Partitioned Dynamics

David Baraff Andrew Witkin

CMU-RI-TR-97-33

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

March 1997

Contents

List of Figures

Abstract

Most physically-based simulation systems for computer graphics target only a single simulation domain, such as particle systems, rigid bodies, cloth, and liquids. By using "layering" techniques, one-sided interactions between different domains are easily produced. For example, one-sided particle system/rigid-body simulation is achieved by first running a rigid-body simulator and then injecting the rigid body motions into a particle-system simulator; in the particle system, particles rebound off the (possibly moving) solid objects without affecting the solid objects' motion. This paper address the problem of combining disparate simulations so as to allow two-way interactions, such as a jet of particles that both deflects off a stack of solid objects, but also causes the stack to topple over realistically. Incorporating multiple simulation domains within a single simulation system is difficult because each simulation problem is best attacked using specialized techniques. Instead, we propose a method that treats each simulator as a "black box" with a simple generic interface. We present a technique called interleaved simulation that achieves geometrically accurate behavior with minimal overhead, compared to the cost of running the simulations independently. The method works best where the component systems' masses differ significantly. We show that interleaved simulation is equivalent to taking one matrix-solving gradient step per simulation time step, with a very good preconditioner. We include an easy-to-implement description of the method, and present a variety of simulation results.

1 Introduction

Our motivation in writing this paper stems from our recent experiences in developing physically-based cloth simulation. To model cloth realistically we must model its interactions with other objects, such as rigid bodies. Initially, we intended to handle these interactions unilaterally, letting non-cloth objects exert forces on cloth, but not vice versa. We would begin with animated or captured motion, or pre-computed simulation results, then add a cloth "layer" to the simulation. This layered approach makes good physical sense when the non-cloth objects so far outweigh the cloth that the latter's effect on the former can be neglected [9]. This is often enough a good approximation; notably, the dynamics of clothing generally have a small effect on the wearer's motion (sheath skirts and straight-jackets not withstanding.) We realized, however, that many interactions are decidedly bilateral, for instance a cloth sack full of fruit, clothes swinging on a hanger or clothes line, or a person jumping on a trampoline.

In previous work we have built a variety of simulation systems: particle-system simulators [16], rigid-body simulators [1] and simulators for flexible objects [17, 2]. Others have simulated fluids and wind [10, 15].

We realized that it would be extremely useful to merge all of these disparate simulation domains into a single framework, allowing mutual simultaneous interactions. For instance, combining particle systems with rigid objects would allow us to spray a jet of particles at a stack of objects, both deflecting the particle stream and toppling the stack. We could have objects bob, or sink, in simulated water, etc. Many of these effects cannot be adequately handled by layered one-sided interactions.

1.1 Difficulties of Combined Simulation

Combining disparate simulators into a single, monolothic simulation system is problematic for various reasons. Simulation code (either in source or as a binary library) is not always available, particularly in production environments where the majority of code used is obtained commercially. Even if the software is available, the cost in programmer-time to merge two simulation systems will often be prohibitively expensive. Additionally, merging simulation capabilities in this manner leads to "quadratic software explosion" in which software must be written to deal with all the permutations in which simulators might be combined.

However, the difficulties go far beyond logistical issues. Algorithmically, handling all these disparate simulation domains within a single monolithic simulation system is all but impossible, because for efficiency each must be attacked with specialized techniques. Particle-system simulators are specialized to handle large ensembles of homogeneous simple objects, with simple interactions. Rigid-body simulators deal with smaller numbers of more complex objects, and must handle the extremely complex interactions that arise due to contact constraints. Soft-body simulation focuses on modeling volumetric deformations, while cloth simulation must cope with arbitrarily complex folding and wrinkling patterns of deformable surfaces, and with the highly stiff interactions resulting from the internal forces.

These differences go far deeper than the implementation details: the differential-equation solving methods we use to handle rigid-body systems differ fundamentally from those that are suited to cloth, or to fluids. While we could in principle construct a "least-common-denominator" simulator, the resulting weak methods would cost literally orders of magnitude in performance degradation. In addition to these fundamental differences, each system gains greatly in performance by exploiting regularities in the objects it models, and we have the practical issue that we would like to use existing specialized systems, many of which have by now been highly optimized for performance and robustness.

1.2 A Modular Approach

These considerations argue for an approach to combining simulation domains that treats each simulation system as a "black box," handling the interactions through some kind of simple, uniform, generic interface. We have investigated this approach, and in this paper we report our initial results.

Combining simulation systems from the outside, while more promising than building a monolithic simulator, is by no means an easy task. Handling some interactions, such as "weak" spring forces, is a simple matter of externally computing the interaction forces, and injecting them into each simulator. More difficult to handle are interactions involving contact and other geometric constraints, because they require a simultaneous constraint-force solution. Typically, constraint forces are obtained by solving a global matrix equation, which would seem to cut right across the simulation boundaries, and into the heart of each system.

Our specific goal is to compute constraint forces across interacting systems, requiring minimal knowledge of the systems' internals, with acceptable accuracy and only minimal performance degradation, when compared to the cost of running each simulator independently. We first show that the global constraint-force equations can be solved iteratively using standard matrix methods, given little more than the ability to apply forces and take time steps. This method does not meet our performance expectations since many iterations might be required to obtain an acceptably accurate result. We then show that we can obtain results that are quite accurate, at least in the sense that the constraints are satisfied geometrically, without the need for multiple iterations per time step, provided that some or all of the systems are capable of handling constraints individually.

The mathematical framework that produces this result suggests a related family of solution methods for the general problem. We introduce one particular solution method, *interleaved simulation*, which achieves our goals. Interleaved simulation handles a broad class of simulation domains, with negligible overhead, and good accuracy over a wide range of situations. The method works best where there are significant (but not necessarily extreme) mass disparities between the systems. We demonstrate the success of interleaved simulation with several simulations that show complex, realistic two-way interactions between cloth and rigid bodies, and between rigid bodies and particle systems. We note that while the time to implement each individual simulator ranged from days to months, the implementation time of the interleaved simulation method was measured in hours. A concrete easy-to-implement description of the combination method is included.

1.3 Overview

A standard approach to imposing geometric constraints is to calculate a set of *constraint forces* whose job it is to keep the constraints satisfied—keeping joints attached, preventing interpenetration, etc.—by ensuring that objects' accelerations are consistent with the constraints. To solve for forces that produce the desired accelerations, we need to know the functional relationship between forces and accelerations. Ideally, this relationship can be captured in a *mass matrix*. In practice, though, a simulation system's effective response to outside forces depends not only on its mass distribution, but on internal constraints and forces, and even on the solution method that is used to step forward through time. Rather than delving into these internal complexities and combining them across disparate systems, we can explore the force/acceleration relationship simply by applying forces, instructing the system to take a step, and seeing what it does: the resulting change in velocity gives an approximation to acceleration. As we will show in detail later on, this procedural approach gives us a way to multiply a proposed constraint-force vector by the constraint matrix, without ever explicitly forming that matrix, even when the constraints span multiple "black box" systems.

