# Interactive Simulation of Solid Rigid Bodies

**David Baraff**
*Carnegie Mellon University*

**This system simulates the dynamics of rigid bodies with contact, collision, and friction. The current implementation achieves interactive simulation rates for moderately complex mechanical systems on a low-end SGI workstation.**

To be interactive, a dynamic simulation system must be fast enough to let users directly manipulate simulated objects as they move about and flexible enough to allow on-the-fly modifications to the simulation environment. Interactive simulation systems that can enforce bilateral constraints between objects are becoming common. The system described in this article extends interactive simulation to encompass contact constraints, thus preventing solid objects from moving through one another.

Figure 1 on the next page (inspired by the work of Sacks and Joskowicz[1]) shows the motion of a simulated block feeder. In an interactive environment, users should be able to design, construct, and easily interact with these sorts of simulated mechanisms and objects.

Contact constraints present a much greater challenge than bilateral constraints (commonly referred to as "equality constraints"). Bilateral constraints, such as those in Figure 1, typically arise in representing idealized geometric connections (for example, universal joints and point-to-point or point-to-surface constraints). Bilateral constraints are relatively static in that they must be both explicitly created or deleted by the user. In contrast, contact constraints come and go as objects come into contact with one another and then later break apart. As a result, the system needs collision detection algorithms to know when to create a new contact constraint. Once established, a contact constraint is treated differently than a bilatoral constraint. The system must relax contact constraints, allowing objects to separate when appropriate. No such need exists for bilateral constraints. Finally, depending on the application, contact constraints may also require the simulation of frictional effects due to contact.

This article describes the implementation of an interactive system for simulating rigid bodies with contact and friction. The system can simulate moderately complex mechanical systems at interactive rates (20-30 Hz on low-end Silicon Graphics workstations). New objects and user-specified constraints can be added into the simulation environment on the fly.

The system uses analytical methods to compute contact forces, as opposed to the penalty methods common in other interactive systems. Currently, the system's weakest feature is that it can fail to detect high-speed collisions. Objects that move at high speeds (relative to the step size of the simulation) are subject to a form of aliasing and may "tunnel" through other objects without causing a collision.

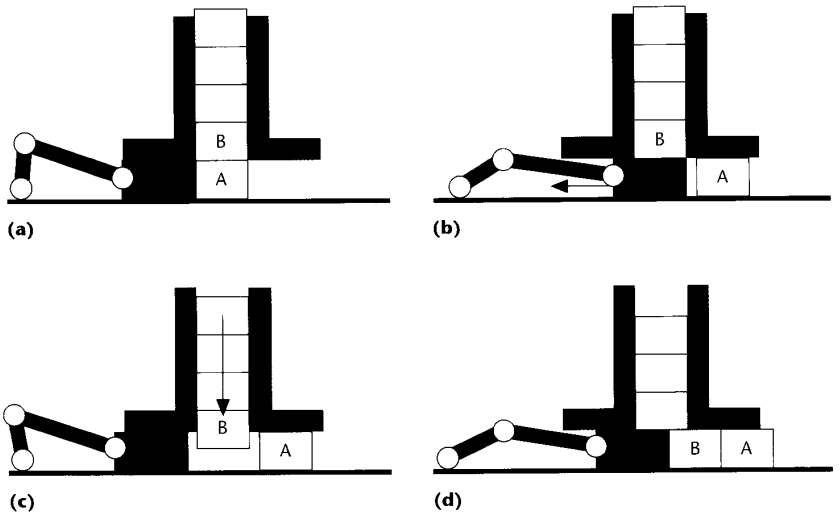
Other simplifications of the system involve approximating collision times and locations by interpolation methods, and periodic error-correction adjustments of geometric tolerances. It is difficult (and not terribly enlightening) to try to quantify the error incurred by a given approximation or trade-off. Some of the design choices will likely curtail the system's use for highly predictive applications (such as simulating the outcome of a roulette wheel). However, they do not seriously affect simulating the basic dynamics of a mechanism like the feeder in Figure 1. In general, the system performs with sufficient accuracy and realism to be considered a viable interactive simulation environment.

## Background

Interactive simulation systems have many applications. Realistic computer animation is one possibility. So are training and education, where systems that support realistic object motion within the geometric constraints of a layout allow users to experiment with and practice strategies for assembling and disassembling equipment and mechanisms.

Applications in robot path-planning and learning seem likely. The ability to quickly and easily simulate the contact and collision dynamics of robots and their surrounding environments could prove enormously useful. Interactive simulation systems could serve as a back end for robot path-planning algorithms, providing a vir-

0272-17-16/95/$4.00 © 1995 IEEE



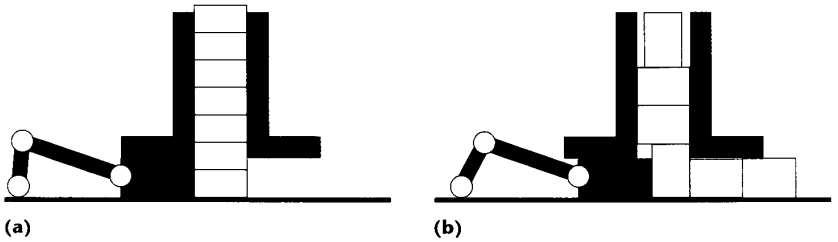

1 A simulated block feeder. By simply grabbing the four-bar linkage and pulling on it, users should be able to control the action of the mechanism and cause the blocks to feed smoothly down and to the right. The white circles indicate bilateral constraints that maintain constant connectivity among the parts of the four-bar linkage.

tual world in which to test a path-planning algorithm.

Sacks and Joskowicz[1] recently described a system for qualitative kinematic analysis. Their system uses an off-line algorithm to infer the allowable motions of objects from an initial configuration. For example, starting from the initial configuration in Figure 1a and given a pre-specified description of the driving force, the system can infer the feeder mechanism's action over time. The algorithm requires some simplifying assumptions on the allowable types of linkages and degrees of freedom between interacting parts in a mechanism.

Sacks and Joskowicz have used their algorithm to successfully analyze some impressively complicated mechanisms. However, to reduce the computational burden, the algorithm examines only behaviors that are realizable from the system's starting configuration, taking into account the (known) driving force. Because of this, the analysis process (as currently described) seems inherently noninteractive.

Sacks and Joskowicz's work results in an extremely "intelligent" but noninteractive simulation system. Given the starting configuration of Figure 1, their program understands how the parts in the mechanism move, but only so long as nothing unexpected happens. Interactive simulation requires a "dumber" mode of simulation, in which the system has no understanding of what it is simulating—but consequently, no expectations. Thus, if the simulation environment changes, as it does in Figure 2, there should be no need to go back and reanalyze the modified configuration. Instead, the simulation should simply reflect what would happen: In both instances in Figure 2, the blocks should cause the mechanism to jam and stop moving.




2 Changes that cause jamming in the block feeder: (a) a badly designed block feeder and (b) misaligned blocks.

## System overview

At a very high level, we can view the simulation as the process of numerically solving the differential equation

$$\frac{d}{dt}\mathbf{Y}(t) = f(\mathbf{Y}(t), t) \tag{1}$$

which describes the evolution of the system over time. The vector $\mathbf{Y}(t)$ describes the *state* of the system at time $t$. For a system with a single planar rigid body, Equation 1 is written as the coupled system

$$\frac{d}{dt}\mathbf{Y}(t) = \frac{d}{dt}\begin{pmatrix} \mathbf{x}(t) \\ \theta(t) \\ m\mathbf{v}(t) \\ I\omega(t) \end{pmatrix} = \begin{pmatrix} \mathbf{v}(t) \\ \omega(t) \\ \mathbf{F}(t) \\ \tau(t) \end{pmatrix} \tag{2}$$

where $\mathbf{x}, \mathbf{v} \in \mathbb{R}^2$ denote the center of mass and linear velocity of the body in world space, and $\theta(t)$ and $\omega(t)$ are scalars describing the orientation and angular velocity of the body. The total force acting on the body is denoted $\mathbf{F} \in \mathbb{R}^2$, and the scalar $\tau$ gives the total torque acting on the body. The mass and moment of inertia are $m$ and $I$, respectively. If the system consists of $k$ bodies, there will be $k$ copies of Equation 2. In this case, $\mathbf{Y}$ refers to the collection of the $k$ sets of $\mathbf{x}$, $\theta$, $\mathbf{v}$, and $\omega$ variables.

