactive system in which a user is presented with the option of capturing or ignoring various aspects of the behavior. Based on their choices, a compiler-like system would then construct a model for culling. Such a system would most likely require significant human guidance, but the user should not be required to manage the low level details of distribution modeling and the like. The key problem is in determining the correct abstractions to utilize in order to make the system sufficiently expressive while keeping complexity down. Modeling on this scale is never likely to reach the simplicity of a word processor (there is no demand), but the technology should be made available to more than a few experts, and the only way to do that is with tools that take care of the details on the user's behalf.

Appendix A

Details of the Directing Simulation Examples

A.1 Bowling Details

A.1.1 Uncertainty model

Our bowling model is derived from online data (such as http://www.brittanica.com). The sources of uncertainty in the model are:

Ball radius Distributed uniformly on $[r_{min}, r_{max})$, where r_{max} is specified by the rules of the game (approximately 11cm) and $r_{min} = 0.8r_{max}$.

Ball density Distributed uniformly on $[\rho_{min}, \rho_{max})$, with $\rho_{max} = 1440\,\mathrm{kg\,m^{-3}}$ (estimated from the maximum allowed mass of the ball and the maximum size) and $\rho_{min} = 0.8\rho_{max}$.

Ball initial position Fixed in line with the end of the lane, at some point uniformly distributed across the width of the lane, and at a height distributed according to $\Phi(1.1r_{max}, 0.1r_{max})$, the distribution defined by the Gaussian density function with mean $1.1r_{max}$ and standard deviation $0.1r_{max}$.

Ball initial velocity The component down the lane (measured in $\mathrm{m\,s^{-1}}$) is distributed according to $\Phi(7.0, 1.0)$. The component across the lane is distributed according to $\Phi(0.0, 0.1)$. The vertical component is set to zero.

Ball initial angular velocity About a vertical axis (measured in $\mathrm{rad\,s^{-1}}$), distributed according to $\Phi(0.0, 0.2)$.

Each pin Fixed shape and mass, horizontally offset from its proper location on the lane in a random direction by a distance distributed according to $\Phi(0.0, 1.0)$ (mm).

With these distributions:

$$\begin{aligned} p_w(A) \quad &\propto \quad e^{-\frac{1}{2}\left(\frac{x_z-1.1r_{max}}{0.1r_{max}}\right)^2} e^{-\frac{1}{2}(v_x-7)^2} e^{-\frac{1}{2}\left(\frac{v_y}{0.1}\right)^2} \times \\ &\quad\;\; e^{-\frac{1}{2}\left(\frac{\omega_z}{0.2}\right)^2} \sum_{pins} e^{-\frac{1}{2}\left(\frac{d_i}{0.001}\right)^2} \end{aligned}$$

where x_z is the ball's height above the lane, v_x and v_y are the velocity components down and across the lane, ω_z is the angular velocity about the vertical axis, and d_i is the distance of pin i from its center location. The above formula for $p_w(A)$ is valid if all the uniformly distributed variables are within their range, and all the fixed variables have their correct values, otherwise $p_w(A) = 0$. We can ignore the uniformly distributed variables in computing $p_w(A)$ because their distribution function is proportional to one.

A.1.2 Proposals

Our proposal mechanism samples a value u, uniform on $[0, 1)$, and then:

- if $u < 0.05$, we sample new values for all the random variables.
- if $0.05 \leq u < 0.125$, we change the radius of the ball by adding an offset distributed according to $\Phi(0.0, 0.04r_{max})$. If the radius lies outside the allowable range, we wrap it back into the range.
- if $0.125 \leq u < 0.2$, we change the density of the ball by adding an offset distributed according to $\Phi(0.0, 0.04\rho_{max})$, wrapping to keep ρ inside the allowable range.
- if $0.2 \leq u < 0.4$, we change the initial position of the ball. We add a horizontal offset distributed according to $\Phi(0.0, 0.2w)$ (w the width of the lane), wrapping if necessary, and add a vertical offset distributed according to $\Phi(0.0, 0.05r_{max})$.
- if $0.4 \leq u < 0.6$, we change the initial velocity of the ball. To its component down the lane, we add an offset distributed according to $\Phi(0.0, 0.5)$. To its component across the lane, we add an offset distributed according to $\Phi(0.0, 0.05)$.

- if $0.6 \leq u < 0.8$, we change the initial angular velocity by adding an offset distributed according to $\Phi(0.0, 0.1)$.
- otherwise, for each pin, with probability $\frac{1}{2}$, change its location by moving it in a random direction by a distance distributed according to $\Phi(0.0, 0.5)$ (mm).

All of these proposals are symmetric, so there is no need to compute the transition probabilities (they cancel when computing the acceptance probability).

A.2 Spelling Ball Details

A.2.1 Model details

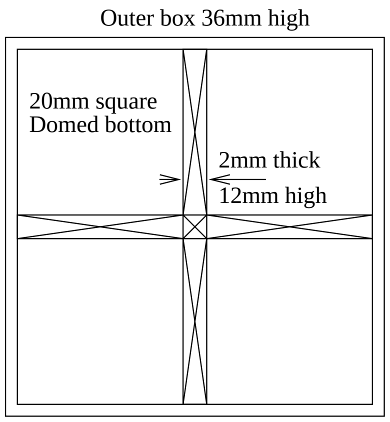


Figure A.1: *The dimensions and tessellation for the box in the spelling ball example.*

Each bin has a side length of 20mm, and each partition is 2mm thick and 12mm high (figure A.1). The floor beneath each bin is domed to help the balls come to rest, and the box in which the bins sit is 36mm deep. Each partition vertex is offset in a random direction by a distance distributed according to $\Phi(0.0, 0.1)$ (mm). Each ball is dropped from rest at a uniformly random location within a rectangle 72mm above the bottom of the box and centered above it. The size and density of the balls is intended to resemble marbles.

The probability of an animation is:

$$p(A) \propto \lambda^k \prod_{\mathcal{V}} e^{-\frac{1}{2}\left(\frac{\|\mathbf{x}_v\|}{0.1\text{mm}}\right)^2} \prod_{\mathcal{B}} \frac{1}{a}$$

where k is the number of designated bins that are filled, $\mathcal{V}$ is the set of partition vertices, $\mathbf{x}_v$ is the offset distance of vertex v (in mm), a is the area of the rectangular region from which the balls may be dropped and $\mathcal{B}$ is the set of active balls, which can vary in size for different animations as balls are activated or deactivated. In this case we must include a term for the uniformly distributed drop position of each ball because the number of balls can vary.

A.2.2 Proposals

Our proposal mechanism uniformly samples a random $u \in (0, 1]$, and then applies one of four strategies:

- if $u < 0.04$, we change all the partition vertices, giving them new randomly chosen offsets, and change all the balls, giving them a new active status and a new initial position.
- if $0.04 \leq u < 0.36$, for each partition vertex, we randomly decide, with probability 0.02, to change its location by adding an offset in a random direction with length distributed according to $\Phi(0.0, 0.05)$ (mm).
- if $0.36 \leq u < 0.68$ we uniformly randomly select an active ball to change, and offset its starting position in a random direction for a distance distributed according to $\Phi(0.0, s)$, where s is the bin size. We wrap the edges of the region from which balls may be dropped.
- if $0.68 \leq u < 0.84$, for each enabled ball we uniformly sample $v \in (0, 1]$ and disable the ball if $v < 0.25$.
- otherwise, for each disabled ball we uniformly sample $v \in (0, 1]$ and enable the ball if $v < 0.25$.