The ability to perform matrix-vector multiplies is all that we need to employ standard iterative techniques to solve for the constraint forces. We can thus impose constraints across multiple partitioned systems, with minimal access to their internals, though perhaps with greater computational cost than we would like to incur, since many iterations may be required.

We next show how the cost of multiple iterations per time step can be avoided, provided one or more of the simulation systems can be made to satisfy constraints individually (a common simulation capability). We then describe a specific instance of the approach, called *interleaved simulation*.

Interleaved simulation, in both detail and implementation, is a simple method, and we present a rough sketch of the method before proceeding: labeling two interacting systems S_a and S_b, we first instruct S_b to take a step, without regard to the constraints coupling the two systems. System S_a is then instructed to take a step consistent with S_b's already computed motion. In the course of doing so, S_a internally computes a constraint force, to which we assume we have access. No simultaneous solution across systems is required, because S_a treats S_b's motion as given. We then step S_b again, but this time apply the previously computed constraint force to S_b. System S_a then takes another constrained step, again feeding its computed constraint force back to S_b, and the cycle repeats. This procedure treats S_a and S_b asymmetrically, and it works best if there is in fact a significant mass disparity between the two systems. Interleaved simulation has the desirable property that the constraint solution is geometrically accurate, even if there is error in the dynamics; empirically it proves to work well over a wide range of conditions. Section 3 gives a mathematical interpretation of interleaved simulation in terms of the framework developed in section 2.2; this interpretation explains the success of the method.

2 Solving For Constraint Forces

We have said informally that we want to treat each simulation system as a "black box", but what information do we really need to extract from a simulation system in order to impose constraints on it? To constrain the behavior of a point on a body, we need a "handle" on the point that follows it as it moves. We will need to query not only the point's position, but its velocity, and we will also need the ability to apply forces to it. Constraints may involve other geometric features such as normals, distances, or areas. To avoid becoming embroiled in the geometric details, we will treat the entire set of such "handles" extracted from a system as a single vector, with position denoted by $\mathbf{x}$, velocity $\dot{\mathbf{x}}$, acceleration $\ddot{\mathbf{x}}$ and applied force $\mathbf{F}$. In addition to operations on geometric handles, we assume that we can instruct the system to take a time step of size Δt, updating the values of $\mathbf{x}$ and $\dot{\mathbf{x}}$, and that we can tell the system to retract a time step, returning to its former state.

2.1 Constraint Force Formulation

Typically, in the context of dynamic simulation, constraints are expressed as sets of simultaneous conditions on bodies' accelerations. Starting with a vector of position-dependent conditions $\mathbf{C}(\mathbf{x}) = \mathbf{0}$, we obtain the corresponding acceleration constraints by differentiating twice with respect to time. If we define $\mathbf{J} = \partial\mathbf{C}/\partial\mathbf{x}$, and $\dot{\mathbf{J}} = \dot{\mathbf{x}}^T(\partial^2 C/\partial\mathbf{x}^2)$, the acceleration conditions, by two applications of the chain rule, are

$$\ddot{\mathbf{C}} = \mathbf{J}\ddot{\mathbf{x}} + \dot{\mathbf{J}}\dot{\mathbf{x}} = \mathbf{0}. \tag{1}$$

In a dynamic simulation, accelerations depends on an externally applied force $\mathbf{F}$ according to

$$\ddot{\mathbf{x}} = \mathbf{M}^{-1}\mathbf{F} + \mathbf{d},$$

where $\mathbf{M}^{-1}$ is an inverse mass matrix and $\mathbf{d}$ is the handle's acceleration due to internal forces. To enforce the acceleration conditions we must compute a constraint force $\mathbf{F}$ that, when combined with the internal forces, produces an acceleration that satisfies $\ddot{\mathbf{C}} = \mathbf{0}$. This condition alone does not uniquely determine the constraint force, but the additional assumption that the constraint force is *passive* (in the sense that it does no work, neither adding nor removing kinetic energy) suffices to determine a unique solution. This extra condition can be enforced by requiring the constraint force to have the form $\mathbf{F} = \mathbf{J}^T\boldsymbol{\lambda}$, where $\boldsymbol{\lambda}$ is an unknown vector of *Lagrange multipliers*. Instead of solving directly for the force, we solve for $\boldsymbol{\lambda}$. Substituting into equation (1) yields

$$\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T\boldsymbol{\lambda} = -(\mathbf{J}\mathbf{d} + \dot{\mathbf{J}}\dot{\mathbf{x}}).$$

This linear system is solved for $\boldsymbol{\lambda}$, which is then multiplied by $\mathbf{J}^T$ to obtain the constraint force $\mathbf{F}$.

2.2 Constraints On Multiple Systems

Suppose now that we have two independent simulation systems S_a and S_b, with geometric handles $\mathbf{x}_a$ and $\mathbf{x}_b$, and a vector of constraints $\mathbf{C}(\mathbf{x}_a, \mathbf{x}_b) = \mathbf{0}$ coupling the two systems. Let $\mathbf{J}_a = \partial\mathbf{C}/\partial\mathbf{x}_a$ and $\mathbf{J}_b = \partial\mathbf{C}/\partial\mathbf{x}_b$. The systems' handles evolve according to the relations

$$\ddot{\mathbf{x}}_a = \mathbf{M}_a^{-1}\mathbf{F}_a + \mathbf{d}_a \qquad \text{and} \qquad \ddot{\mathbf{x}}_b = \mathbf{M}_b^{-1}\mathbf{F}_b + \mathbf{d}_b$$

where

$$\mathbf{F}_a = \mathbf{J}_a^T\boldsymbol{\lambda} \qquad \text{and} \qquad \mathbf{F}_b = \mathbf{J}_b^T\boldsymbol{\lambda}$$

are the constraint forces applied to S_a and S_b respectively, and $\mathbf{d}_a$ and $\mathbf{d}_b$ are the handles' accelerations due to internal forces. Defining

$$\mathbf{A} = \mathbf{J}_a\mathbf{M}_a^{-1}\mathbf{J}_a^T, \qquad \mathbf{B} = \mathbf{J}_b\mathbf{M}_b^{-1}\mathbf{J}_b^T$$

and

$$\mathbf{D} = -(\mathbf{J}_a\mathbf{d}_a + \mathbf{J}_b\mathbf{d}_b + \dot{\mathbf{J}}_a\dot{\mathbf{x}}_a + \dot{\mathbf{J}}_b\dot{\mathbf{x}}_b),$$

the combined acceleration condition is simply

$$(\mathbf{A} + \mathbf{B})\boldsymbol{\lambda} = \mathbf{D}. \tag{2}$$

Equation (2) when solved for $\boldsymbol{\lambda}$ yields the constraint forces on both systems. Unfortunately, solving equation (2) is problematic. Of the quantities that comprise $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{D}$, some—$\mathbf{J}_a$, $\dot{\mathbf{J}}_a$, $\mathbf{J}_b$, $\dot{\mathbf{J}}_b$—involve only the relation between the geometric handles and the constraints we have imposed. These lie entirely outside the simulation systems, and are therefore accessible to us. However, $\mathbf{M}_a^{-1}$, $\mathbf{M}_b^{-1}$, and system S_a and S_b's internal accelerations $\mathbf{d}_a$ and $\mathbf{d}_b$ lie inside the black box.