Equation 2 represents the equations of motion according to Newtonian dynamics. As such, the system experiences inertia: Objects tend to keep moving until brought to rest by forces in the simulation. In contrast, it is often useful to have a simulation world where objects do not move unless explicitly acted on by forces. Such a world is called a *first-order*

dynamic world and evolves according to the equation

$$\frac{d}{dt}\mathbf{Y}(t) = \frac{d}{dt}\begin{pmatrix}\mathbf{x}(t)\\ \theta(t)\end{pmatrix} = \begin{pmatrix}\mathbf{F}(t)/m\\ \tau(t)/I\end{pmatrix}. \quad \textbf{(3)}$$

In a first-order world, an object subject to a constant force travels at a constant rate. Thus, there is velocity, but no acceleration. First-order worlds are very useful for evaluating system kinematics.

A system evolving according to Equation 2 is called a *second-order* world. The system described in this article simulates both first- and second-order worlds. Most design issues in the simulator are independent of the type of simulation performed. Regardless of the specifics of the differential equation, the system's major goal is always to advance from a given state $\mathbf{Y}(t_0)$ to a new state $\mathbf{Y}(t_0 + \Delta t)$, where $\Delta t$ is the *step size* of the simulation.

### Computing dY/dt

The simulation system uses numerical integration to advance its state from $\mathbf{Y}(t_0)$ to a new state $\mathbf{Y}(t_0 + \Delta t)$. Numerical integration techniques require evaluating the right-hand side of Equation 3 or Equation 2 for particular values of $t$. This, in turn, requires computing the force $\mathbf{F}$ and torque $\tau$ acting on each object at the specified instant of time. The total force (that is, both $\mathbf{F}$ and $\tau$) acting on an object consists of external and internal forces. External forces are those known at any instant of time; they include gravity, velocity-based damping (for second-order systems), or user-specified interactions, such as a mouse-spring pulling on an object. Internal forces are the unknown forces that arise to satisfy the bilateral and contact constraints in the system, including friction.

The major work of computing $d\mathbf{Y}/dt$ lies in determining the internal forces. Once this is done, the numerical integration procedure proposes a new state for the system, based on the derivative computed for $\mathbf{Y}$. The system then attempts to move to the newly proposed state.

### Detecting collisions and contact

The differential equation $(d/dt)\mathbf{Y}(t) = f(\mathbf{Y}(t), t)$ contains no information about the geometry of objects in the simulation. Therefore, we cannot determine when collisions occur solely on the basis of the differential equation. A separate collision/contact detection component must check each proposed state $\mathbf{Y}(t_0)$ for possible intersection between objects. Given an initial state $\mathbf{Y}(t_0)$ in which no two objects interpenetrate, the newly proposed state $\mathbf{Y}(t_0 + \Delta t)$ must be checked for object interpenetration. If no interpenetration is detected in the proposed state, the system adopts it as the actual state for time $t_0 + \Delta t$. Otherwise, the system must determine at what time a collision first occurred between time $t_0$ and $t_0 + \Delta t$. It then advances the state to that point. The collision/contact detection module facilitates this process by providing a measure of the interpenetration depth between objects at varying times between $t_0$ and $t_0 + \Delta t$. To keep the display rate as constant as possible, intermediate states between $t_0$ and $t_0 + \Delta t$ are usually not displayed.

Once the system has determined a new state, it must find contact points between objects. At any point during the simulation, objects may come into contact or break contact. Whenever objects contact at a point, the system must maintain a contact constraint at that point to prevent the objects from interpenetrating in that vicinity. Multiple points of contact between a pair of objects yield multiple contact constraints.

### Constraint force computation

Once all the points of contact are determined, the system must compute the constraint forces that prevent interpenetration while also enforcing bilateral constraints. Contact constraints are challenging because they require simultaneously determining both the constraint forces and the possible removal of some of the contact constraints (allowing contact to be broken). The simultaneous computation of constraint forces and determination of when to turn off contact constraints leads to problems that are combinatorial in nature. Computing friction forces adds similar (but worse) complications.

A second-order system that includes rigid bodies may require discontinuities in an object's velocity. These discontinuities usually, but not always, result from a collision between two objects. Discontinuities are handled by computing *impulsive* constraint forces—that is, forces that act instantaneously with the units of momentum. In a first-order system, impulses do not arise.

### Error correction

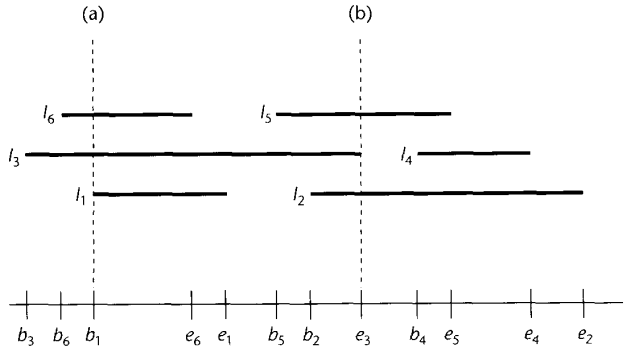
Numerical error inevitably builds up during a simulation. It is important to be able to correct this error. The error-correction facility in the system periodically readjusts the object configurations to reduce the degree of error in the various constraints being maintained. Error correction (when performed) occurs immediately after the system adopts a newly proposed state.

## Collision and contact determination

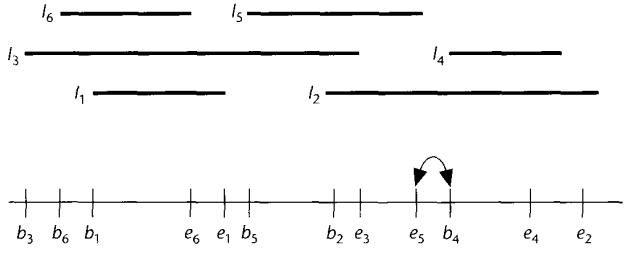
The current system supports objects defined as the union of some number of polygons, concave or convex. Concave polygons are internally broken down into a union of convex polygons. Thus, the system treats all objects as a union of convex polygons.

The previous off-line simulation system by Baraff[2] introduced fast collision-detection algorithms that exploited geometric coherence. The current system uses these algorithms for checking interpenetration and finding contact points between a pair of convex polygons. Because the algorithms exploit coherence, the time required to check for interpenetration or find contact points is, for all practical purposes, $O(1)$ for a given pair of polygons. Lin and Canny[3] offer a more sophisticated and detailed treatment of the subject.

However, note that these coherence-based methods cost $O(1)$ per *pair* of polygons checked. For a simulation with $n$ objects, $O(n^2)$ comparisons costing $O(1)$ each is obviously unacceptable. We need a way to determine which pairs of objects really require consideration by the collision-detection algorithm. To simplify the problem, we can imagine that each object in the simu-




**3 The sweep/sort algorithm. (a) When $b_1$ is encountered, the active list contains intervals 3 and 6; interval 1 is reported to overlap with these two intervals. Interval 1 is added to the active list and the algorithm continues. (b) When $e_3$ is encountered, the active list contains intervals 2, 3, and 5. Interval 3 is removed from the active list.**




**4 A coherence-based method of detecting overlaps. The order produced in Figure 3 is nearly correct for this arrangement of intervals. Only $b_4$ and $e_5$ need to be exchanged. When the exchange occurs, the change in overlap status between intervals 4 and 5 is detected.**

lation is enclosed in a bounding box whose sides are parallel to the coordinate axes. Given objects $A$ and $B$, if their bounding boxes do not overlap, there is no need to subject the objects to any further detection. Clearly, we can implement this technique hierarchically. An object can contain sub-bounding boxes of some number of its polygons. Then if two objects are close together, only some subset of their polygons will require pairwise comparison. The problem at any level of the hierarchy, of course, is to quickly determine the pairwise overlap of the bounding boxes themselves.

We can employ methods from computational geometry. A basic result is that all pairwise overlaps among $n$ bounding boxes in two dimensions can be found in time $O(n \log n + k)$, where $k$ is the number of intersections. In the mechanical simulation environment described here, a coherence-based approach can achieve even better results and is considerably easier to implement than an $O(n \log n + k)$ algorithm.

### The one-dimensional case

Consider the problem of detecting an overlap between one-dimensional bounding boxes aligned with the coordinate system. A one-dimensional bounding box can be described simply as an interval $[b, e]$, where $b$ and $e$ are real numbers. Let's consider a list of $n$ such intervals, with the $i$th interval being $[b_i, e_i]$. The problem is then to determine all pairs $i$ and $j$ such that the intervals $[b_i, e_i]$ and $[b_j, e_j]$ intersect.