All but the last two proposals are symmetric. If a ball is disabled, the ratio $\frac{q(A_i|A_c)}{q(A_c|A_i)} = a$ (the area of the rectangle from which the ball may be dropped). If a ball is enabled, $\frac{q(A_i|A_c)}{q(A_c|A_i)} = \frac{1}{a}$.

Appendix B

The City Model Used for Culling

This appendix describes in detail the city model used for the culling experiments in chapter 6. We begin with the process of generating the geometric city model, before moving on to a description of the complete simulation model and the model used for culling.

B.1 City Modeling

The aim of the automatic city generation system is to create a roadmap resembling that of a city, and then create all the fixed geometry associated with the city (walls, sidewalks and signs). Additional information must also be associated with the map to assist with the car simulation, such as the number of lanes on a road.

City generation begins by sampling random points according to a Poisson point process of a fixed density in a rectangle. The Voronoi diagram for the points is computed, and then any points joined by a short edge in the diagram are merged to eliminate the edge. The diagram is cropped by removing any edges that pass outside the rectangle, and the remaining edges form the roads.

A width is then assigned to each road. To add variety, a subset of points are designated centers, and roads along the shortest path between nearby centers are made four lanes wide (two in each direction). All other roads are two lanes wide. The roads are thickened to their required width, then intersected to form a 2D map like that shown in figure B.1.

To complete the city, walls and sidewalks are extruded from the map, and stop lines and signs are added. Each road in the resulting city stores its geometry, its width, the location of the stop lines, its lanes, and a pointer to the intersections at either end, which themselves list

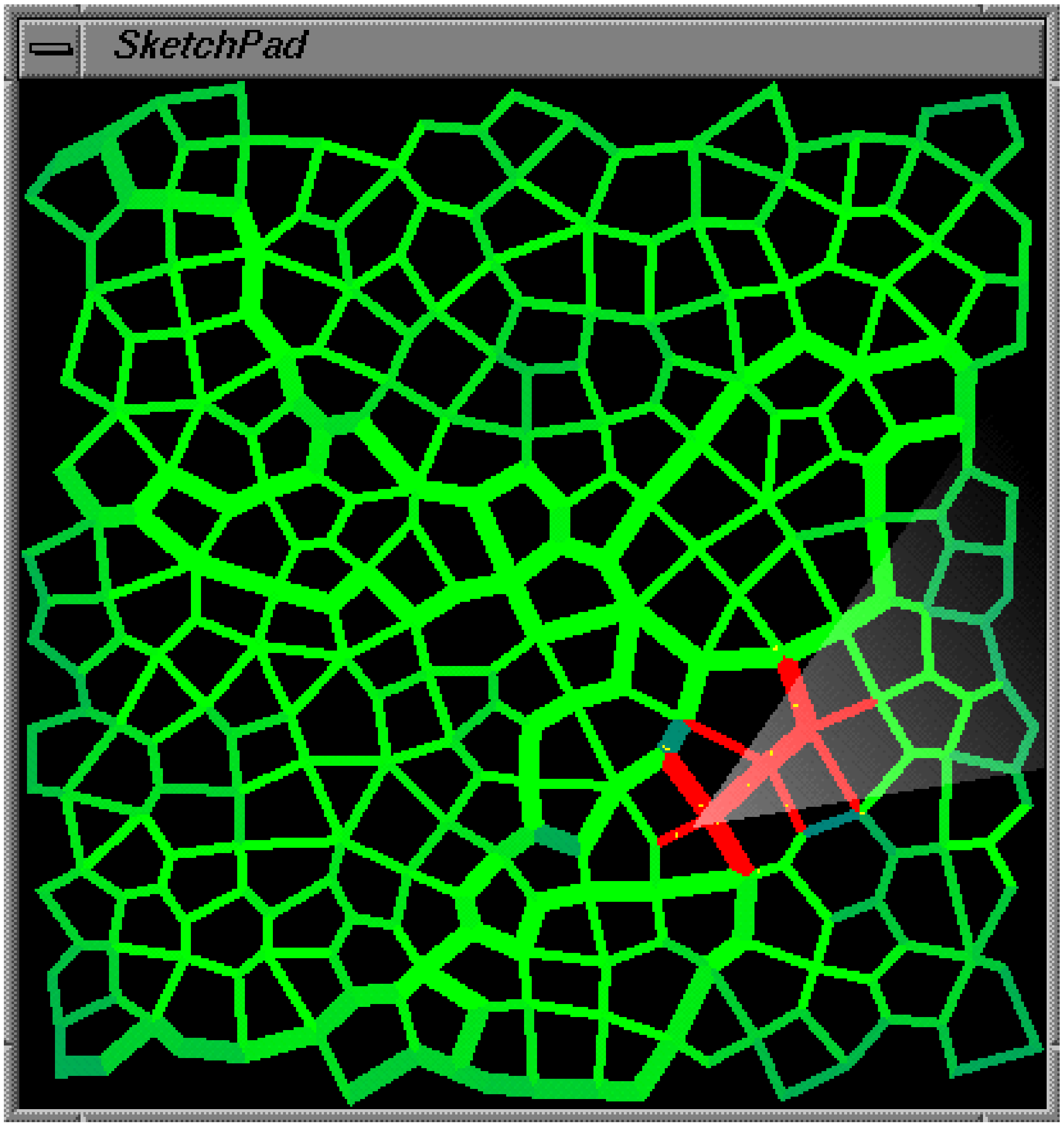


Figure B.1: *A map of a 446-road city. The red cells are potentially visible from the viewer's location (the apex of the transparent white triangle). The yellow spots indicate the position of potentially visible cars. In this image, cells are color coded according to how many cars have bounds including that cell, weighed by the total number of cells in each car's bound. Cells with more cars potentially on them are colored green. Blue cells have no cars on them.*

the roads that enter each intersection. The result is essentially an adjacency list representation of the underlying Voronoi diagram with geometry associated with each edge.

B.1.1 Visibility in the City

The geometry for a city must be rendered at interactive rates despite the large size of the model. As in many virtual environments, most of the world is not visible most of the time, in this case because the buildings and twisting roads combine to hide things behind buildings and around corners.

Each road in the city is considered a unit of visibility. The viewer's location is known, as is their gaze direction. Using standard technology, we walk the road map to locate visible roads, starting at the viewer's location. The following algorithm is used, called initially with the road, R that the viewer is on and the view frustum, V:

Algorithm B.1 Rendering the visible roads

```
renderRoad(road: $R$, view: $V$)
    draw($R$)
    $V' \leftarrow$ crop($V, R$)
    $R.fnum \leftarrow framenum$
    for all $R' \in$ adjacentRoads($R$)
        if intersect($R', V'$)
            renderRoad($R', V'$)
```

The crop(V, R) function crops the view volume to all the openings between the buildings on a road. adjacentRoads(R) returns the set of roads that meet a road, R. intersect(R, V) returns true if a road and a view frustum intersect (and hence the road is visible).

On each frame, the above algorithm is run to render the roads of the city before the simulation is called to compute state for and then render the cars. All the roads that are drawn are marked with the frame number to inform the simulation that the road is visible.