In practice, these internal quantities can be far more complex than they might at first appear, involving internal constraints, inertial effects, and even vagaries of the differential equation solvers, since we can only really observe the systems' time stepping behavior, rather than true instantaneous accelerations. In fact, these quantities may often only exist implicitly, in the relation between input forces and resultant state changes, rather than as explicit internal matrix/vector datastructures. It is therefore all but hopeless to expect every simulation system we wish to combine to provide these quantities to us explicitly.

2.3 Iterative Solution

We do not have explicit access to the inverse mass matrices $\mathbf{M}_a^{-1}$ and $\mathbf{M}_b^{-1}$, nor the internal handle accelerations $\mathbf{d}_a$ and $\mathbf{d}_b$. We can however obtain access to all of these quantities procedurally. We know that $\ddot{\mathbf{x}}_a = \mathbf{M}_a^{-1}\mathbf{F}_a + \mathbf{d}_a$. To approximately evaluate $\ddot{\mathbf{x}}_a$ for a given $\mathbf{F}_a$, we instruct the simulator S_a to take a step of size Δt with force $\mathbf{F}_a$ acting on the system, and observe the change in velocity, $\Delta\dot{\mathbf{x}}_a$. Then

$$\ddot{\mathbf{x}}_a \approx \frac{\Delta\dot{\mathbf{x}}_a}{\Delta t}.$$

Computing $\mathbf{d}_a$ is straightforward: we evaluate $\ddot{\mathbf{x}}_a$ with $\mathbf{F}_a = \mathbf{0}$. The acceleration $\mathbf{d}_b$ is computed similarly.

Having evaluated $\mathbf{d}_a$, we have procedural access to $\mathbf{M}_a^{-1}$, in the sense that we can compute the product $\mathbf{M}_a^{-1}\mathbf{F}_a$ for arbitrary $\mathbf{F}_a$. To do so, we apply the force $\mathbf{F}_a$, evaluate $\ddot{\mathbf{x}}_a$, and subtract $\mathbf{d}_a$, yielding $\mathbf{M}_a^{-1}\mathbf{F}_a$. We can now compute $\mathbf{A}\boldsymbol{\lambda}$ for any $\boldsymbol{\lambda}$; since $\mathbf{J}_a$ is known to us, we compute $\mathbf{F}_a = \mathbf{J}_a^T\boldsymbol{\lambda}$, then procedurally compute the product $\mathbf{M}^{-1}\mathbf{F}_a$, then multiply by $\mathbf{J}_a$ to obtain $\mathbf{A}\boldsymbol{\lambda}$. The product $\mathbf{B}\boldsymbol{\lambda}$ is computed similarly using simulator S_b. Computing $\mathbf{D}$, given $\mathbf{d}_a$ and $\mathbf{d}_b$ is also straightforward.

In summary, we can evaluate $(\mathbf{A}+\mathbf{B})\boldsymbol{\lambda}$ and the right hand side, $\mathbf{D}$. The ability to multiply $\mathbf{A}+\mathbf{B}$ by an arbitrary vector is all that is required to apply a variety of standard iterative matrix solution methods, such as Gauss-Seidel or conjugate gradient [11], to solve equation (2) for $\boldsymbol{\lambda}$. This iterative method gives a general solution for coupling simulation systems with constraints, but at considerable computational cost. Each evaluation of $(\mathbf{A}+\mathbf{B})\boldsymbol{\lambda}$ incurs the full cost of a time step. Using the conjugate gradient method, for example, would require $O(n)$ such evaluations, where n is the number of constraints.

2.4 Avoiding "In-place" Iteration

Typically, iterative matrix solution methods continue until an error tolerance is met. In our context, this means taking time steps at each iteration, then retracting those steps to return to the previous step. If $\boldsymbol{\lambda}$ is changing relatively slowly over time, it might be possible to accelerate the solution by spreading the matrix-solver iterations over time, allowing the simulation clock to advance with each iteration rather than resetting the state. This is equivalent to using the previous solution for $\boldsymbol{\lambda}$ as an initial guess in the current iteration (almost always a good idea), but terminating the solver after a single iteration (not necessarily a good idea).[1]

2.5 Constraint "Repair"

As we consider solution techniques that might introduce appreciable error, it is worth noting that constraint force solutions can err in two very different ways, corresponding to the two conditions that we imposed on the solution: $\ddot{\mathbf{C}} = \mathbf{0}$, and the "passivity" condition $\{\mathbf{F}_a = \mathbf{J}_a^T\boldsymbol{\lambda}, \quad \mathbf{F}_b = \mathbf{J}_b^T\boldsymbol{\lambda}\}$. The first condition captures the geometry of the constraints; violations will take the form of separated joints, interpenetrating objects, and so forth. Violations of the second condition are more subtle in their effect: constraint forces that satisfy both conditions are the minimal forces needed to "do the job" of enforcing the constraints. Forces satisfying the first but not the second condition enforce the acceleration constraint $\ddot{\mathbf{C}} = \mathbf{0}$, but with additional stray forces that may add or remove energy from the system, giving correct geometry but incorrect dynamics.

[1] Note that if we do spread the solution process out over time, then reducing the step size of the simulation corresponds closely to increasing the number of iterations per step. In general then, increased accuracy is simply obtained by reducing the time step, obviously with commensurate expense.

Both geometric and dynamic error can be very disturbing if sufficiently large, but in our experience people are far less tolerant of geometric error than of dynamic error.

Constrained simulation techniques are well known, and many simulation systems—our own included—support a variety of constraints, such as point-to-point attachment, contact, etc. When these capabilities are available in one or more of the simulation systems that we wish to combine, we can improve substantially on the general combination method described above, by "repairing" low-quality constraint force estimates.

Suppose that a simulation system allows us to impose constraints on its "handles," to control individual points, attach points together, etc. A typical interface might allow us to define constraints by providing procedures that evaluate $\mathbf{C}(\mathbf{x})$, $\mathbf{J}(\mathbf{x})$, and $\dot{\mathbf{J}}(\mathbf{x}, \dot{\mathbf{x}})$, using the notation introduced previously for constraint functions and their derivatives. Internally, the system would compute its constraint forces, invoking these procedures as needed, resulting in a time step that respects the constraints.