The problem can be solved initially by a *sort-and-sweep* algorithm. A sorted list of all $b_i$ and $e_i$ values is created, from lowest to highest. The list is then swept, and a list of *active* intervals (initially empty) is maintained. Whenever some value $b_i$ is encountered, all intervals on the active list are output as overlapping with interval $i$, and interval $i$ is then added to the list (Figure 3a). Whenever some value $e_i$ is encountered, interval $i$ is removed from the active list (Figure 3b). The cost of this process is $O(n \log n)$ to create the sorted list, $O(n)$ to sweep through the list, and $O(k)$ to output all the overlaps. The algorithm's total cost of $O(n \log n + k)$ is optimal for an initial solution to the problem.

Subsequent comparisons can be improved. First, there is no need to use an $O(n \log n)$ algorithm to form the sorted list of $b_i$ and $e_i$ values. It is considerably more efficient to start with the order found for $b_i$ and $e_i$ values from the previous time step. If coherence is high, this ordering will be nearly correct for the current time step. An *insertion sort* algorithm permutes the nearly sorted list into a sorted list by exchanging pairs of adjacent items, and it runs in time $O(n + c)$ where $c$ is the number of pairwise exchanges between adjacent elements needed to reorder the list. For example, the only difference between Figures 4 and 3 is that interval 4 has moved to the right. Starting from the ordered list of $b_i$ and $e_i$ values of Figure 3, only a single exchange is necessary to sort the list for Figure 4.

An insertion sort is not recommended in general because its worst-case running time is $O(n^2)$. However, it is a good algorithm for sorting a nearly sorted list, which occurs in the highly coherent environment of the system described here. To complete the algorithm, note that if two intervals $i$ and $j$ overlap at the previous time step, but not the current time step, one or more exchanges involving either a $b_i$ or $e_i$ value and a $b_j$ or $e_j$ value must occur. The converse is true as well when intervals $i$ and $j$ change from not overlapping at the previous time step to overlapping at the current time step.

Thus, if we maintain a table of overlapping intervals at each time step, the table can be updated at each time step with a total cost of $O(n + c)$. Assuming coherence, the number of exchanges $c$ necessary will be close to the actual number $k$ of changes in overlap status, and the extra $O(c - k)$ work will be negligible. Thus, for the one-dimensional bounding-box problem, the coherence view yields a simple, efficient algorithm that approaches optimality as coherence increases.

### The two-dimensional case

Efficient computational geometry algorithms for the bounding-box intersection problem in two or more dimensions are much more complicated than the sort-

and-sweep method for one dimension. However, these algorithms all include a step that is essentially a sort along a coordinate axis, as in the one-dimensional case. In the two-dimensional case, each bounding box is described as two independent intervals $[b_i^{(x)}, e_i^{(x)}]$ and $[b_i^{(y)}, e_i^{(y)}]$, which represent the intervals spanned on the two coordinate axes by the $i$th bounding box. Thus, one way to improve the efficiency of a computational geometry algorithm for coherent situations would be to sort one list containing the $b_i^{(x)}$ and $e_i^{(x)}$ values and a second list containing the $b_i^{(y)}$ and $e_i^{(y)}$ values. Again, such a step will involve $O(n + c)$ work, where $c$ is now the total number of exchanges involved in sorting both lists.

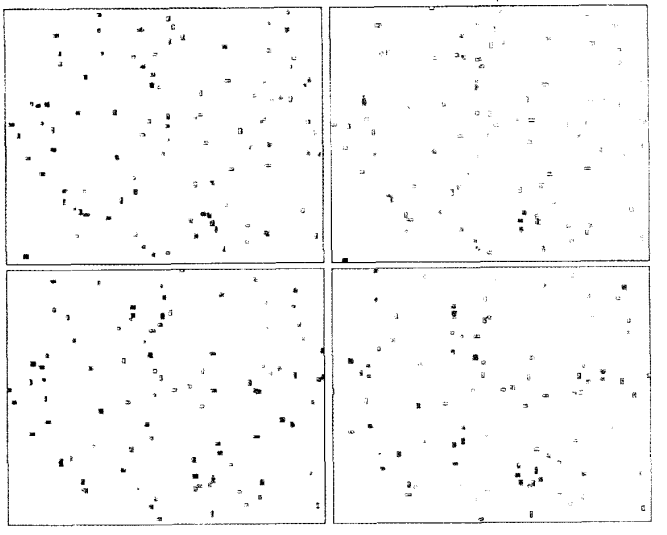
The problem now is to figure out how to use the ordered lists to detect intersections efficiently. However, observe that checking two bounding boxes for overlap is a constant-time operation. It therefore follows that simply checking the bounding boxes $i$ and $j$ for overlap whenever values indexed by $i$ and $j$ (on either the $x$ or $y$ axis) are exchanged will reveal all changes in overlap status in $O(n + c)$ time. That is, $O(n + c)$ work has already occurred in reordering the lists. Asymptotically, no further work is needed because swapping two entries on the list from boxes $i$ and $j$ and checking whether $i$ and $j$ overlap require only constant time. Thus, we can find the change in overlap between boxes in $O(n+c)$ time.

By maintaining a table of overlapping bounding boxes and updating it at each time step in $O(n + c)$ time, the extra work involved is again $O(c - k)$. For the two-dimensional case, extra work can occur if the extents of two bounding boxes change on one coordinate axis without an actual change of their overlap status. In the simulation environment considered here, the extra work has proved to be negligible, and the algorithm runs essentially in time $O(n + k)$. The method described is extremely simple to implement and very fast. In particular, the insertion-sort code is short, with an inner loop of only two or three lines of code.

Table 1 gives the running time of the algorithm for an environment with randomly distributed boxes. Figure 5 shows the configuration of 100 moving boxes at four different times. (There are 10 time steps between each successive configuration in this figure.) The boxes are constrained to stay within a rectangular region 20 units wide and 16 units high. Box widths and heights range from 0.1 to 0.4 unit and are chosen randomly. The starting location of each box is random. Box velocities are also random, with magnitudes ranging between 0 and 0.05 unit per time step and directions uniformly distributed. Overlapping boxes are colored red, while nonoverlapping boxes are green. The running times in Table 1 are based on a constant density of boxes; that is, as the number of boxes increases, the size of the region in which the boxes must remain increases accordingly so that the box density remains constant. The running times reflect the time required by the algorithm (running on a Mips R4000 CPU) to find all changes in overlap status among the boxes for a single time step (exclusive of display time).

**Table 1. Algorithm running times with randomly distributed boxes.**

| No. of Boxes | Running Time (ms) |
|---|---|
| 100 | 0.96 |
| 200 | 2.05 |
| 400 | 4.41 |
| 800 | 9.83 |
| 1,600 | 22.30 |
| 3,200 | 53.30 |



5 Four configurations of 100 randomly moving boxes spaced 10 time steps apart. Overlapping boxes are colored red.

## Advancing the system state

Advancing the system state from some initial state $\mathbf{Y}(t_0)$ to a later state $\mathbf{Y}(t_0 + \Delta t)$ is straightforward in simulations with only bilateral constraints. It is more difficult with contact constraints because of the possibility of collisions.

### *Detecting collisions*

The system advances from an initial state $\mathbf{Y}(t_0)$ to a later state $\mathbf{Y}(t_0 + \Delta t)$ by using the fourth-order Runge-Kutta method to numerically solve $(d/dt)\mathbf{Y}(t) = f(\mathbf{Y}(t))$. The user chooses the step size $\Delta t$ employed by the simulator. (Even though adaptive step sizing would make the simulator more reliable, it was not used because it can cause potentially large variances in the frame rate.) The strategy for detecting collisions is simple, if somewhat error prone: Assuming that the initial state $\mathbf{Y}(t_0)$ is free from intersection, if the state $\mathbf{Y}(t_0 + \Delta t)$ computed by the Runge-Kutta formula is also free from intersection, it is assumed that the system has no collisions on the interval $[t_0, t_0 + \Delta t]$. This is arguably the weakest feature of the simulation system, in that objects having a high velocity compared with the step size $\Delta t$ might "tunnel" through other objects without causing a collision.

Thus, given a computed state $\mathbf{Y}(t_0 + \Delta t)$, the system uses the collision-detection algorithms to check for intersection. If no intersection is found, the simulator then attempts to move the system to time $t_0 + 2\Delta t$. If intersection is found, the simulator must attempt to find the time $t_c$ between $t_0$ and $t_0 + \Delta t$ when the collision first occurred.

### Finding collision times

The simulator treats the problem of finding the collision time $t_c$ as a root-finding problem. For simplicity, let's assume that the interval $[t_0, t_0 + \Delta t]$ has only one collision, and it occurs between a pair of polygons $A$ and $B$.

When the simulator checks $\mathbf{Y}(t_0 + \Delta t)$ for intersection, it finds that polygons $A$ and $B$ intersect. The collision-detection algorithms then try to measure the extent of interpenetration between $A$ and $B$. The current system uses a very simple measure. It checks whether a vertex of one polygon lies inside the other polygon.