B.2 Car Dynamics

This section focuses on the basic car simulation model, assuming the complete simulation is running. Each car (a tricycle) consists of a body and a steering wheel (figure 6.1 on page 97). Each car performs a random walk on the road structure: it travels along the current

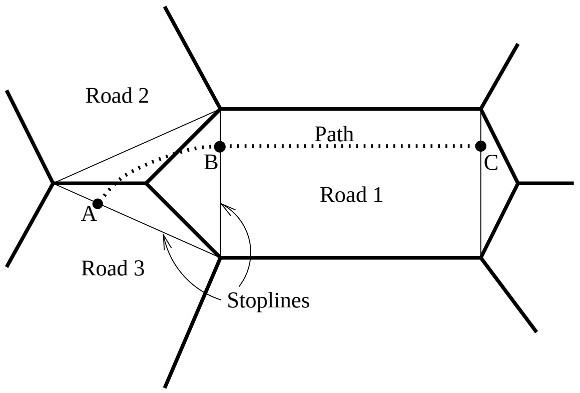


Figure B.2: *The configuration of a path. The path begins at the stop line at A, moves through the intersection to the stop line at B, and on down the lane to the line at C. Many paths will share the same lane BC, each coming from a different lane entering the intersection. The position of a car on the path is parameterized from 0 to 1.0 on the curve AB, and 1.0 to 2.0 for the straight section BC.*

road until it reaches an intersection, at which point it chooses at random a new road out of the intersection. Cars give way to other traffic as necessary, to avoid collisions both at intersections and due to running up the back of the car in front.

We begin by adding *paths* to the city structure. A path is a curve to follow from the an entry point of an intersection to the end of a lane out of that intersection (figure B.2). The position of a car is parameterized by the path it is on and a parameter, $u \in [0.0, 2.0)$, along that path. $u \in [0.0, 1.0)$ defines positions along a hermite curve through the intersection, while $u \in [1.0, 2.0)$ corresponds to positions on the straight section.

Each car's complete state consists of (figure B.3):

p	The path
u	Parameter value on the path
$\dot{u}$	Time derivative of u
o	Orientation of the body in world coordinates
$\mathbf{x}$	2D location in world space, derived directly from u
s	Steering angle, derived directly from u
w_1, w_2, w_3	Orientation of each wheel

Note that all the roads are flat, so $\mathbf{x}$ has two degrees of freedom and o has only one. According to the car design, s and the w_i are scalar quantities representing rotations about a fixed axis.

The acceleration of the center of mass of the car along the path is set directly to

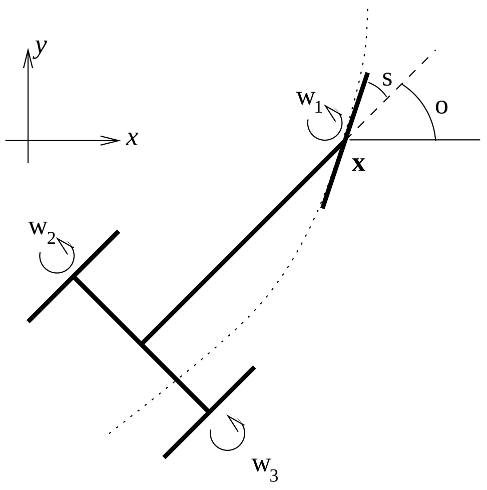


Figure B.3: *State variables for the car.* $\mathbf{x}$ *is the world location of the car.* o *is the orientation of the car body in world space.* s *is the steering angle offset from the body.* w_i *are the wheel orientations. Not shown is* u*, the position of the car along the (dotted) path, nor* $\dot{u}$*, the derivative of* u *with respect to time.*

control the car's motion, according to the following rules:

- if the distance to the car in front or the end of the path is less than the minimum stopping distance for the car, apply the maximum deceleration.
- otherwise, if the car's speed is higher than its maximum allowed speed, apply the maximum deceleration.
- otherwise, if the car is traveling at the maximum speed, apply no acceleration.
- otherwise, apply maximum acceleration.

In the absence of other cars, these rules cause the car to accelerate from zero speed at the start of the path until it reaches a maximum speed, then hold that speed until it approaches the end of the path, when it starts decelerating. If there is another vehicle in front, the car will slow down to avoid hitting it, which is the first source of interdependency between the motion of different cars.

The acceleration rules determine the evolution of the location parameter, u, and its time derivative, $\dot{u}$. The steering angle for the car is set to align the steering wheel with the tangent to the path at u. The remaining state variables, o and the w_i, are related to u and $\dot{u}$ by a set of differential equations that cause the body of the car to follow the steering wheel. All the differential equations are solved using an explicit Euler algorithm.

When a car reaches the end of its current path, it randomly selects another path out of the intersection it has just reached. But before the car is allowed to start along the new path, the intersection must be clear. If it isn't, the car is required to wait in a FIFO queue until its turn comes to pass through the intersection. This introduces the second source of interaction between different cars.

The inter-car interactions must be taken into account by any completeness algorithm for two reasons:

- They influence the motion of the cars, and must be taken into account when determining when and where a car should re-enter the view.
- They introduce simulation invariants. In particular:
  - Cars should remain a minimum distance apart, because as soon as cars get closer, the following car is slowed.
  - Only one car can be in an intersection at any given moment, because the others are forced to wait.

B.2.1 Car Visibility

If performing a simulation without culling, we still wish to avoid rendering cars that aren't visible. To achieve this, each road keeps a list of those cars that may be visible if the road is visible. Rendering all the cars on potentially visible roads then renders all potentially visible cars. The lists are kept up to date by noting visibility transition points: points at which a car's visibility may change (see below). Each time a car crosses such a point the visibility lists are updated appropriately.

To locate the potentially visible roads for the purpose of drawing their cars, we use a procedure analogous to renderRoad in section B.1.1, but rather than check for visibility against the view frustum, we just look at the frame number marking for each road. The road is visible only if its frame number matches the current frame.

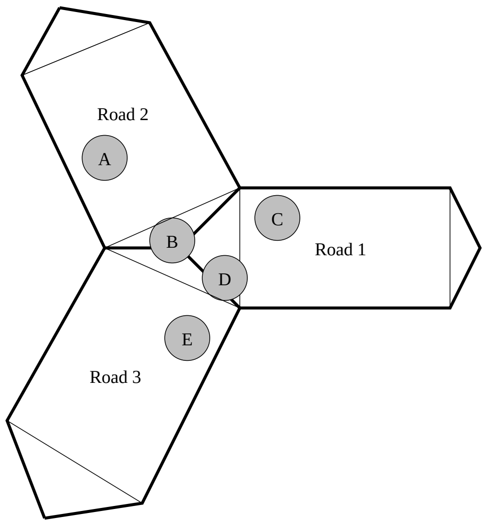


Figure B.4: *To rapidly determine visible cars we associate with each road all the cars that maybe visible if that road is visible, which are the cars whose geometric bound (a circle in this 2D picture) intersects the road. Road 1 has cars C and D associated with it. Road 2 has cars A and B, and road 3 has cars B, D and E. Note that each car can be on multiple roads as it straddles the boundary between them.*

Visibility Transition Points

Each car must be associated with all the roads that are intersected by the car's geometry (figure B.4). If that is the case, we can be certain of drawing all the necessary cars if we draw all the cars associated with visible roads. The visibility transition points are places where there is a change in the set of roads that are intersected by the car's geometry. We need only be conservative in determining intersections – it doesn't matter if we identify an intersection that doesn't exist, but it is a problem if we fail to find an intersection.

To speed up visibility computations, we specify a bounding sphere for each car centered on its origin and large enough to contain the car regardless of its orientation and internal configuration. A car is considered to be intersecting a road if its sphere is.