Having computed an inaccurate $\boldsymbol{\lambda}$, we can perform the "repair" step as follows: first, use the computed value to step one of the systems, say S_b, forward, obtaining a value $\ddot{\mathbf{x}}_b$. Then, use this value as an input to S_a's constraint solver, commanding it to take a step consistent with S_b's motion. In terms of the notation developed above, system S_a would internally be solving for a revised vector, $\boldsymbol{\lambda}^{(r)}$ that satisfies $\mathbf{A}\boldsymbol{\lambda}^{(r)} = \mathbf{D} - \mathbf{B}\boldsymbol{\lambda}$. This repairs the geometric error, because the resulting handle accelerations satisfy $\ddot{\mathbf{C}} = 0$; however, since the constraint forces on S_a and S_b are based on different values of $\boldsymbol{\lambda}$, the "passivity" condition is no longer guaranteed to hold.

3 Interleaved Simulation

Let us summarize the situation up to this point. We have cast combined simulation with constraints as the problem of solving at each time step an equation $(\mathbf{A} + \mathbf{B})\boldsymbol{\lambda} = \mathbf{D}$, with $\boldsymbol{\lambda}$ describing the constraint forces acting between the simulations. Our ability to access $\mathbf{A}$ and $\mathbf{B}$ is restricted to procedurally performing matrix-vector products $\mathbf{A}\boldsymbol{\lambda}$ or $\mathbf{B}\boldsymbol{\lambda}$, which in turns limits us to iterative solution methods. Since each matrix-vector product is expensive, keeping the number of iterations low is important. We already have two ideas toward this goal. First, whenever we decide to stop iterating, we can repair the "geometric" error of the constraints as described in section 2.5, although this still leaves us with some "dynamic" error. Because reducing geometric error is more important for animation, the ability to repair constraints lets us stop iterating much sooner than we could without it. Second, we can reduce the number of iterations by advancing the simulation clock on *each* iteration, thereby avoiding in-place iteration, again at the cost of some error. The interleaved simulation method we sketched in section 1.3 uses both of these techniques.

There is one more factor to consider in reducing the number of iterations. All iterative methods can be improved (that is, achieve faster convergence) by use of a good preconditioner. A good preconditioner for a system $\mathbf{M}\mathbf{x} = \mathbf{b}$, is a matrix $\mathbf{C}$ that is close to $\mathbf{M}^{-1}$; given $\mathbf{C}$, we solve $(\mathbf{C}\mathbf{M})\mathbf{x} = \mathbf{C}\mathbf{b}$ for $\mathbf{x}$. As with any iterative method, the "perfect" preconditioner would be $\mathbf{C} = \mathbf{M}^{-1}$, which insures convergence in a single step—but of course, if $\mathbf{C} = \mathbf{M}^{-1}$ were available in the first place, there would be no need to use an iterative solution technique.

As we noted in section 2.5, the ability to constrain each system individually allows us in effect to solve equations $\mathbf{A}\boldsymbol{\lambda} = \mathbf{b}$ for an arbitrary vector $\mathbf{b}$. This ability in turn gives us procedural access to $\mathbf{A}^{-1}$. We have similar procedural access to $\mathbf{B}^{-1}$. Since we have access to these inverses, we should consider using them in some fashion to help precondition the iterative solution method. Unfortunately, there is no universal method for combining $\mathbf{A}^{-1}$ and $\mathbf{B}^{-1}$ into a good approximation of $(\mathbf{A} + \mathbf{B})^{-1}$; a good combination will be situation dependent. In particular, for simulations where there is a mass disparity between systems, it will turn out that either $\mathbf{A}^{-1}$ or $\mathbf{B}^{-1}$ alone is a good preconditioner.

To see this, consider a particle system S_a, interacting with a rigid-body system S_b, with the rigid bodies outweighing the individual particles. Since the rigid bodies are comparatively heavy, the rigid-body system's *inverse* mass matrix, $\mathbf{M}_b^{-1}$, is much smaller than the particles system's inverse matrix $\mathbf{M}_a^{-1}$. Thus, $\mathbf{B} = \mathbf{J}_b\mathbf{M}_b^{-1}\mathbf{J}_b^T$ is much smaller than $\mathbf{A} = \mathbf{J}_a\mathbf{M}_a^{-1}\mathbf{J}_a^T$. If the difference between $\mathbf{A}$ and $\mathbf{B}$ is significant, then $\mathbf{A}^{-1}$ becomes a reasonable approximation to $(\mathbf{A}+\mathbf{B})^{-1}$; that is, it becomes sensible to use $\mathbf{A}^{-1}$ as a preconditioner in solving $(\mathbf{A}+\mathbf{B})\boldsymbol{\lambda} = \mathbf{D}$. (Note that in the limit when the rigid bodies are infinitely heavy, that is, completely unaffected by the particles, then $\mathbf{M}_b^{-1}$ is zero. As a result, $\mathbf{B}$ is zero, and $\mathbf{A}^{-1}$—which is the same as $(\mathbf{A}+\mathbf{B})^{-1}$—is the perfect preconditioner.)

The question now is how to effectively take advantage of $\mathbf{A}^{-1}$ as a preconditioner. It turns out that interleaved simulation, which already avoids in-place iteration and uses constraint-repair, also exploits $\mathbf{A}^{-1}$ as a preconditioner, though this is not obvious. To see this, we must cast the interleaved simulation method in terms of section 2.2's iterative framework.

As previously described, the $(i+1)$th time step of the interleaved simulation involves both S_a and S_b taking their own individual steps, with system S_b going first. System S_b ignores the constraints that couple it to system S_a, but subjects itself to a force $\mathbf{J}_b^T\boldsymbol{\lambda}^{(i)}$, with $\boldsymbol{\lambda}^{(i)}$ the constraint force computed by system S_a during the previous step. (We will take $\boldsymbol{\lambda}^{(0)}$ to be zero.) After S_b finishes its step, the acceleration of its handle vector, $\mathbf{x}_b$, is fixed: we have $\ddot{\mathbf{x}}_b = \mathbf{M}_b^{-1}\mathbf{J}_b^T\boldsymbol{\lambda}^{(i)} + \mathbf{d}_b$.