Let's consider the case when there is such a vertex on polygon $A$. The interpenetration depth is defined simply as the shortest distance between this vertex and the edges of polygon $B$. Let's adopt the convention that when the vertex lies inside $B$, the interpenetration depth is negative; when it lies outside $B$, the depth is positive.

The system now searches for a configuration having an interpenetration depth equal to zero. However, the simulator has only an initial configuration $\mathbf{Y}(t_0)$ and some derivatives of $\mathbf{Y}$ at intermediate configurations between $t_0$ and $t_0 + \Delta t$ required by the Runge-Kutta method. Given that a collision has occurred between $t_0$ and $t_0 + \Delta t$, some of those derivatives may be incorrect. Nevertheless, the simulator temporarily uses those derivatives to approximate the configuration at any intermediate time $t_0 + h$ for $h < \Delta t$. The approximation is based on interpolating derivatives computed by the Runge-Kutta method. The formula for $\mathbf{Y}(t_0 + \Delta t)$, according to fourth-order Runge-Kutta, is

$$\begin{aligned}
\mathbf{k}_1 &= f(\mathbf{Y}(t_0)) \\
\mathbf{k}_2 &= f(\mathbf{Y}(t_0) + \frac{\Delta t}{2}\mathbf{k}_1) \\
\mathbf{k}_3 &= f(\mathbf{Y}(t_0) + \frac{\Delta t}{2}\mathbf{k}_2) \\
\mathbf{k}_4 &= f(\mathbf{Y}(t_0) + \Delta t\mathbf{k}_3) \\
\mathbf{k}_5 &= \frac{\mathbf{k}_1}{6} + \frac{\mathbf{k}_2}{3} + \frac{\mathbf{k}_3}{3} + \frac{\mathbf{k}_4}{6}
\end{aligned} \qquad (4)$$

followed by

$$\mathbf{Y}(t_0 + \Delta t) = \mathbf{Y}(t_0) + \Delta t\mathbf{k}_5 .$$

While searching for collisions, for $h < \Delta t$ we approximate $\mathbf{Y}(t_0 + h)$ as

$$\mathbf{Y}(t_0 + h) \cong \mathbf{Y}(t_0) + h\left((1 - \frac{h}{\Delta t})\mathbf{k}_1 + \frac{h}{\Delta t}\mathbf{k}_5\right). \qquad (5)$$

Using Equation 5 to compute configurations between $t_0$ and $t_0 + \Delta t$, the simulator begins the root-finding process. The system computes the interpenetration depth $d_1$ between $A$ and $B$ in the configuration $\mathbf{Y}(t_0 + \Delta t)$ given by the Runge-Kutta method. Then it computes the interpenetration depth $d_0$ between $A$ and $B$ in their original configuration $\mathbf{Y}(t_0)$. Since the initial state was free of interpenetration, $d_0$ will be positive, while $d_1$ will be negative. Newton's method is then used to estimate the time of collision as

$$t_e = t_0 + \Delta t \frac{d_0}{d_0 - d_1} .$$

The simulator then computes $\mathbf{Y}(t_e)$ according to Equation 5 and evaluates the interpenetration depth $d_3$ between the features in the state $\mathbf{Y}(t_e)$. If $d_3$ is positive, then $t_e$ is too early, so the simulator must search for a collision in the interval $[t_e, t_0 + \Delta t]$. Similarly, if $d_3$ is negative, then $t_e$ is too late, and the simulator now searches the interval $[t_0, t_e]$. Using the newly computed depth $d_3$ at $t_e$, the simulator makes a new prediction for $t_c$, and the process continues until a time is found at which the interpenetration depth is sufficiently close to zero. (Tolerances are discussed in the later section "Tolerance correction.") After the simulator finds an appropriate time, the regular simulation loop continues as the simulator tries to move forward a full time step to $t_c + \Delta t$.

There are several things to note about this method. First, it is very fast. Computing a configuration at time $t_1$ based on Equation 5 takes little time. Likewise, computing the interpenetration depth between the polygons $A$ and $B$ is also relatively quick. Even if the simulation contains many objects, only $A$ and $B$ are examined for each estimated configuration $\mathbf{Y}(t_e)$ during the root-finding loop. There are several ways in which the root-finding process can at first go astray, but it eventually always finds the time of collision.

### Multiple collisions

We have assumed only a single collision on the interval $[t_0, t_0 + \Delta t]$. Suppose there are actually two or more collisions and, further, the collision between $A$ and $B$ is not the first in this interval. When the system finds a state $\mathbf{Y}(t_c)$ for which $A$ and $B$ have just made contact, the simulator—having apparently resolved the collision between $A$ and $B$—proceeds to completely check all objects for intersection in this state. Upon discovering that two other polygons—say $C$ and $D$—interpenetrate, the simulator begins the process anew, searching for a time prior to $t_c$ when $C$ and $D$ had just collided.

Given any prior collisions, the simulator has wasted time searching for a collision between $A$ and $B$ that is no longer important. (In fact, the collision between $A$ and $B$ may be completely spurious. The earlier collision between $C$ and $D$ may alter events so that $A$ and $B$ no longer collide.) So far, this has occurred rarely. In fact, it is so infrequent that when the simulator checks a proposed state $\mathbf{Y}(t_0 + \Delta t)$ for interpenetration, as soon as two polygons $A$ and $B$ are found to interpenetrate, the intersection check aborts. The system makes no attempt to find other intersections for the proposed state $\mathbf{Y}(t_0 + \Delta t)$. This strategy is usually optimal.

If there are several collisions on an interval $[t_0, t_0 + \Delta t]$, drawing the configuration for each one would slow the display rate. The current system keeps the display rate as constant as possible by refreshing the screen whenever the simulator has moved forward in time by 70 percent or more of a regular time step.

### Ambiguous collisions

The measure of interpenetration depth can also give a misleading answer, or none at all. If polygons $A$ and $B$ have

an intersection such that no vertex of $A$ is contained in $B$ or vice versa, then no measure of the interpenetration depth is given. In this case, the simulator simply estimates a collision time by bisecting the time interval being searched. After some number of bisections, a vertex of one polygon will lie inside the other, as shown in Figure 6.

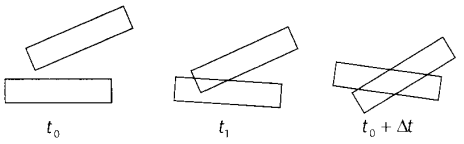
Another possibility is that measuring the shortest distance between a vertex and an edge of the two polygons gives a misleading estimate of the interpenetration depth. Again, when the interval being searched becomes small enough, the interpenetration depth will be computed correctly, as illustrated in Figure 7. Similarly, if two or more vertices occur inside the other polygon, it is not clear which vertex better measures interpenetration. The simulator again takes the simplest strategy, using the first vertex it sees to measure interpenetration depth. In all cases, the simulator guards against slow convergence to the collision time by incorporating bisection steps when the estimated collision time lies near one end of the current interval.

In brief, the root-finding process can be misled in several ways, but the system does not try to guard against these because the occasional expense incurred by a useless computation does not justify the price of a more complicated algorithm.

## Constraint forces

To compute the derivative $(d/dt)\mathbf{Y}(t_0) = f(\mathbf{Y}(t_0))$ of a given state, we need to determine the total force and torque acting on each object. The external forces acting on objects are given, but we must compute the forces that act on objects to maintain the bilateral and contact constraints of the simulation. The simulation system uses an analytical formulation to model bilateral, contact, and friction forces.[4,5] In previous off-line simulation, the computation of these forces has been cast as an optimization problem and solved using various large-scale optimization software packages.[6,7] This section describes the methods used by the current system to speed the determination of these forces. (The current system does not use any of the cited software packages; instead, it solves for the forces using a new algorithm that computes contact, bilateral, and friction forces.[8] The speedups described in this section are mostly orthogonal to the actual method used to determine the forces.)

For simplicity, let's consider the case of frictionless contact, without bilateral constraints. The enhancements apply in a straightforward manner to systems with bilateral constraints and friction forces. The process of computing forces to maintain contact constraints is briefly summarized, followed by a discussion of the speedup methods.