We could find all the parameter values for the car's location at which the intersections may change (it is the same as finding all the crossings between the paths and a polygon offset from the edges of the road). As a simpler approach, we conservatively identify transition points (figure B.5). If the car is between these points, it only intersects the current road, and hence is only visible if the current road is visible. Otherwise, we consider it to intersect all the roads incident on the road intersection beyond the transition, and the car to be visible if any of the

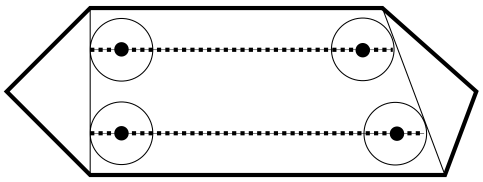


Figure B.5: *To speed visibility computations we locate transition points: the position parameter values where a car's geometry can conservatively intersect new roads. Here, four such points are marked with filled dots. If the car is outside the interval between the transition points then it is considered to intersect all the roads at the nearest intersection. Rather than compute the exact positions where the car's bound would cross the (thick) boundary of the road, we instead compute points where the bound crosses the (thin) stop lines. These are much easier to find because the car is traveling on a straight path at the transition points, rather than a hermite curve. The simplification will result in a small number of cars being considered visible a few frames before they actually should be, which does not incur a significant additional cost.*

roads are visible.

B.3 Car Models for Culling

To enable culling, the car simulation model must be extended to include the operations described in section 6.1, listed again here:

- Determine if an object may be visible, given its position.
- Bound an object's future locations, given its last known position and compute when that bound expires.
- Find the set of objects with visible bounds.
- Find the actual location of an object, given its last known position.
- Set the state of an object after some period out of view.
- Sample an initial position for an object.

Each component is now described in turn.

B.3.1 Determining Car Visibility

It is is necessary to determine whether or not a car is visible in order to keep the set of visible cars up to date. A car is visible if the road it is currently on is visible, or if its location parameter falls outside a visibility transition points and any road incident on the corresponding intersection is visible. Hence we can test a car's visibility, given its location parameter, in time $O(M)$, where M is the maximum number of roads at any intersection (typically less than five in our cities).

In algorithm 6.1 in section 6.1, the current visibility of cars that were in the visible set on the previous frame is checked after those cars are updated for the current frame, and cars are removed if necessary. New cars are added to the set in a later phase, when cars that were not visible on the previous frame are, if necessary, updated.

B.3.2 Bounding Future Locations

Each car not in the visible set must spatially bound its possible locations over some future time interval, given the car's current location, and specify when the interval ends.[1]In the virtual city, the bound consists of the set of roads that the car may traverse in the interval. Each car keeps pointers the roads in its bound, and each road keeps pointers to the cars that have bounds including the road. The pointers in roads allow us to efficiently determine which cars may be visible for a given frame — they are all the cars pointed to by visible roads.

Before generating the bound for a given car, we must first decide either which interval to bound or how many roads the bound should include (how big it should be). We choose the latter in this case, and determine the interval based on the required bound size.

A car's bound must be updated by the simulation when one of two things happen, each of which implies a desirable bound size:

- When the interval covered by the bound expires. This suggests large bounds covering long periods of time.
- When a road contained in the bound is seen by the viewer. This suggests small bounds covering very few roads.

It may be possible to analytically determine the optimal bound size at any given moment, which is the size that minimizes the expected cost of updating the bound on a given frame (or maximizes the expected number of frames between updates). However, the expectations depend on both how likely it is that a viewer will see a given bound, and how quickly the bound

[1]The bounds are conceptually identical to those used in the rigid-body simulation algorithm of chapter 3. In that case, bounds are used to guarantee that two objects cannot collide. Here, they are used to guarantee that an object is not visible.

grows with respect to the interval covered. Good models for these effects appear to be difficult to determine.

In the experiments described here we take a heuristic approach to setting the bound size, k. We fix a maximum bound size, k_{max}, and allow the actual size for a given car to vary between one and k_{max}. Each time a car's bound is updated because it was potentially visible, the size for that car is reset to one. Each time the bound is updated because it expired, the size is doubled (up to the maximum size). The result is that cars in the region surrounding the viewer tend to have smaller bounds than those a long way from the viewer. This is reasonable behavior, because for cars near the viewer updates due to visibility dominate, while the cost for cars a long way away is dominated by expirations. For the experiments described in section 6.2, we use $k_{max} = 32$.

Given the bound size, $c.k$ for a car c, a procedure is required that will return approximately $c.k$ roads that the car may next traverse, and the time at which the car may first leave that set of roads (which is the time the bound expires). All the roads that the car can reach before the expiration time must appear in the bound.

We assume that the car is on a particular path, $c.path$, and we know when the car first entered the path, and the current time. A preprocessing step computes the minimum time a car can take to traverse each path, and the minimum time a car can take to get to the visibility transition point at the end of the path.

Because the cars perform random walk, we use a weighted breadth first search algorithm to locate roads for the bound. The algorithm maintains a priority queue, Q, in which paths are stored, ordered by when the car will first cross the visibility transition point onto the path. Along with each path, the queue stores the time the car's origin enters the path, which is necessary for computing the time the car will get to its next visibility transition point.

The bound is initialized to contain the car's current path, $c.path$, and Q is seeded with all the legal paths out of the intersection at the end of $c.path$, keyed by the earliest time the car can cross the visibility transition point at the end of $c.path$. That time may be in the past, because the car may have already crossed the point. It is computed based on the current time, the time the car entered the path and the minimum time to get to the transition point. Also, if the car has not yet crossed the visibility transition point at the start of $c.path$, then all the roads leading into the current path must be included in the bound.

The procedure then repeatedly pops the minimum entry in Q, adds the road for the minimum path p to the bound (if it isn't already there), and adds any paths out of p to Q keyed on the earliest time the car can get to the transition point at the end of p. It stops when the number of roads in the bound reaches $c.k$. The expiration time for the bound is the key of the minimum entry in Q when the procedure terminates, because it is the earliest time that the car can get to a road not in the bound. The car is then inserted into the global queue of expiration times, keyed by its new expiration time.

B.3.3 Positioning Cars

At two points in the algorithm it is necessary to choose a location for previously non-visible cars: when the car's bound overlaps a visible road so the car may be visible, and when a car's bound expires (see the previous section). Each car's location is specified by the path it is currently on, and its parametric position on that path.

In determining a car's location, it is necessary to take into account all that might have happened to the car while it was out of view: the car should have stopped at all the intersections, it may have had to wait for other cars to pass, or it may have been slowed down by cars in front of it. However, we are only interested in the cumulative effect of all these things in determining where the car is now. Rather than try to model them directly, we build a probabilistic approximation of their effect, and sample from that approximation to choose the location.

The particular random variable we model is the traversal time for a given path. That is, the time taken from when the car starts out from the stop-line at the beginning of the path in question to the time it actually enters the intersection at the end, and starts out on a new path, including the time spent waiting to enter the intersection. A exponential distribution was assumed for each traversal time in the implementation described here. A more recent implementation uses mixtures of exponentials, estimated using an EM algorithm [8].

An exponential distribution is parameterized by its mean. Given a set of samples from the distribution, we estimate the true mean with the sample mean, which is the maximum likelihood estimate. To gather the samples, we run the complete, un-culled simulation in a preprocessing step and record a sample each time any car traverses a path. The distribution of the samples then captures the expected behavior of a car traversing each path, which is exactly the distribution we wish to use when culling. The underlying assumption is that if a viewer measures reasonable traversal times in the culled simulation they will not detect the presence of culling.

The exponential distribution is defined on $[0, \infty)$, so instead of using the samples for a given path directly we first subtract off the minimum traversal time seen among the samples for that path. The minimum time is also needed for bounding the a car's location, as discussed above.