After computing this result, the acceleration $\ddot{\mathbf{x}}_b$ is imported to system S_a, as system S_a takes its $(i+1)$th step. System S_b's acceleration has now been determined for the ith step, and nothing system S_a does will change it. Given S_b's now determined motion, system S_a computes a new vector $\boldsymbol{\lambda}^{(i+1)}$ that will satisfy the motion constraint $\ddot{\mathbf{C}} = \mathbf{0}$. The accelerations at the end of the $(i+1)$th step are

$$\ddot{\mathbf{x}}_a = \mathbf{M}_a^{-1}\mathbf{J}_a^T\boldsymbol{\lambda}^{(i+1)} + \mathbf{d}_a \quad \text{and} \quad \ddot{\mathbf{x}}_b = \mathbf{M}_b^{-1}\mathbf{J}_b^T\boldsymbol{\lambda}^{(i)} + \mathbf{d}_b.$$

What value does system S_a compute for $\boldsymbol{\lambda}^{(i+1)}$? Remember that system S_a treats system S_b's motion as fixed, and tries to find a motion that will not violate the constraints. Since the acceleration of S_b's handles $\ddot{\mathbf{x}}_b$ are computed in terms of $\boldsymbol{\lambda}^{(i)}$ and *not* $\boldsymbol{\lambda}^{(i+1)}$, the correct condition for system S_a to satisfy is not $(\mathbf{A}+\mathbf{B})\boldsymbol{\lambda}^{(i+1)} = \mathbf{D}$, but instead

$$\mathbf{A}\boldsymbol{\lambda}^{(i+1)} + \mathbf{B}\boldsymbol{\lambda}^{(i)} = \mathbf{D}.$$

Since $\boldsymbol{\lambda}^{(i)}$ is already known, system S_a simply takes a step that enforces the constraint

$$\mathbf{A}\boldsymbol{\lambda}^{(i+1)} = \mathbf{D} - \mathbf{B}\boldsymbol{\lambda}^{(i)}. \tag{3}$$

The newly computed $\boldsymbol{\lambda}^{(i+1)}$ is then fed back to system S_b as the cycle continues.

Let us express $\boldsymbol{\lambda}^{(i+1)}$ in terms of $\mathbf{A}$, $\mathbf{B}$ and $\boldsymbol{\lambda}^{(i)}$. Solving equation (3) for $\boldsymbol{\lambda}^{(i+1)}$, we obtain

$$\begin{aligned}
\boldsymbol{\lambda}^{(i+1)} &= \mathbf{A}^{-1}(\mathbf{D} - \mathbf{B}\boldsymbol{\lambda}^{(i)}) \\
&= \mathbf{A}^{-1}(\mathbf{D} - \mathbf{B}\boldsymbol{\lambda}^{(i)}) + (\boldsymbol{\lambda}^{(i)} - \boldsymbol{\lambda}^{(i)}) \\
&= \mathbf{A}^{-1}(\mathbf{D} - \mathbf{B}\boldsymbol{\lambda}^{(i)} - \mathbf{A}\boldsymbol{\lambda}^{(i)}) + \boldsymbol{\lambda}^{(i)} \\
&= \mathbf{A}^{-1}(\mathbf{D} - (\mathbf{A}+\mathbf{B})\boldsymbol{\lambda}^{(i)}) + \boldsymbol{\lambda}^{(i)}.
\end{aligned} \tag{4}$$

If not for the factor $\mathbf{A}^{-1}$ in equation (4), the change in $\boldsymbol{\lambda}$ at each step would be $\mathbf{D} - (\mathbf{A}+\mathbf{B})\boldsymbol{\lambda}^{(i)}$. This particular iterative update is the method of "steepest descent," and is known to converge to the answer, provided that the eigenvalues of $\mathbf{A}+\mathbf{B}$ are less than one [7].[2] Of course, steepest descent is not particularly effective on its own; however, equation (4) describes a preconditioned steepest descent iteration, with $\mathbf{A}^{-1}$ as

[2]By simply scaling the update, that is, by writing $\boldsymbol{\lambda}^{(i+1)} = \boldsymbol{\lambda}^{(i)} + s(\mathbf{D} - (\mathbf{A}+\mathbf{B})\boldsymbol{\lambda}^{(i)})$ where s is a suitably small number, the method can always be made to converge, assuming $\mathbf{A}+\mathbf{B}$ is positive definite. However the convergence can be slow, if s needs to be very small.

the preconditioner. Mathematically then, each step of interleaved simulation implements one step of a preconditioned steepest descent iteration; as we have noted, $\mathbf{A}^{-1}$ is a good preconditioner for mass-disparate simulations. Our results in section 5 show the empirical success of the method.

4 Implementation

We tested our theories on partitioned simulation by implementing the interleaved simulation method. This section gives an overview of the simulation systems used, followed by an exact description of the interface between the simulation systems. Following this, we present the results for several combined simulation runs.

Particle-System Simulator

Particle systems were first described by Reeves [12]; a tutorial overview of particle system dynamics and implementation can be found in Witkin [16]. We implemented a very basic particle-system simulator for use in combination simulation.

Our particle-system simulator computes the motion of particles subject to gravity and other field forces. No particle/particle interactions were implemented. The simulator takes into account collisions between the particles and solid objects with kinematically defined motion paths. Solid objects are implemented very simply as triangles, with each triangle's vertex locations enumerated at fixed points over time. (If the system is instructed to step forward from time t_0 to time t_1, it is assumed that all vertices have a defined position at some time before t_0 and some time after t_1. The system determines vertex locations at times between t_0 and t_1 by linear interpolation). Particles bounce off triangles they collide with; a particle's change in velocity at each collision is determined by a standard simple collision law [16].

Cloth Simulator

Cloth simulation has been an active area in computer graphics for years. Many simulation paradigms have been proposed; a recent comprehensive survey of the field is given by Hing and Grimsdale [8]. In recent work, Breen *et al*. [3] adopted a particle-based model of cloth in conjunction with real-world data for cloth energy functions, producing very realistic examples of cloth draped on solid objects. Eberhardt *et al*. [6] extended Breen *et al*.'s work by having the system produce animation results (Breen *et al*.'s system was geared toward producing only final static resting poses). Eberhardt *et al*. describe their use of Maple [5] to generate optimized source code for derivative expressions. Their results improve upon Breen *et al*.'s; in particular, Eberhardt *et al*. report achieving an average running time of between 21 CPU minutes and 27 CPU minutes per frame of animation for a 52×52 particle mesh (on an SGI R8000 processor).

The integration method used by the above two systems employs an explicit integration step; this severely limits the size of the time steps that can be taken. The cloth-animation system described by Volino *et al*. [14] (including a predecessor system described by Carignan *et al*. [4]) also uses explicit integration techniques. In contrast, much earlier work by Terzopoulos *et al*. [13] on deformable surfaces and solids treats cloth differently. Terzopoulos *et al*. attacked cloth using an implicit integration method, which allows a step size that is often orders of magnitude large than the steps that can be taken by explicit methods. The complication here is that each step of the implicit method requires formulating and solving a matrix system of size $n \times n$, where n is the number of variables used to represent the cloth's spatial pose. This step is obviously expensive. For example, a 50×50 node cloth grid (with each node constituting three variables) requires the formation and solution of a $7,500 \times 7,500$ matrix at each step. This matrix, though obviously

quite large, turns out to be extremely sparse. Unfortunately, the sparsity pattern mirrors the mesh topology, so that a direct banded solver (such as banded Cholesky) cannot be used.