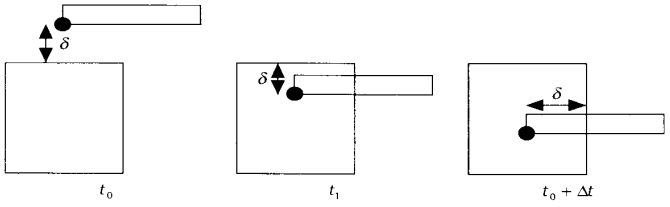

**6 At time $t_0 + \Delta t$, no vertex of either polygon lies inside the other. The bisection process will at some point find a time $t_1$ with $t_0 < t_1 < t_0 + \Delta t$ so that a vertex of one polygon lies inside the other.**

### Contact constraints

Suppose that the state $\mathbf{Y}(t_0)$ represents a configuration with $n$ points of contact between objects. Suppose that the $i$th point of contact occurs between objects $A$ and $B$, and that the unit vector $\mathbf{n}_i$ is normal to the contact surface at the contact point and directed outwards from $B$ towards $A$. If a force of strength $f_i$ is required between $A$ and $B$ to prevent interpenetration, the force will act along $\mathbf{n}_i$. Object $A$ will be subject to a force of $f_i\mathbf{n}_i$ from the contact, and $B$ will be subject to a force $-f_i\mathbf{n}_i$. (The contact force will also contribute to the total torque acting on each object.) Once the strength of all the contact forces is known, $(d/dt)\mathbf{Y}(t_0)$ is trivially computed.

In a second-order system, if two objects are colliding at a point (that is, if they are approaching one another in the $\mathbf{n}_i$ direction), a discontinuous change in their velocities must occur to prevent interpenetration. This change requires the application of an impulsive force (whose units are mass times velocity) to abruptly change the velocities. Conversely, if two objects are moving apart at a contact point, neither an impulsive nor a regular force occurs at that point. In a first-order system, neither possibility occurs because the dynamics equation $f = mv$ dictates that objects have no intrinsic velocity of their own; equivalently, the velocity at any given instant depends completely on the forces acting at that instant. (In a second-order system, velocity is completely independent of the forces in the system at a given instant.) The simulation system uses previously described methods for computing impulsive forces.[4,9] Let's therefore assume that the relative normal velocity between objects at a contact point is zero (although the relative tangential velocity may be nonzero in a second-order system). In a first-order system, the term "acceleration" refers to the velocity induced by the forces in the system.

Let the vector $\mathbf{f} \in \mathbb{R}^n$ denote the collection of $f_i$'s (that is, the $i$th component of $\mathbf{f}$ is $f_i$). The relative normal acceleration $a_i$ between two objects at a contact point is linearly related to $\mathbf{f}$. We *will* adopt the convention that $a_i > 0$ indicates that objects are separating at the $i$th contact, while $a_i < 0$ indicates that objects are accelerating




**7 The interpenetration depth $\delta$ measured at time $t_0 + \Delta t$ gives a misleading reading of the depth. The measurements at times $t_0$ and $t_1 < t_0 + \Delta t$ accurately represent the extent of interpenetration.**

toward one another. Defining $\mathbf{a} \in \mathbb{R}^n$ as the collection of the relative normal accelerations, the relation

$$\mathbf{a} = \mathbf{A}\mathbf{f} + \mathbf{b} \quad (6)$$

may be established, where $\mathbf{A} \in \mathbb{R}^{n \times n}$ and $\mathbf{b} \in \mathbb{R}^n$.[4,10] The matrix $\mathbf{A}$ depends on the inertias and contact geometry of objects. It is computed exactly the same for first- and second-order systems. The vector $\mathbf{b}$ reflects external forces and, in second-order systems, inertial forces due to velocity.

To satisfy the contact constraints, the relative acceleration $a_i$ at each contact point must be nonnegative to prevent objects from accelerating toward one another. The contact force strength $f_i$ must also be nonnegative so that the force is repulsive at each contact point. Finally, energy conservation dictates that if separation occurs at the $i$th contact point (that is, $a_i > 0$), then $f_i$ must be zero. Computing the contact forces therefore involves finding $\mathbf{f}$ such that

$$f_i \geq 0, \quad a_i \geq 0, \quad \text{and} \quad f_i a_i = 0 \quad (7)$$

for all $i$, with $\mathbf{a} = \mathbf{A}\mathbf{f} + \mathbf{b}$. Baraff[8] describes a fast, simple computational method for solving Equation 7. In practice, the method is about two to three times more expensive than simply solving the linear system $\mathbf{A}\mathbf{f} + \mathbf{b} = \mathbf{0}$ (which is how $\mathbf{f}$ would be computed if the contact constraints were instead bilateral constraints).

### Linear subsystems

The real trick to solving Equation 7 is determining which of the $a_i$'s are zero and which are not. Suppose we solve Equation 7 and determine that $k$ of the $a_i$'s are nonzero. (For frictionless systems, the solution $\mathbf{f}$ of Equation 7 may not be unique; however, $\mathbf{a} = \mathbf{A}\mathbf{f} + \mathbf{b}$ is unique.) Since the order of contact points is arbitrary, let's order them so that $a_1 = a_2 = \ldots = a_k = 0$ and $a_i > 0$ for all $k + 1 \leq i \leq n$. Since $f_i$ must be zero when $a_i$ is nonzero, this yields $f_i = 0$ for all $k + 1 \leq i \leq n$. Let us partition $\mathbf{a}$, $\mathbf{f}$, $\mathbf{b}$, and $\mathbf{A}$ by writing

$$\mathbf{a} = \begin{pmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{pmatrix}, \mathbf{f} = \begin{pmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{pmatrix}$$

and

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix}$$

where $\mathbf{a}_1$, $\mathbf{b}_1$, and $\mathbf{f}_1$ are in $\mathbb{R}^k$, $\mathbf{A}_{11} \in \mathbb{R}^{k \times k}$ and $\mathbf{A}_{22}$ is an $(n-k) \times (n-k)$ matrix.

Since the first $k$ components of $\mathbf{a}$ are zero, we have $\mathbf{a}_1 = \mathbf{0}$. Similarly, the last $n-k$ components of $\mathbf{f}$ are zero, so $\mathbf{f}_2 = \mathbf{0}$. Using $\mathbf{a} = \mathbf{A}\mathbf{f} + \mathbf{b}$, we obtain

$$\begin{pmatrix} \mathbf{0} \\ \mathbf{a}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix} \begin{pmatrix} \mathbf{f}_1 \\ \mathbf{0} \end{pmatrix} + \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{pmatrix}$$

yielding

$$\mathbf{A}_{11}\mathbf{f}_1 = -\mathbf{b}_1. \quad (8)$$

Also, note that $\mathbf{a}_2 = \mathbf{A}_{21}\mathbf{f}_1 + \mathbf{b}_2 > \mathbf{0}$.

Suppose that at the next time step, the contact geometry remains the same—that is, we still have the same $n$ contact points. Typically, the matrix $\mathbf{A}$ and vector $\mathbf{b}$ will differ only slightly compared with the previous time step. An obvious thing to do is to partition the new $\mathbf{A}$ and $\mathbf{b}$ and solve the linear system $\mathbf{A}_{11}\mathbf{f}_1 = -\mathbf{b}_1$ (with $\mathbf{A}_{11}$ and $\mathbf{b}_1$ obtained from the new $\mathbf{A}$ and $\mathbf{b}$). The first $k$ components of $\mathbf{f}$ are the newly computed vector $\mathbf{f}_1$, and the last $n-k$ components of $\mathbf{f}$ are set to zero. Similarly, the first $k$ components of $\mathbf{a} = \mathbf{A}\mathbf{f} + \mathbf{b}$ will be zero, and the last $n-k$ components are given by $\mathbf{a}_2 = \mathbf{A}_{21}\mathbf{f}_1 + \mathbf{b}_2$. Clearly, $f_i a_i = 0$ for all $i$. If $\mathbf{f}_1 \geq \mathbf{0}$ and $\mathbf{a}_2 \geq \mathbf{0}$, then we have $\mathbf{a} \geq \mathbf{0}$ and $\mathbf{f} \geq \mathbf{0}$, and we have found a valid solution $\mathbf{f}$. Thus, we may be able to obtain a solution to Equation 7 by simply solving the linear equation $\mathbf{A}_{11}\mathbf{f}_1 = -\mathbf{b}_1$. Since solving linear equations is faster than solving Equation 7, this represents a potentially large computational speedup.

The current simulator exploits this speedup by trying to solve Equation 8 at each time step, based on the partitioning obtained from the previous time step. Whenever the resulting $\mathbf{f}_1$ and $\mathbf{a}_2 = \mathbf{A}_{21}\mathbf{f}_1 + \mathbf{b}_2$ are nonnegative, the more expensive solution algorithm is avoided. However, if the contact geometry changes (one or more contact points are added or deleted), then the system abandons the previous partitioning and immediately uses the more expensive solution method.