In summary, we model the probability that a car took a certain time, t to traverse a path, p, with the density function:

$$f(t) = \frac{1}{\beta_p} e^{\frac{t - t_{min,p}}{\beta_p}}, t \geq t_{min,p}$$

where $t_{min,p}$ is the minimum possible time required to traverse the road (in the absence of other cars) and β_p is the mean of the distribution, estimated by:

$$\beta_p = \frac{1}{N} \sum_{i=1}^{N} (t_i - t_{min_p})$$

for N sample times t_i, computed in a pre-processing step.

The first stage in positioning a car is determining which road it is on. We do a random walk starting at the previous location and keep track of how much time has elapsed, until the elapsed time has brought us up to the current frame. The following pseudocode describes the algorithm, where $c.path$ is the car's current path, $c.t_{entry}$ is the time the car started on its current path, $c.t_{current}$ is the time corresponding to the car's current location, and t_{frame} is the frame time. sampleTraversal samples a traversal time from the model described above. At the end of the procedure, the car is on a new path and the time we think it entered that road has also been recorded.

Algorithm B.2 An algorithm to sample the location of a car

```
sampleRoad(car : $c$, real : $t_{frame}$)
    do
        $t_{exit} \leftarrow c.t_{entry}$ + sampleTraversal($c.path$)
    until $t_{exit} > c.t_{current}$
    while $t_{exit} < t_{frame}$
        $c.t_{entry} \leftarrow t_{exit}$
        $c.path \leftarrow$ nextPath($c.path$)
        $t_{exit} \leftarrow c.t_{entry}$ + sampleTraversal($c.path$)
    $c.t_{current} \leftarrow t_{frame}$
```

To determine the car's parametric position along the path, u, we use approximations. In a preprocessing stage for each path, we fit a piecewise cubic function, $\hat{u}_{path}(\Delta t)$, that returns

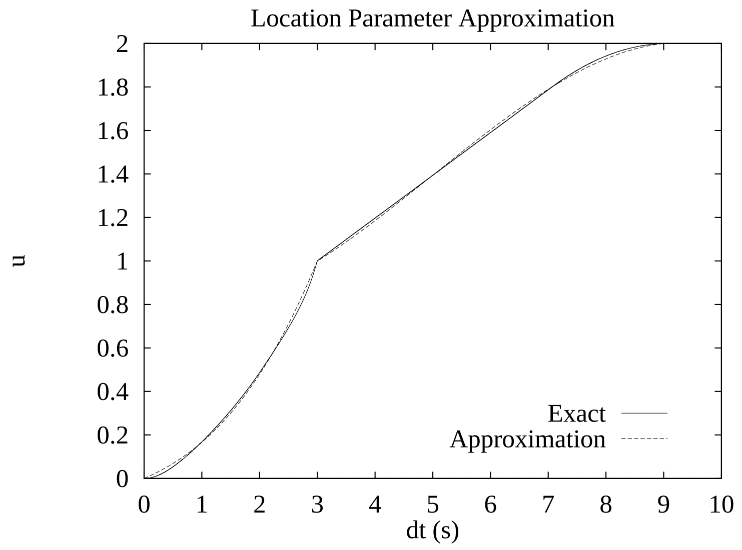


Figure B.6: *An example of the approximation function used for a car's location on a given path, compared to the exact function. The horizontal axis is the time spent on the path so far, Δt. This path takes just over nine seconds to traverse in the absence of other cars. The vertical axis is a car's location parameter, u, which ranges from zero at the start of the path to two at the end. Note the discontinuity at $u = 1$, where the parameterization switches from the parameter value for the hermite curve through the intersection to that for distance along the straight road. Note also the reduction in the gradient as the car slows down to stop at the end of the path. There is a close match between the exact and approximate representations, so the viewer will not see strange locations as cars re-enter the view.*

the car's approximate position on the path given how long it has been on the path, $\Delta t = t_{frame} - c.t_{entry}$, assuming that no other cars influenced its motion (figure B.6). To find the parameter value for a car, we simply compute Δt and evaluate the approximation. The approximation is only defined for $\Delta u \in [0, t_{min})$, where t_{min} is the time taken for a car to traverse the road in the absence of other cars. But the traversal time we sampled for this road in procedure sampleRoad may be greater than t_{min}, because it takes into account the effects of other cars, and hence Δt may be grater than t_{min}. If that is the case, we assume the car is queued at the end of the road, and tentatively set its parameter value to 2.0.

At this point the car's position may violate the simulation invariants implied by the car collision avoidance scheme:

- Only one car is in any intersection at any time.
- Cars are a minimum distance, d_{min}, apart.

If the viewer cannot see the car, it does not matter that the invariants are violated. So at this point in the car placement algorithm we do a visibility test for the car (section B.3.1). If the car is not visible, we generate a new bound for the car (section B.3.2) and continue.

If the car is visible, we must deal with any violated invariants. We assume that all the cars currently in the visible set are in their correct position for this frame, and cannot be moved. Hence we can only adjust the position of the car we are trying to place. We apply the following rules, in order, to reposition a car that violates invariants:

1. If the car is in an intersection ($u < 1.0$), then reject this position and re-run procedure sampleRoad.
2. Search along the path for empty spaces between or at the end of existing cars that are large enough to fit this car, and put this car in the closest one to its tentative u.
3. If there are no gaps for the car, reject this position and try again.

It is possible that many tentative positions may be rejected before a good position is found, or even that there is no free position among those that are likely for the car (where likely is defined by the distributions on traversal times for the roads the car may traverse). As a heuristic to reduce the number of rejections, we delay positioning cars until we have identified all the cars that must be positioned on this frame. We then sort the cars according to their time out of view, and try to position cars that have been out of view for shorter periods first.

The idea is that cars that have been out of view for longer have more variance in their possible positions, and hence are easier to find places for, even on cluttered streets. Even this heuristic fails sometimes, so if a car is rejected too many times we pretend that the time is later than it actually is, and try again. This introduces errors, because cars will appear to travel faster than they should, but these errors are not noticeable when the simulation is viewed. The time skew can be removed later if desired.

Adjusting the position of cars can affect its visibility, so we delay the final decision of whether or not a car is in the visible set until after placement is complete.

B.3.4 Setting Car State

Once the location of a visible car is known, it remains to set the rest of that car's state. Precisely, given the location of the car, how long it has been out of view, and the neighboring cars, set the car's orientation, speed and wheel orientations (the coordinates of the car and its steering angle come directly from the car's location). We assume that the viewer cannot predict the orientation of the car's wheels if the car been out of view. To do so would require a viewer predicting exactly how many wheel revolutions were required for the distance the car traveled while out of view, which is so difficult that a random orientation suffices for each wheel.

To set the orientation of the car, we use piecewise cubic approximation functions, $\hat{o}_{path}(u)$, defined for every path (figure B.7). These functions return the approximate orientation at the car's location, and are fitted as part of a preprocessing step. We can use approximations because, while the car's orientation is specified as the solution to a differential equation, the orientation at the start of each path (the initial condition) is derived only from the lane leading into the path, and the orientation evolves along the path in the same way each time.