We have achieved promising results in cloth simulation by building a simulator that uses an underlying particle-system base to represent the cloth surface, and a triangular mesh topology for describing the bend/stretch/shear energy functions that give cloth its behavior. The triangular mesh also defines the geometry of the cloth for collision detection purposes, and allows for non-rectangular boundaries and arbitrary surface topologies. The simulator steps forward in time using the backward Euler implicit integration technique, coupled with sparse matrix storage methods and a sparse conjugate gradient solution method [11]. The conjugate gradient algorithm solves the sparse $n \times n$ matrix system generated by the implicit solver step in nearly linear time. The system detects and responds to cloth/cloth contact using spring forces (which are also implicitly integrated). The system also allows for solid objects with scripted motion. Cloth particles which contact a solid object are prevented from interpenetrating the object by imposing motion constraints on the cloth particles.

In initial experiments with systems of size up to 14,000 particles, we have found that the simulation's running time is nearly independent of the material parameters of the cloth, and comes very close to being linear in the number of particles being simulated. As a timing comparison, the system can simulate the motion of a 51×51-particle grid (resulting in a 2,621 particle/5,054 triangle mesh) with a running time of approximately 3–4 CPU seconds per step on an R10000/199Mhz SGI Indigo-2 processor. (A suitable static draping pose of this size mesh over solid objects can be achieved after perhaps two to three minutes of CPU time.) Collision detection (of both the cloth with itself and with the solid objects) accounts for less than 10% of the system's running time; the majority of the simulator's effort goes into formulating and solving the sparse matrix system underlying the implicit integration step. The system usually requires 1–2 steps per frame of animation. More precise timings are given in the next section.

Rigid-Body Simulator

For the rigid-body simulations, we used the CORIOLIS™ rigid-body simulation system. CORIOLIS is the 3D extension of the 2D rigid-body simulator described in Baraff [1]. CORIOLIS simulates arbitrarily shaped polyhedral rigid bodies, with an emphasis on persistent contact, collision and friction. In our combination simulations, the running time spent by CORIOLIS in computing rigid-body motion is negligible compared with the running time of the other simulators.

4.1 Collision Detection

To enforce contact constraint requires collision detection. If neither simulator supports collision detection internally, some external agent must take responsibility for examining the simulator's states (through their handles) and formulating the proper constraints. Fortunately, many (if not most) simulation systems support some kind of collision detection and response. Our own simulators each support collisions between dynamic objects (those whose motion is computed by the simulator) and kinematic objects (those whose motion is scripted). When this is the case, essentially all of the work of collision detection and the associated constraint formulation for the interleaved simulation can be delegated to the individual simulators. The main additional requirement is that each simulator be able to export its geometry over the course of the simulation. (Our rigid-body simulator, for example, exports the geometric shape of each its body once, at the beginning of the simulation. At the conclusion of each step thereafter, the simulator exports only the initial and final position and velocity of each body's reference frame, for use by the other simulator.)

4.2 Combining Particle Systems with Rigid Bodies

We combined particle-system motion with rigid-body motion by designating the particle-system simulator as simulation system S_a, and the rigid-body simulator as system S_b. We chose a fixed step size Δt for the problem that seemed consistent with the time-scale of the interactions between the particles and the rigid bodies. Each simulation step began with the rigid-body simulator advancing its system clock from time t_0 to $t_0 + \Delta t$. Once the step was completed, the output motion was described to the particle-system simulator in terms of the initial and final positions and velocities of each rigid body. The particle-system simulator used the initial and final position to define the motion trajectory of each kinematic triangle over the range t_0 to $t_0 + \Delta t$. The particle simulator then computed the motion of the particles, and summed the total constraint forces exerted on the particles as they collided with the rigid bodies. [3] This force was then fed back to the rigid-body simulator for use at the next step.

The constraint forces needed by the particle-system simulator are trivial to compute. Suppose that the particle-system detects a collision between a particle of mass m and a rigid body. If the particle system alters the particle's velocity by an amount $\Delta\mathbf{v}$ due to the collision, then the constraint *impulse* acting on the particle is $m\Delta\mathbf{v}$. Even though the collision was instantaneous, it is reasonable to regard the constraint impulse instead as a force that acted for time Δt on the particle. Thus, we can regard the particle as being subject to the constant constraint force $m\Delta\mathbf{v}/\Delta t$ over the simulation interval. Since an opposite constraint force acts on the rigid body, the particle-system simulator records that the body struck by the particle has been subject to a force of $-m\Delta\mathbf{v}/\Delta t$. If the collision takes place at world-space point $\mathbf{p}$, and the rigid body's center of mass has world-space location $\mathbf{c}$, then the rigid-body is also subject to a torque of $(\mathbf{p} - \mathbf{c}) \times (-m\Delta\mathbf{v}/\Delta t)$.

Rather than reporting each collision separately, the particle-system keeps track of the net constraint force and torque due to collision for each rigid body. Whenever a body is struck by a particle, the particle system adds the constraint force and torque into the running sum for that body. At the end of the simulation step, the particle system exports the net constraint force and torque exerted on each body due to particle collisions (and then zeros out these sums in preparation for the next time step). The rigid-body simulator takes the net constraint force and torque for each body and applies those forces and torques over the next time step of the simulation. New motion is computed for the rigid bodies, and the cycle continues.

4.3 Combining Cloth with Rigid Bodies

We combined our cloth simulator with our rigid-body simulator in much the same manner. The only significant difference between the two combinations involved the manner in which the cloth simulator determined the constraint force acting on the cloth. To advance forward in time, our cloth simulator solves an equation of the form $\mathbf{K}\Delta\dot{\mathbf{Y}} = \mathbf{f}$ where $\Delta\dot{\mathbf{Y}}$ is a vector giving the change in the cloth-particle velocities during the step, $\mathbf{f}$ is the force apart from the constraints, and $\mathbf{K}$ is the sparse, square matrix determined by the implicit solver. However, if the ith cloth particle is constrained in some manner (the simulator supports contact, friction, and fixed-point attachments with objects), then the ith component of $\mathbf{K}\Delta\dot{\mathbf{Y}} - \mathbf{f}$ may not be zero. This difference, if it is not zero, tells us the net constraint force that acted on the ith cloth particle. Again, once this constraint force is known, the cloth system adds the opposite force (and corresponding torque) into the running sum of the appropriate rigid body.

[3] For each collision at time t, with $t_0 < t < t_0 + \Delta t$, the colliding triangle's velocity was computed by linear interpolation between the triangles initial and final velocities during the step. This interpolated velocity was used in determining the response of the particle due to the collision.

5 Results

The interleaved simulation method gave excellent results for the cloth/rigid-body and particle-system/rigid-body combinations we tried. We see no obstacle to combining all three simulation systems. The method could also be applied to merge volumetric flow models (for example, a volumetric representation of water flow, or finely calculated wind effects) with solid objects, either deformable or rigid, which would then react and influence other.