### Singular linear subsystems

One problem is that the submatrix $\mathbf{A}_{11}$ of $\mathbf{A}$ may be singular. For example, consider a configuration with four contact points where the relation between $\mathbf{a}$ and $\mathbf{f}$ is

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix} + \begin{pmatrix} -1 \\ -1 \\ -1 \\ 1 \end{pmatrix}.$$

One solution $\mathbf{f}$ for this problem is $f_1 = f_2 = f_3 = 1/3$ and $f_4 = 0$, which yields $\mathbf{a} = (0, 0, 0, 1)$. However, $f_1 = f_3 = 1/2$ and $f_2 = f_4 = 0$ also yields $\mathbf{a} = (0, 0, 0, 1)$.

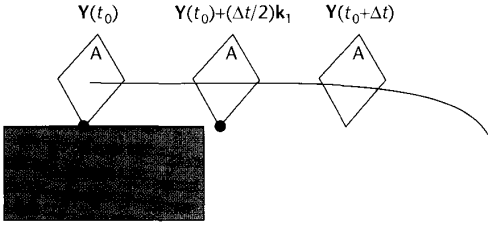
In this case, if we partition the system as above, we would simply be solving

$$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 2 \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix}. \quad (9)$$

However, this linear system is singular (multiplying the second row by 2 and subtracting the first row yields the third row). The problem lies not so much in solving Equation 9, since a solution $\mathbf{f}$ exists, but in being certain to obtain a particular type of solution.

Suppose that the procedure we used to solve Equation 9 computed the solution $f_1 = f_3 = 2, f_2 = -3$, and $f_4 = 0$. This is a valid solution for Equation 9, but not the solution we want, since $f_2 < 0$. In this particular case, this solution would still yield the correct result $\mathbf{a} =$

(0, 0, 0, 1). In general, though, if we solve and find some $f_i < 0$, this might indicate that our partitioning of Equation 7 is incorrect. Whenever $\mathbf{A}_{11}$ is singular, there is no general way of knowing when a negative $f_i$ indicates an incorrect partition (requiring the more complete solution algorithm to be run).




**8 The marked vertex of object *A* contacts the marked edge of object *B* in the initial state $\mathbf{Y}(t_0)$. At the intermediate state $\mathbf{Y}(t_0) + \Delta t/2\mathbf{k}_1$, the contact geometry has not been updated. At time $t_0 + \Delta t$, the contact geometry is updated, and *A* realizes it no longer has any support.**

We can, however, attempt to guide the numerical procedure used to solve Equation 8. This is particularly important for systems with friction where different solutions of $\mathbf{f}$ might yield different accelerations $\mathbf{a}$.[5] In this case, it is important to make the solutions $\mathbf{f}$ computed over time as continuous as possible. Given that the complete algorithm for solving Equation 7 computes a particular solution $\mathbf{f}$ for the system, if we compute $\mathbf{f}$ for the next time step by solving Equation 8, we would like the newly computed $\mathbf{f}$ to be close to the previously computed $\mathbf{f}$.

The simulation system solves Equation 8 using the factorization

$$\mathbf{RA}_{11} = \mathbf{U}$$

of $\mathbf{A}_{11} \in \mathbb{R}^{k\times k}$, where $\mathbf{R} \in \mathbb{R}^{k\times k}$ is invertible, and $\mathbf{U} \in \mathbb{R}^{k\times k}$ is upper triangular. To solve $\mathbf{A}_{11}\mathbf{f}_1 = -\mathbf{b}_1$, the system first computes $\mathbf{c} = -\mathbf{Rb}_1$ and then solves the triangular system

$$\mathbf{Uf} = \mathbf{c} \qquad (10)$$

(because if $\mathbf{Uf} = \mathbf{c}$, then $\mathbf{A}_{11}\mathbf{f} = (\mathbf{R}^{-1}\mathbf{U})\mathbf{f} = \mathbf{R}^{-1}\mathbf{c} = -\mathbf{b}$).

If $\mathbf{A}_{11}$ is singular, then $\mathbf{U}$ will be singular as well. Let $q$ denote the rank deficiency of $\mathbf{A}_{11}$. By permuting the order of the contact points, we can always assume that Equation 10 can be partitioned in the form

$$\begin{pmatrix} \mathbf{U}_1 & \mathbf{U}_2 \\ \mathbf{0} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{pmatrix}$$

where $\mathbf{U}_1 \in \mathbb{R}^{(k-q)\times(k-q)}$ is upper triangular and invertible, $\mathbf{U}_2 \in \mathbb{R}^{(k-q)\times q}$, and $\mathbf{f}_1$ is broken into two smaller vectors $\mathbf{x}_1 \in \mathbb{R}^{k-q}$, and $\mathbf{x}_2 \in \mathbb{R}^q$. Because of the structure of the problems we are solving, Equation 10 is always compatible,[9] that is, the last $q$ components of $\mathbf{c}$ will always be zero. Since $\mathbf{U}_1$ is invertible, we have complete freedom in our choice of $\mathbf{x}_2$: Given any $\mathbf{x}_2$, we determine $\mathbf{x}_1$ by solving

$$\mathbf{U}_1\mathbf{x}_1 = \mathbf{c}_1 - \mathbf{U}_2\mathbf{x}_2.$$

We can make the newly computed $\mathbf{f}$ similar to the previous $\mathbf{f}$ by simply setting $\mathbf{x}_2$ equal to the corresponding components of the previous solution of $\mathbf{f}$. Setting $\mathbf{x}_2$ in this manner completely determines $\mathbf{x}_1$. This allows us to find the same mode of solution whether we compute $\mathbf{f}$ by solving Equation 8 or by solving the more expensive system of Equation 7. This technique lets us deal with singular, compatible systems in a very robust manner. The same discussion applies to solving Equation 8 using similar factorization techniques such as LU or QR decompositions.

### *Road-runner physics*

Solving the linear system of Equation 8 is faster than solving Equation 7. However, whenever the contact geometry changes, we cannot solve Equation 8 in place of Equation 7. One simplification in the current simulator fixes the contact geometry at the beginning of each Runge-Kutta step. I call this technique "road-runner physics," for reasons that will be obvious shortly.

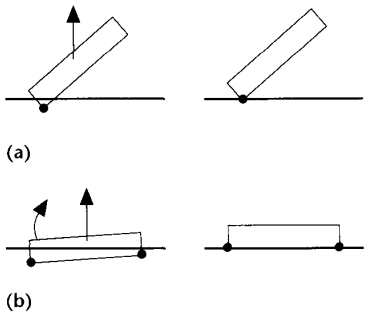
The Runge-Kutta solution method requires four evaluations of $d\mathbf{Y}/dt$, which in turn requires solving for $\mathbf{f}$ four times. Since the collision-detection process is not run until the system checks the proposed new state $\mathbf{Y}(t_0 + \Delta t)$, the contact geometry is unknown for the intermediate states $\mathbf{Y}(t_0) + (\Delta t/2)\mathbf{k}_1$, $\mathbf{Y}(t_0) + (\Delta t/2)\mathbf{k}_2$, and $\mathbf{Y}(t_0) + \Delta t\mathbf{k}_3$ used in Equation 4. Rather than run the contact-determination procedure to determine the contact geometry for each state, the current system simply adopts the contact geometry of the initial state $\mathbf{Y}(t_0)$. This gives a speedup in two ways: by avoiding extra contact determination and by ignoring some contact changes that would require us to solve Equation 7 from scratch.

This introduces error into the simulation. Consider Figure 8. At state $\mathbf{Y}(t_0)$, the contact-determination procedure identifies the marked vertex of object *A* as lying on the marked edge of object *B*. This contact state is presumed to be in effect until state $\mathbf{Y}(t_0 + \Delta t)$ is computed. At the intermediate evaluations of Equation 4, object *A* will behave as if supported by object *B*. This technique is called road-runner physics because object *A* behaves somewhat like the cartoon character: It doesn't begin to fall until it arrives at a new state $\mathbf{Y}(t_0 + \Delta t)$, "looks down," and realizes that it now has no support.

The matrix $\mathbf{A}$ computed for an intermediate state reflects the new positions of the objects. For example, in Figure 8, the new position of the marked vertex of object *A* is taken into account at the intermediate state $t_1$ when $\mathbf{A}$ is computed. However, that vertex is still treated as if it is in contact with the marked edge. In practice, no discernible error has occurred from this simplification.

## Tolerance correction

Over the course of simulation, numerical error inevitably accumulates in the constraints that the system is supposed to maintain. In a simulation system with only bilateral constraints, it is not too difficult to keep the constraint error bounded by introducing force-feedback terms into the constraint equations.[10] The force-feedback terms function as very mild spring forces whose




9 Tolerance corrections (drawn with exaggerated scale). (a) A correction that is a pure translation. (b) A correction consisting of both a translation and a rotation.

strength depends on the amount of constraint error and its first derivative. In a system with contact constraints, keeping the constraint error under control is even more important, but unfortunately more difficult.