The speed of a car in the city depends on both its location and the presence of neighboring cars. If there are no cars nearby, the speed can be approximated by a set of piecewise cubic functions $\hat{s}_{path}(u)$ defined for each path (figure B.8). Approximations are suitable because the car starts each path with zero speed (it was stopped at the stop sign), and so the differential equations for the speed always evolve in the same way provided there is no nearby car to slow this one down. If there is a car close in front of this car, we set the car's speed to match that of the one in front. Note that this gives correct queuing behavior, because if the car in front is stopped then this one will also be stopped behind it. However, matching speeds in this way does not always give the following car the correct speed. A better approach would fit

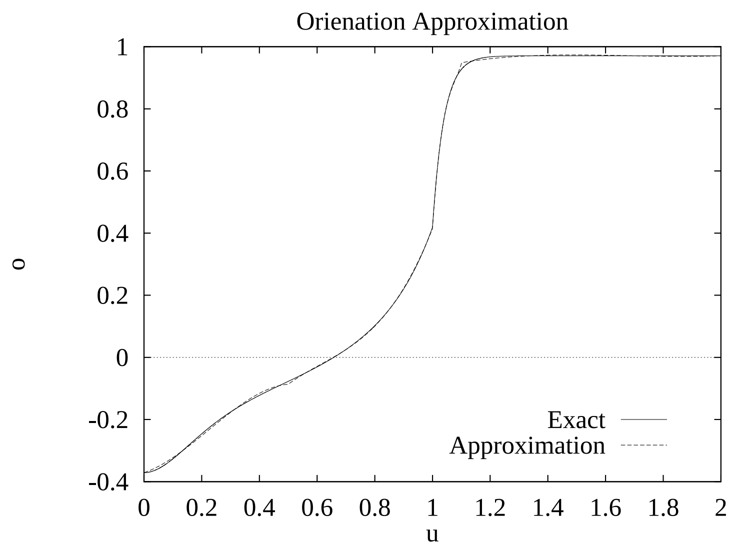


Figure B.7: *An example of the approximation function used for a car's orientation on a given road, compared to the exact function. The car's location parameter, u, increases along the horizontal axis. The orientation, o, measured in radians, is plotted on the vertical axis. The orientation varies as the car goes around the corner through the intersection, then becomes constant as the car traverses the straight road portion of the path. The approximation is close to the real function in most places, so the user will see reasonable orientations. Even where the approximation deviates, the dynamics will drive it quickly back to the correct value as the car continues along the path.*

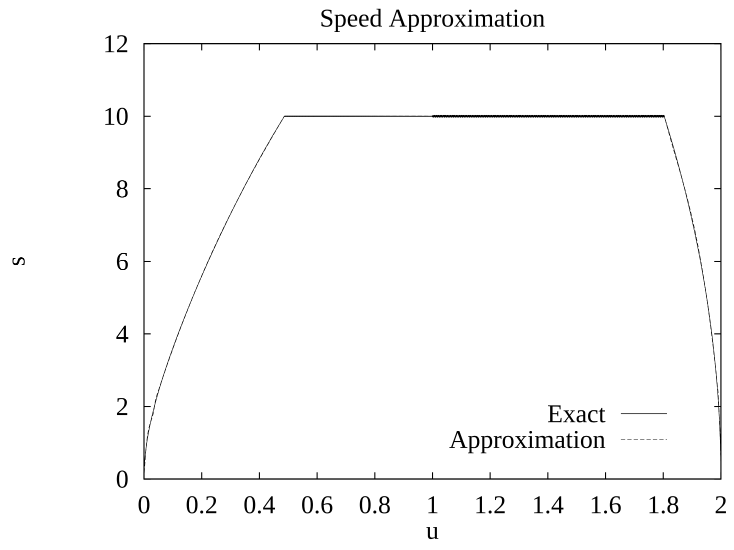


Figure B.8: *An example of the approximation function used for a car's speed on a given road in the absence of other cars, compared to the exact function. The horizontal axis corresponds to the car's location parameter value, u, while the vertical plots speed, s, measured in meters per second. The first discontinuity is due to the car ceasing to accelerate when it reaches its maximum speed. The right discontinuity is at the location where the car begins decelerating to stop at the end of the path. The approximation and exact function are essentially identical, which ensures that the car does not overrun the end of the path.*

the speed and following distance as a function of the speed of the car in front and the location on the road. This would improve the behavior as cars approach intersections.

B.3.5 Sampling Initial Positions

At the beginning of the simulation, each car must be given a position. Before the simulation begins the viewer has no information about the location of specific cars, so the correct approach is to sample car locations according to the probability that each car is on any given path at a random time. We estimate these probabilities in a preprocessing step, where we run the full simulation with all cars interacting, and record the total time spent by all the cars on each path. The probability that a car is on any given path at the start of the simulation is equal to the ratio of the time spent by all cars on that path to the total time spent by all cars on all paths.

To choose an initial path for each car, we take a sample from the distribution on paths. A parameter value for the location on the path is still required, as is the time the car entered the path (required for bounding the car's motion as described in section B.3.2). To set these values, we take a sample t_{max} from the distribution of traversal times for the path in question, and then assume the car has been on the path for a random time $t \in [0, t_{max}]$. Given that the simulation starts at time zero, $-t$ is the entry time for the path. We can then use the approximations described in section B.3.3 to select a location parameter value u.

At the start of the simulation no cars are visible (the viewer doesn't exist yet), so all the cars set their bounds and expiration times. On the first frame the visible roads will be known and then potentially visible cars can have their state set as usual.

B.3.6 Preprocessing Summary

Two major preprocessing steps must be run to build all the approximation functions and distributions required by the culling algorithm. One simulation is run with all the cars, and is used to gather samples for estimating the traversal time distributions and the probabilities used for the initial positioning of cars. A new sample is obtained every time a car in the simulation completes a traversal of a path, at which point the measured traversal time is a new sample for the traversal time estimation. The time spent on the path is also added to the time spent by all cars on the path. The distributions estimated in this phase depend on the number of cars in the simulation, as well as the city model. Hence, this step must be conducted every time the city or number of participants is changed.

A second preprocessing step looks at every path in turn to build the approximation functions $\hat{u}_{path}(\Delta t)$, $\hat{o}_{path}(u)$ and $\hat{s}_{path}(u)$. For each path, a car is positioned at the start of the path with zero speed and an orientation aligning it with the lane leading into the path. The equations of motion for the car are then solved with a small time step to obtain a large set of sample states. We then use a least squares approach to fit piecewise cubic curves to the necessary functions. The approximation functions depend only on the dynamics of an individual car and the city model, so need only be run once for each city.

Bibliography

[1] John M. Airey, John H. Rohlf, and Jr. Frederick P. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, pages 41–50, March 1990.

[2] Ken Arnold and James Gosling. *The Java programming language: Second edition*. Addison-Wesley, 1998.

[3] Joel Auslander, Alex Fukunaga, Hadi Partovi, Jon Christensen, Lloyd Hsu, Peter Reiss, Andrew Shuman, Joe Marks, and J. Thomas Ngo. Further experience with controller-based automatic motion synthesis for articulated figures. *ACM Transactions on Graphics*, 14(4):311–336, October 1995.

[4] David Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *Computer Graphics*, volume 23(3), pages 223–232. ACM SIGGRAPH, July 1989. Boston, Massachusetts.

[5] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *Computer Graphics, Proceedings of SIGGRAPH 94*, pages 23–34. ACM SIGGRAPH, 1994.

[6] Ronan Barzel and Alan H. Barr. A modeling system based in dynamic constraints. In *Computer Graphics: Proceedings of SIGGRAPH 88*, pages 179–188, 1988. Atlanta, Georgia, August 1-5.

[7] Ronan Barzel, John F. Hughes, and Daniel N. Wood. Plausible motion simulation for computer graphics animation. In *Computer Animation and Simulation '96*, pages 184–197, 1996. Proceedings of the Eurographics Workshop in Poitiers, France, August 31-September 1, 1996.

[8] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[9] Bruce M. Blumberg and Tinsley A. Galyean. Multi-level direction of autonomous creatures for real-time virtual environments. In *Computer Graphics: Proceedings of SIGGRAPH 95*, August 1995. Los Angeles, CA.