The overhead imposed by interleaved simulation is negligible; the time for combined simulation is almost exactly the same as running each individual simulation system by itself. The rigid-body simulator accounts for approximately 40% of the total running time in the particle/rigid-body simulation of figure 1. The sequence in figure 1 has a running time of 0.23 CPU seconds per frame, on a single SGI R10000/199Mhz processor.

For the cloth/rigid-body combined simulations, the rigid-body system accounts for no more than 1% of the total run time. The sequence in figure 2, involving a 5,054 triangle mesh piece of cloth, has a running time of 3.9 CPU seconds per frame of animation, on a single SGI R10000/199Mhz processor. Figure 3 involves a 1,088 triangle mesh with a running time of 1.27 CPU seconds per frame. Finally, figure 4, in which rigid posts support a cloth surface which in turn supports some rigid spheres, illustrates very complex and subtle interactions. The running time for this simulation, with 3,017 cloth particles, organized as a 5,802 triangle mesh, is 5.7 CPU seconds per frame of animation.

We also believe that interleaved simulation might be applied as a partitioning method for homogeneous systems, such as rigid-body simulation. In order to compute contact forces between objects, the CORIOLIS simulator formulates and solves a modified linear complementarity problem (LCP), which involves an $O(n) \times O(n)$ matrix, for situations involving n contact points.[4] Although this matrix is usually sparse, it is difficult for the LCP solution process to exploit this sparsity, and dense methods are used instead. This leads to an $O(n^3)$ solution method. However, if it is possible to partition the simulation into two separate simulations, substantial savings could result: solving two $n/2 \times n/2$ matrix systems with an $O(n^3)$ method is four times faster than solving a single problem of size $n \times n$. If the problem can be broken into even more partitions, the savings could be higher. In some cases, it is possible that interleaved simulation might not yield adequate results; however, the framework developed in section 2 suggests several ways to refine the accuracy of the solutions to achieve a suitable result.

6 Acknowledgments

This research was supported in part by an ONR Young Investigator award, an NSF CAREER award, and NSF grants MIP-9420396, IRI-9420869 and CCR-9308353.

[4] This is assuming that that the n contacts occur among a single cluster of connected bodies. The matrix equations for disconnected clusters of bodies decouple and CORIOLIS exploits this.

References

[1] D. Baraff. Interactive simulation of solid rigid bodies. *IEEE Computer Graphics and Applications*, 15:63–75, 1995.

[2] D. Baraff and A. Witkin. Dynamic simulation of non-penetrating flexible bodies. *Computer Graphics* (*Proc. SIGGRAPH*), 26:303–308, 1992.

[3] D.E. Breen, D.H. House, and M.J. Wozny. Predicting the drape of woven cloth using interacting particles. *Computer Graphics* (*Proc. SIGGRAPH*), pages 365–372, 1994.

[4] M. Carignan, Y. Yang, N. Magenenat-Thalmann, and D. Thalmann. Dressing animated synthetic actors with complex deformable clothes. *Computer Graphics* (*Proc. SIGGRAPH*), pages 99–104, 1992.

[5] B.W. Char, K.O. Geddes, H.G. Gaston, B.L. Leong, M.B. Monagan, and S.M. Watt. *Maple V*. Springer-Verlag, 1991.

[6] B. Eberhardt, A. Weber, and W. Strasser. A fast, flexible, particle-system model for cloth draping. *IEEE Computer Graphics and Applications*, 16:52–59, 1996.

[7] G. Golub and C. Van Loan. *Matrix Computations*. John Hopkins University Press, 1983.

[8] N.N. Hing and R.L. Grimsdale. Computer graphcis tehcniques for modeling cloth. *IEEE Computer Graphics and Applications*, 16:28–41, 1996.

[9] J.K. Hodgins, W.L. Wotten, D.C. Brogan, and J.F. O'Brien. Animating human athletics. *Computer Graphics* (*Proc. SIGGRAPH*), pages 71–78, 1995.

[10] M. Kass and G. Miller. Rapid, stable fluid dynamics for computer graphics. *Computer Graphics* (*Proc. SIGGRAPH*), pages 49–58, 1990.

[11] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes*. Cambridge University Press, 1986.

[12] W.T. Reeves. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2:91–108, 1983.

[13] D. Terzopoulos, J.C. Platt, and A.H. Barr. Elastically deformable models. *Computer Graphics* (*Proc. SIGGRAPH*), 21:205–214, 1987.

[14] P. Volino, N. Magnenat-Thalmann, S. Jianhua, and D. Thalmann. An evolving system for simulating clothes on virtual actors. *IEEE Computer Graphics and Applications*, 16:42–51, 1996.

[15] J. Wejchert and D. Haumann. Animation aerodynamics. *Computer Graphics* (*Proc. SIGGRAPH*), pages 19–22, 1991.

[16] A. Witkin. *An Introduction To Physically Based Modeling*, chapter Particle System Dynamics. SIGGRAPH Course Notes, ACM SIGGRAPH, 1995.

[17] A. Witkin and W. Welch. Fast animation and control of nonrigid structures. *Computer Graphics* (*Proc. SIGGRAPH*), 24:243–252, 1990.

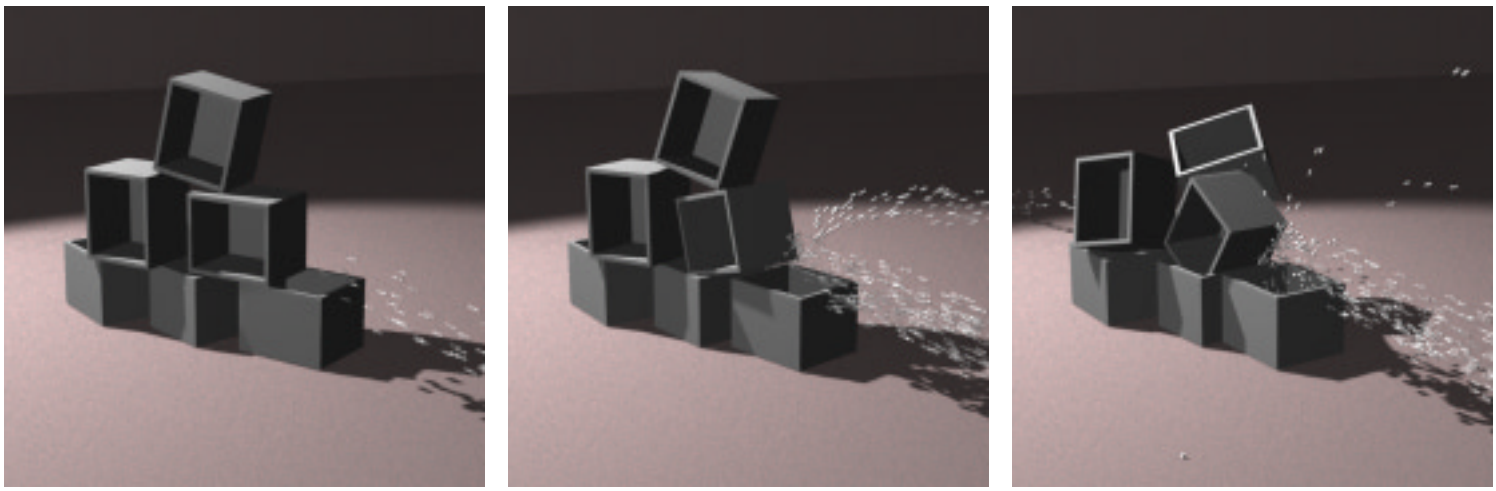

Figure 1: Crates being knocked over by particle stream.