As described earlier, the collision/contact detection routines are responsible for reporting when contact or interpenetration occurs between two objects. Clearly, this decision must be made with some error tolerance $\varepsilon$ in mind. (The current system has a default value for $\varepsilon$ that users can override. In the default display environment, the width of the window is 20 units, and the default value for $\varepsilon$ is 0.02. On a monitor with 1,000-pixel horizontal resolution, a distance of $\varepsilon = 0.02$ is about one pixel.) If two objects do not approach within a distance $\varepsilon$ of one another, the objects are declared not to contact one another. If the objects are less than $\varepsilon$ apart and have an interpenetration depth of $\varepsilon$ or less, then they are declared to have contacted one another. Objects with an interpenetration depth greater than $\varepsilon$ are declared to be intersecting.

The root-finding process used to find the time of a collision between two objects terminates when a state is found such that objects contact one another within the tolerance $\varepsilon$.

### Alternative approaches

The root-finding process can potentially leave the simulation system in a very precarious state. Suppose that in trying to resolve a collision between objects $A$ and $B$, a state is found for which $A$ and $B$ interpenetrate with a depth just slightly less than $\varepsilon$. This is acceptable to the root-finding process: It has resolved the collision to within the specified numerical tolerance, and the system can now begin enforcing a contact constraint between $A$ and $B$.

However, a very small numerical drift can now cause $A$ and $B$ to interpenetrate with a depth greater than $\varepsilon$. This would cause the collision-detection routine to hunt continually for a supposed collision between $A$ and $B$, when in fact there was none.

One obvious remedy is to force the root-finding process to refine its estimate of the collision time until the interpenetration depth is no greater than some stricter tolerance $\varepsilon_s < \varepsilon$. This partially alleviates the problem, at the expense of making collisions costlier to resolve and the entire process much more complex. For example, while checking for intersection, suppose two objects have an interpenetration depth less than $\varepsilon$ but greater than $\varepsilon_s$. If the interpenetration at the point in question represents a new contact being formed, the simulation must back up to when the interpenetration depth is less than $\varepsilon_s$. However, if the interpenetration represents a previous existing contact, numerical drift has caused the interpenetration depth to increase from less than $\varepsilon_s$ to greater than $\varepsilon_s$, which does not signal a collision.

Thus, collision detection decisions now have a temporal component. This results in an extremely complex control structure for a collision-resolution algorithm that is not especially reliable. (I used this scheme in my off-line simulation system.[8] Debugging the collision-resolution algorithm was an ongoing struggle for an embarrassingly long period of time. The resulting code is ridiculously complex and insufficiently reliable for an interactive system.)

The interactive system uses a completely different scheme for dealing with numerical tolerances. No attempt is made to trap collisions to some finer tolerance than $\varepsilon$ or to add force-feedback terms to the contact constraint equations (although force-feedback is applied to the bilateral constraints). Instead, objects are displaced periodically whenever the interpenetration depth increases past some warning threshold $\varepsilon_w$. (Currently, we choose $\varepsilon_w = 3/4\varepsilon$.) We call this displacement a *tolerance correction*. Tolerance corrections occur fairly infrequently in our simulation environment (generally less than once every 5 to 10 seconds).

### Computing the tolerance correction

A tolerance correction occurs after the system checks a proposed state for interpenetration and finds it to be valid, except that two or more objects have an interpenetration depth between $\varepsilon_w$ and $\varepsilon$. For simplicity, let's denote the newly valid state as $\mathbf{Y}(t_0)$. Suppose that the interpenetration depth between a pair of objects $A$ and $B$ is $\delta$, with $\varepsilon_w < \delta < \varepsilon$. We would like to displace $A$ and $B$ so that, in their new positions, the interpenetration depth between $A$ and $B$ becomes zero. If only translational displacements were used (as illustrated in Figure 9a), we could do this exactly. However, if rotational displacements are required (as in Figure 9b), determining the exact rotational displacement requires solving nonlinear kinematics equations. The current system treats the displacements as virtual motions, thereby linearizing the problem. The displacements obtained are therefore correct only to within first order. This is not a problem, however, because the displacement magnitudes are always small.

However, we must be careful that the displacements do not introduce new errors into other contact constraints (or bilateral constraints) in the system. Depending on the currently active constraints, the system might have to displace many objects. In Figure 10, for example, the interpenetration depth between objects $B$ and $C$ is not large enough to require correction. However, the error between $A$ and $B$ is sufficiently large to call for a tolerance correction. Since all three objects ($A$, $B$, and $C$) must be displaced, the error between $B$ and $C$ is reduced to zero. In contrast, it is not necessary to adjust objects $D$ and $E$, because they are disjoint from $A$, $B$, and $C$.

Let's regard the simulation environment as a first-order world momentarily. In a first-order world, the state $\mathbf{Y}(t_0)$ contains purely geometrical information and no velocity information. We need to find a small displacement $\Delta\mathbf{Y}$ such that the displaced state $\mathbf{Y}(t_0) + \Delta\mathbf{Y}$ has a constraint error of zero at several constraints. Let $\mathbf{A}$ be the matrix described earlier for contact constraints. Previously, the vector $\mathbf{b}$ in Equation 6 depended on external forces and inertial forces (for second-order worlds). In this section, the value of $b_i$ will be the error in the $i$th constraint of the system. (If the separation depth at a contact is $\delta$, then we set $b_i$ to $\delta$. If the interpenetration depth at a contact is $\delta$, we set $b_i$ to $-\delta$. The error in a bilateral constraint is also signed.) We will compute constraint force magnitudes by treating all the contact constraints as bilateral constraints and solving the linear equation

$$\mathbf{Af} = -\mathbf{b}.$$

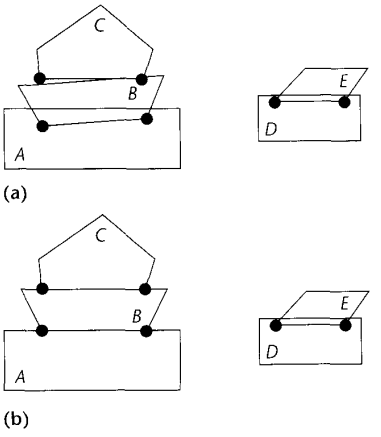
In a first-order system, we would ordinarily compute $(d/dt)\mathbf{Y}(t_0)$ based on the forces specified by $\mathbf{f}$, in addition to the external forces acting. For a tolerance correction, we compute $(d/dt)\mathbf{Y}(t_0)$ solely on the basis of $\mathbf{f}$ just computed, ignoring any external forces. We define the displacement $\Delta\mathbf{Y} = (d/dt)\mathbf{Y}(t_0)$ and then instantaneously displace the system from $\mathbf{Y}(t_0)$ to $\mathbf{Y}(t_0) + \Delta\mathbf{Y}$. Apart from the error due to linearization, the constraint error in the configuration $\mathbf{Y}(t_0) + \Delta\mathbf{Y}(t_0)$ will be zero (for the constraints we have just corrected). We can proceed with the simulation from this new state. (If the simulation environment is second-order, we continue with the same velocities that the objects had prior to being displaced.)

### Physical justification

This method is only one way of computing the displacement $\Delta\mathbf{Y}$. However, it has a very natural physical justification. Clearly, the object's displacements should be as small as possible, subject to reducing the constraint error to zero, but how should we measure the magnitude of a given displacement $\Delta\mathbf{Y}$? If the object displacements are limited to translations, we can simply define the displacement magnitudes by using the Euclidean norm to measure the length of a vector.

Is it clear, however, that the translation of each object in the system should be weighted equally? For example, suppose object $A$ is very large and massive compared to object $B$. In this case, it might be sensible to let object $B$ have a larger displacement than object $A$. In some cases, it is necessary to both translate and rotate an object (Figure 9b). In this case, how should translational displacements compare with rotational displacements?

In general, there is no single "correct" way to measure displacement. It is an application-dependent issue. One sensible criterion is to make the displacement measure coordinate-invariant—that is, whether we measure rotation and translation in radians and meters or in radians and centimeters, we should arrive at the same displacements. The method described here satisfies this criterion.