[10] David Brogan and Jessica Hodgins. Group behaviors for systems with significant dynamics. In *Proceedings of the 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 528–534, 1995.

[11] Lynne Shapiro Brotman and Arun N. Netravali. Motion interpolation by optimal control. *Computer Graphics (SIGGRAPH 88 Conference Proceedings)*, 22(4):309–315, August 1988.

[12] John Calsamiglia, Scott W. Kennedy, Anindya Chatterjee, Andy L. Ruina, and James T. Jenkins. Anomalous frictional behavior in collisions of thin disks. *Journal of Applied Mechanics*, 66(1):146–152, 1999.

[13] 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.

[14] Marc Cavazza, Rae Earnshaw, Nadia Magnenat-Thalmann, and Daniel Thalmann. Survey: Motion control of virtual humans. *IEEE Computer Graphics & Applications*, 18(5):24–31, 1998.

[15] Anindya Chatterjee and Andy Ruina. A new algebraic rigid body collision law based on impulse space considerations. *Journal of Applied Mechanics*, 65(4):939–951, December 1998.

[16] Stephen Chenney. Culling dynamical systems in virtual environments. Master's thesis, University of California at Berkeley, 1997. EECS, Computer Science Division.

[17] 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.

[18] Jon Christensen, Joe Marks, and J. Thomas Ngo. Automatic motion synthesis for 3D mass-spring models. *The Visual Computer*, 13(3):20–28, Jan 1997.

[19] Jonathan Cohen, Ming C. Lin, Dinesh Manocha, and Madhav Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large scale environments. In *Proceedings of the 1995 ACM Symposium on Interactive 3D Graphics*, pages 189–196, 1995.

[20] Michael F. Cohen. Interactive spacetime control for animation. In *Computer Graphics: Proceedings of SIGGRAPH 92*, volume 26(2), pages 293–302, 1992.

[21] Satyan Coorg, Neel Master, and Seth Teller. Acquisition of a large pose-mosaic dataset. In *Proceedings of CVPR 98*, pages 827–878, 1998.

[22] Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. In *Proceedings 1997 Symposium on Interactive 3D Graphics*, pages 83–90, April 1997. Providence, RI, April 27-30.

[23] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, 1997.

[24] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approcah. In *Computer Graphics: Proceedings of SIGGRAPH 96*, pages 11–20. ACM SIGGRAPH, 1996. New Orleans, LA.

[25] Frédo Durand, George Drettakis, and Claude Puech. The 3d visibility complex: A new approach to the problems of accurate visibility. In *Proceedings of the Eurographics Rendering Workshop 1996*, pages 245–256, June 1996.

[26] Jeff Erickson, Leonidas J Guibas, Jorge Stolfi, and Li Zhang. Separation-sensitive collision detection for convex objects. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 327–336, 1999.

[27] Nick Foster and Demitri Metaxas. Realistic animation of liquids. In *Graphics Interface '96*, pages 204–212, 1996.

[28] Alain Fournier and William T. Reeves. A simple model of ocean waves. In *Computer Graphics: Proceedings of SIGGRAPH 86*, volume 20(4), pages 75–84, 1986. Dallas, TX, August 18-22.

[29] Thomas A. Funkhouser and Carlo H. Sequin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Computer Graphics*, pages 247–254. ACM SIGGRAPH, 1993.

[30] Thomas A. Funkhouser, Carlo H. Sequin, and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. In *Proceedings 1992 Symposium on Interactive 3D Graphics*, pages 11–20, 29 March - 1 April 1992. Cambridge, Massachusetts.

[31] Charles J. Geyer and Elizabeth A. Thompson. Annealing Markov chain Monte-Carlo with applications to ancestral inference. Technical Report 589, School of Statistics, University of Minnesota, 1994.

[32] Walter R Gilks, Sylvia Richardson, and David J Spiegelhalter. *Markov chain Monte Carlo in Practice*. Chapman & Hall, 1996.

[33] Michael Gleicher. Motion editing with spacetime constraints. In *Proceedings 1997 Symposium on Interactive 3D Graphics*, pages 139–148, April 1997. Providence, RI, April 27-30.

[34] Michael Gleicher. Retargetting motion to new characters. In *Computer Graphics, Proceedings of SIGGRAPH 98*, pages 33–42. ACM SIGGRAPH, 1998.

[35] Suresh Goyal, Elliot N Pinson, and Frank W Sinden. Simulation of dynamics of interacting rigid bodies including friction II: Software system design and implementation. *Engineering with Computers*, 10:175–195, 1994.

[36] Radek Grzeszczuk and Demetri Terzopoulos. Automated learning of muscle-actuated locomotion through control abstraction. In *Computer Graphics, Proceedings of SIGGRAPH 95*, pages 63–70, August 1995. Los Angeles, California.

[37] Radek Grzeszczuk, Demetri Terzopoulos, and Geoffrey Hinton. Neuroanimator: Fast neural network emulation and control of physics-based models. In *Computer Graphics, Proceedings of SIGGRAPH 98*, pages 9–20. ACM SIGGRAPH, 1998.

[38] John Guckenheimer and Philip Holmes. *Nonlinear Oscillations, Dynamical Systems, and Bifurcations of Vector Fields*. Springer-Verlag, New York, 1983.

[39] James K. Hahn. Realistic animation of rigid bodies. In *Computer Graphics*, volume 22(4), pages 299–308. ACM SIGGRAPH, August 1988. Atlanta, Georgia.

[40] Vincent Hayward, Stéphane Aubry, André Foisy, and Yasmine Ghallab. Efficient collision prediction among many moving objects. *The International Journal of Robotics Research*, 14(2):129–143, April 1995.

[41] Jessica Hodgins and Nancy Pollard. Adapting simulated behaviors for new creatures. In *Computer Graphics: Proceedings of SIGGRAPH 97*, pages 153–162, August 1997. Los Angeles, CA.

[42] Jessica K. Hodgins, James F. O'Brien, and Jack Tumblin. Perception of human motion with different geometric models. *IEEE Transactions on Visualization and Computer Graphics*, 4(4):307–316, 1998.

[43] 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.

[44] Chieh Su Hsu. *Cell-to-cell mapping: a method of global analysis for nonlinear systems*. Springer-Verlag, New York, 1987.

[45] Philip M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, Sept 1995.

[46] Mark Jerrum and Alistair Sinclair. Approximating the permanent. *SIAM Journal of Computing*, 18:1149–1178, 1989.

[47] Mark Jerrum and Alistair Sinclair. The Markov chain Monte Carlo method: an approach to approximate counting and integration. In D.S.Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*. PWS Publishing, Boston, 1996.

[48] Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. In *Computer Graphics: Proceedings of SIGGRAPH 90*, volume 24(4), pages 49–55, 1990. Dallas, TX, August 6-10.

[49] 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.

[50] Dong-Jin Kim, Leonidas J. Guibas, and Sung-Yon Shin. Fast collision detection among multiple moving spheres. *IEEE Transactions on Visualization and Computer Graphics*, 4(3):230–242, 1998.

[51] Ming C Lin and Stefan Gottschalk. Collision detection between geometric models: A survey. In *Proceedings of IMA Conference on Mathematics of Surfaces*, 1998. To appear.

[52] Zicheng Liu, Steven J. Gortler, and Michael F. Cohen. Hierarchical spacetime control. In *Computer Graphics: Proceedings of SIGGRAPH 94*, pages 35–42, 1994.

[53] David Luebke and Chris Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 105–106, April 1995.