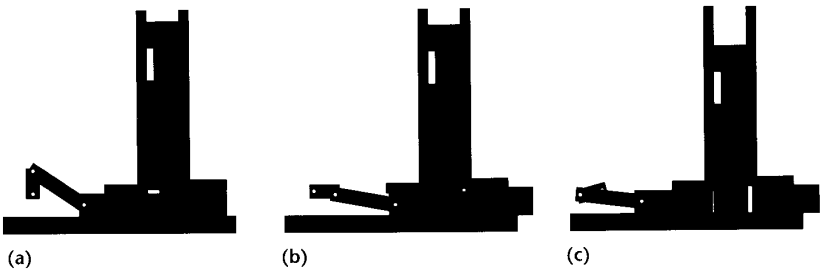

**10 (a) An interpenetration depth greater than $\varepsilon_w$ exists between *A* and *B*, and triggers a tolerance correction involving *A*, *B*, and *C*. (b) In the corrected configuration, objects *D* and *E* have not been displaced.**

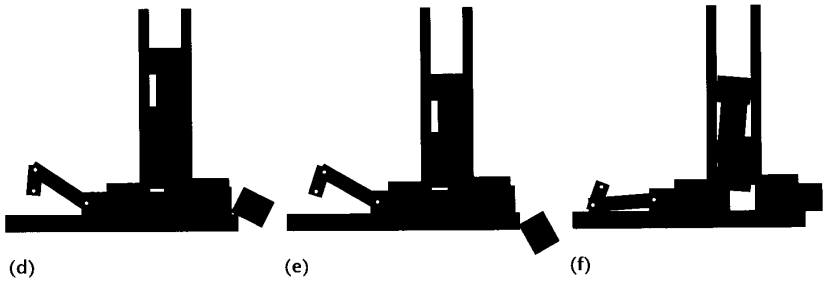
It can be shown that the displacement $(d/dt)\mathbf{Y}(t_0)$ obtained above is the minimum-length displacement, as measured in the kinetic-energy norm of the system, that reduces the constraint error to zero. This is a particularly natural choice for measuring displacement magnitudes, because the constraint forces are computed according to the exact same principle. (Baraff and Mattikalli[11] showed that the actual acceleration of a system with contact constraints is at any moment the acceleration of smallest magnitude that satisfies the constraints of the system, where magnitude is measured according to the kinetic-energy norm.) The advantage of this approach is that it requires almost no extra coding. The matrix $\mathbf{A}$ is the same as it is for constraint forces (Equation 6), and the same procedures are used to compute $\mathbf{f}$.

Using the kinetic energy norm to find the smallest displacement also benefits from the small displacements that massive objects undergo. Although the use of the kinetic energy norm is not strictly necessary (we could regard all objects as having the same mass and moment of inertia for the purposes of tolerance correction), it seems a natural choice, given that lighter objects tend to move more than heavier objects in general.

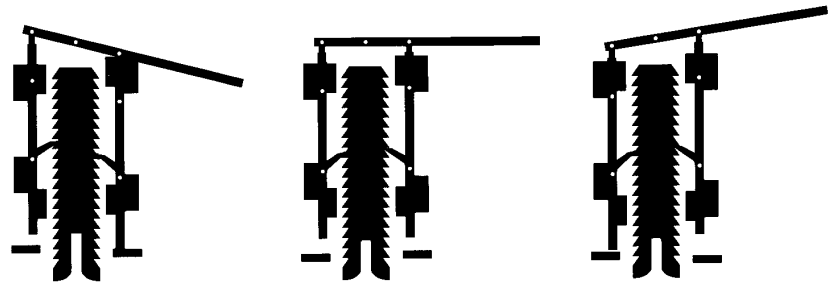
Minimizing according to the kinetic-energy norm is particularly appropriate, because the current system removes translational or rotational degrees of freedom by treating the mass and moment of inertia, respectively, as infinite. (Since $\mathbf{A}$ reflects only the reciprocals of those quantities, this approach is numerically stable. The reciprocals are just taken to be zero. This is much more efficient than freezing objects by computing constraint forces, and simpler than removing degrees of freedom by redefining the state vector $\mathbf{Y}$.) Thus, the displacement $(d/dt)\mathbf{Y}(t_0)$ computed using the method described here never translates an object with infinite mass or rotates an object with infinite moment of inertia.

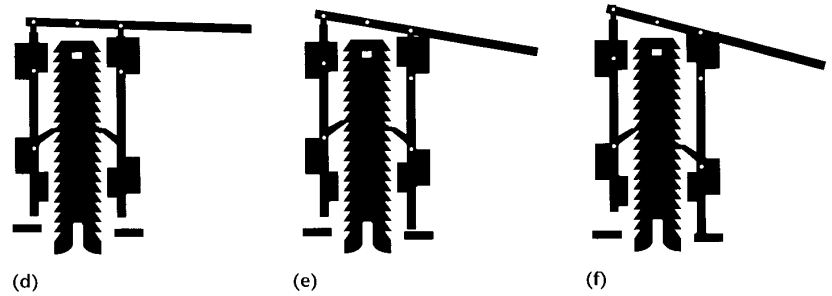
For example, in Figure 9a, if neither translation nor rotation is constrained, the actual tolerance correction would consist of a combined rotational and translational displacement. A pure translational displacement (as drawn) would occur only if the object had an infinite moment of inertia.




11 Time-lapse simulation sequence of a block feeder.




12 Time-lapse simulation sequence of a double-action jack.

## Results

The tolerance-correction facility has proved most important to the reliability of the system. The linear systems solution technique has decreased the solution time for the simulator and helped maintain a smooth solution for systems with friction. As built, the simulation world of the interactive system described here is 2 1/2-dimensional: Objects are planar but can be assigned to different "layers." No contact or collision occurs between pairs of objects unless they are in the same layer. Figures 11 and 12 show simulations of two mechanisms. Fixed objects are colored in black. Objects in different levels are different colors. White circles indicate a bilateral point-to-point constraint between two objects. In Figure 12, the green circles indicate contact points. Both systems can be simulated robustly at a consistent frame rate of between 20 and 30 Hz on a low-end Silicon Graphics workstation. ■

## References

1. E. Sacks and L. Joskowicz, "Automated Modeling and Kinematic Simulation of Mechanisms," *Computer-Aided Design*, Vol. 25, No. 2, Feb. 1993, pp. 106-118.
2. D. Baraff, "Curved Surfaces and Coherence for Nonpenetrating Rigid Body Simulation," *Computer Graphics* (Proc. Siggraph), Vol. 24, No. 4, Aug. 1990, pp. 19-28.
3. M.C. Lin and J.F. Canny, "A Fast Algorithm for Incremental Distance Calculation," *Proc. Int'l Conf. Robotics and Automation*, IEEE Computer Society Press, Los Alamitos, Calif., 1991, pp. 1,008-1,014.
4. P. Lötstedt, "Numerical Simulation of Time-Dependent Contact Friction Problems in Rigid Body Mechanics," *SIAM J. Scientific Statistical Computing*, Vol. 5, No. 2, 1984, pp. 370-393.
5. D. Baraff, "Issues in Computing Contact Forces for Nonpenetrating Rigid Bodies," *Algorithmica*, Vol. 10, 1993, pp. 292-352.
6. P. Gill et al., "User's Guide for QPSOL: A Fortran Package for Quadratic Programming," Tech. Report Sol84-6, System's Optimization Laboratory, Dept. of Operations Research, Stanford Univ., Palo Alto, Calif., 1984.
7. R.E. Marsten, "The Design of the XMP Linear Programming Library," *ACM Trans. Mathematical Software*, Vol. 7, No. 4, 1981, pp. 481-497.
8. D. Baraff, "Fast Contact Force Computation for Nonpenetrating Rigid Bodies," *Computer Graphics* (Proc. Siggraph), Vol. 28, July 1994, pp. 23-34.
9. M.T. Mason and Y. Wang, "Modeling Impact Dynamics for Robotic Operations," *Proc. Int'l Conf. Robotics and Automation*, IEEE Computer Society Press, Los Alamitos, Calif., 1987, pp. 678-685.
10. R. Barzel and A.H. Barr, "A Modeling System Based on Dynamic Constraints," *Computer Graphics* (Proc. Siggraph), Vol. 22, No. 4, Aug. 1988, pp. 179-188.
11. D. Baraff and R. Mattikalli, "Impending Motion Direction of Contacting Rigid Bodies," Tech. Report CMU-RI-TR-93-15, Carnegie Mellon Univ., 1993.



***David Baraff*** *is an assistant professor in Carnegie Mellon University's Robotics Institute and in its School of Computer Science. His research interests include physical simulation and modeling for computer graphics, robotics, and animation. Baraff received his BsE from the University of Pennsylvania in 1987 and his PhD from Cornell University in 1992, both in computer science. In 1995, he was named an Office of Naval Research Young Investigator.*

*Readers may contact Baraff at Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213, e-mail baraff@cs.cmu.edu.*