[54] Maurits Malfait and Dirk Roose. Convergence of concurrent Markov chain Monte-Ccarlo algorithms. *Concurrency, Practice and experience*, 8(3):167–189, April 1996.

[55] J. Marks, B. Andalman, P.A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryall, J. Seims, and S. Shieber. Design galleries: A general approach to setting parameters for computer graphics and animation. In *Computer Graphics: Proceedings of SIGGRAPH 97*, pages 389–400, August 1997. Los Angeles, CA.

[56] Michael D. McCool and Peter K. Harwood. Probability trees. In *Graphics Interface '97*, pages 37–46, 1997.

[57] Michael McKenna, Steve Pieper, and David Zeltzer. Control of a virtual actor: The roach. In *ACM SIGGRAPH 1990 Symposium on Interactive 3D Graphics*, volume 24(2), pages 165–174, 1990.

[58] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. In *Computer Graphics: Proceedings of SIGGRAPH 95*, pages 39–46, August 1995. Los Angeles, CA.

[59] Gavin S. P. Miller. The motion dynamics of snakes and worms. In *Computer Graphics (Proceedings of SIGGRAPH 88)*, volume 22(4), pages 169–178, August 1988.

[60] Brian Mirtich. *Impulse-based Dynamics for Rigid-Body Simulation*. PhD thesis, University of California, Berkeley, 1996.

[61] Brian Mirtich. V-clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics*, 17(3):177–208, 1998.

[62] Brian Mirtich and John Canny. Impulse-based simulation of rigid bodies. In *Proceedings of the 1995 ACM Symposium on Interactive 3D Graphics*, pages 181–188, 1995.

[63] Brian Mirtich, Yan Zhuang, Ken Goldberg, John Craig, Rob Zanutta, Brian Carlisle, and John Canny. Estimating pose statistics for robotic part feeders. In *Proceedings 1996 IEEE International Conference on Robotics and Automation*, volume 2, pages 1140–1146, 1996.

[64] Matthew Moore and Jane Wilhelms. Collision detection and response for computer animation. In *Computer Graphics*, volume 22(4), pages 289–298. ACM SIGGRAPH, August 1988. Atlanta, Georgia.

[65] Ketan Mulmuley. *Computational Geometry : an Introduction Through Randomized Algorithms*. Prentice-Hall, 1994.

[66] Radford M. Neal. Sampling from multimodal distributions using tempered transitions. *Statistics and Computing*, 6:353–366, 1996.

[67] Radford M. Neal. Annealed inportance sampling. Technical Report 9805, Department of Statistics, University of Toronto, 1998.

[68] Michael Neff and Eugene Fiume. A visual model for blast waves and fracture. In *Proceedings of Graphics Interface '99*, pages 193–202, June 1999.

[69] J. Thomas Ngo and Joe Marks. Spacetime constraints revisited. In *Computer Graphics: Proceedings of SIGGRAPH 93*, pages 343–350, 1993.

[70] James F. O'Brien and Jessica K. Hodgins. Dynamic simulation of splashing fluids. In *Proceedings of Computer Animation '95*, pages 198—205, April 1995.

[71] James F. O'Brien and Jessica K. Hodgins. Graphical modeling and animation of brittle fracture. In *Computer Graphics: Proceedings of SIGGRAPH 99*, pages 137–146, August 1999.

[72] M. Paterson and F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete & Computational Geometry*, 5(5):485–503, 1990.

[73] Jovan Popović, Steven Seitz, Michael Erdmann, Zoran Popović, and Andrew Witkin. Interactive Manipulation of Rigid Body Simulations. In *SIGGRAPH 2000 Conference Proceedings*. ACM SIGGRAPH, July 2000.

[74] Zoran Popović and Andrew Witkin. Physically based motion transformation. In *Computer Graphics: Proceedings of SIGGRAPH 99*, pages 11–20. ACM SIGGRAPH, 1999. Los Angeles, California, August 8-13, 1999.

[75] 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.

[76] 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*, pages 72–81, July 1995. University of Iowa, Iowa City, IA.

[77] Karl Sims. Evolving virtual creatures. In *SIGGRAPH 94 Conf. Proc.*, pages 15–22, Orlando, Florida, July 1994.

[78] Alistair Sinclair, 1999. Personal communication.

[79] Jos Stam. Stable fluids. In *Computer Graphics: Proceedings of SIGGRAPH 99*, pages 121–128, August 1999.

[80] Oded Sudarsky and Craig Gotsman. Output-sensitive visibility algorithms for dynamic scenes with applications to virtual reality. In *EUROGRAPHICS '96*, volume 15(3), 1996.

[81] Oded Sudarsky and Craig Gotsman. Dynamic scene occlusion culling. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):13–29, 1999.

[82] Richard Szeliski and Demetri Terzopoulos. From splines to fractals. In *Computer Graphics*, volume 23(3), pages 51–60. ACM SIGGRAPH, July 1989. Boston, Massachusetts.

[83] Diane Tang, J. Thomas Ngo, and Joe Marks. N-body spacetime constraints. *The Journal of Visualization and Computer Animation*, 6:143–154, 1995.

[84] Seth J. Teller and Carlo H. Sequin. Visibility preprocessing for interactive walkthroughs. In *Computer Graphics*, volume 24(4), pages 61–70. ACM SIGGRAPH, July 1991. Las Vegas, Nevada.

[85] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. Elastically deformable models. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, pages 205–214, July 1987.

[86] Xiaoyuan Tu and Demetri Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. In *Computer Graphics: Proceedings of SIGGRAPH 94*, pages 43–50, 1994.

[87] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Computer Graphics: Proceedings of SIGGRAPH 97*, pages 65–76, August 1997. Los Angeles, CA.

[88] B C Vemuri, Y Cao, and L Chen. Fast collision detection algorithms with applications to particle flow. *Computer Graphics Forum*, 17(2):121–134, 1998.

[89] Baba C Vemuri, Chhandomay Mandal, and Shang-Hong Lai. A fast Gibbs sampler for synthesizing constrained fractals. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):337–351, 1997.

[90] The virtual reality modeling language, 1997. International Standard ISO/IEC 14772-1:1997. Available from http://www.web3d.org/.

[91] Robert Webb and Mike Gigante. Dynamic bounding volume hierarchies to improve efficiency of rigid body simulations. In *Visual Computing (Proceedings of Computer Graphics International)*, pages 825–842, 1992.

[92] Henrik Weimer and Joe Warren. Subdivision schemes for fluid flow. In *Computer Graphics: Proceedings of SIGGRAPH 99*, pages 111–120, August 1999.

[93] J R Williams and R O'Connor. A linear complexity intersection algorithm for discrete element simulaton of arbitrary geometries. *Engineering Computations*, 12:185–201, 1995.

[94] Andrew Witkin and Michael Kass. Spacetime constraints. In *Computer Graphics: Proceedings of SIGGRAPH 88*, pages 159–168, 1988. Atlanta, Georgia, August 1-5.

[95] Qinxin Yu and Demetri Terzopoulos. Synthetic motion capture: Implementing an interactive virtual marine environment. *The Visual Computer*, pages 377–394, 1999.

[96] Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth E. Hoff III. Visibility culling using hierarchical occlusion maps. In *Computer Graphics: Proceedings of SIGGRAPH 97*, pages 77–88, August 1997. Los Angeles, CA.

[97] Yan Zhuang and John Canny. Real-time global deformations. In *Proceedings of the Fourth International Workshop on Algorithmic Foundations of Robotics (WAFR)*, 2000. Hanover, NH. March 16-18.