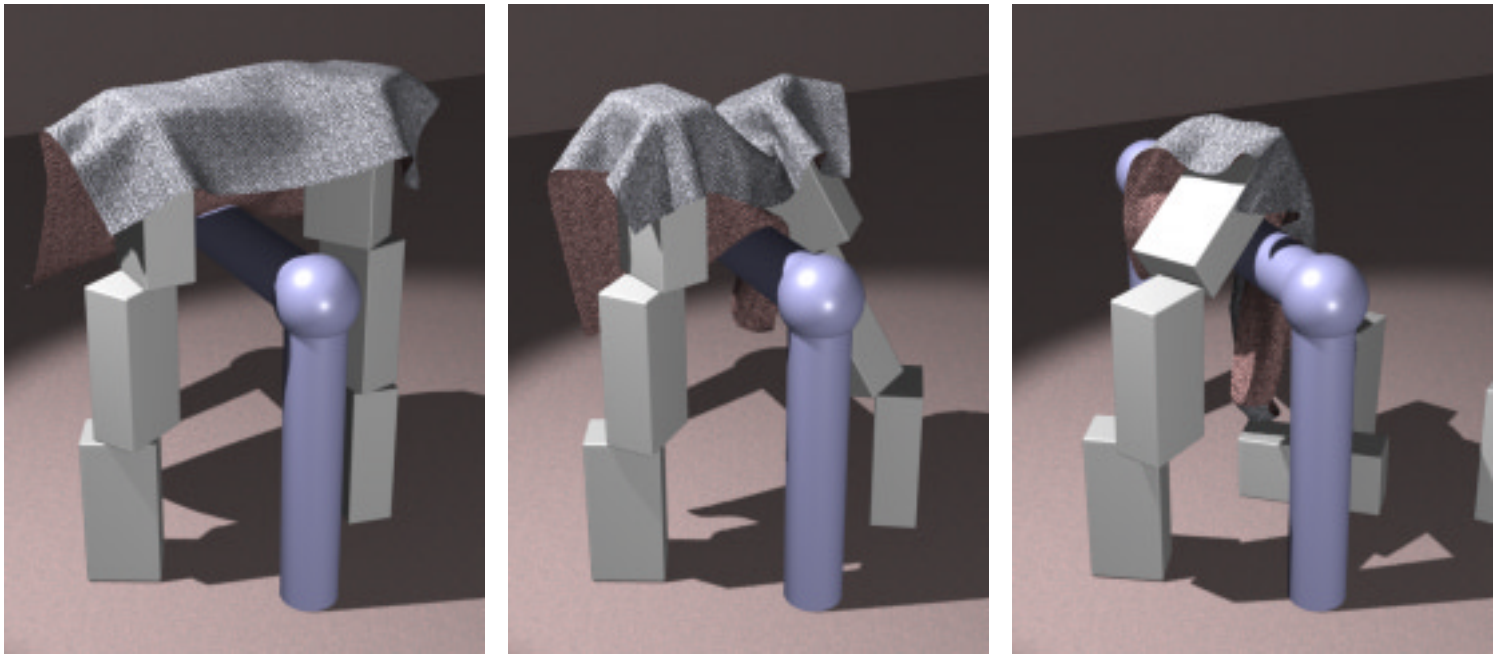

Figure 2: Heavy cloth falling on blocks.

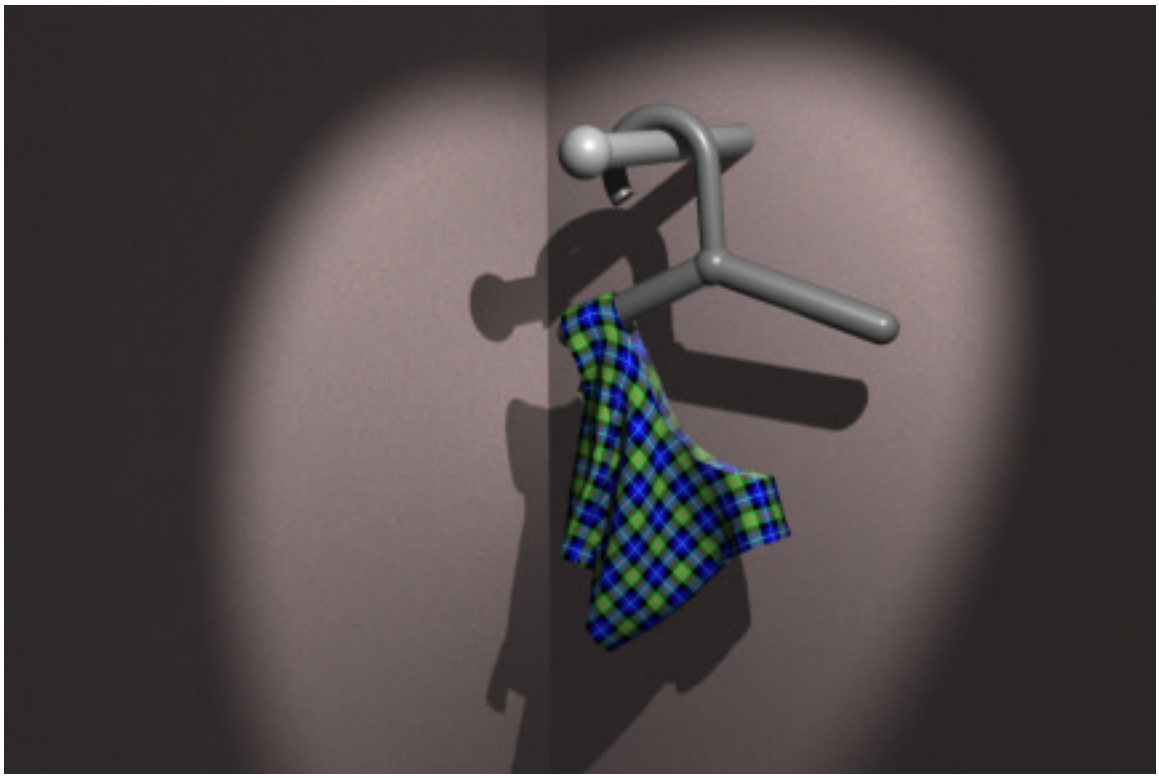

Figure 3: Top falling off hanger.

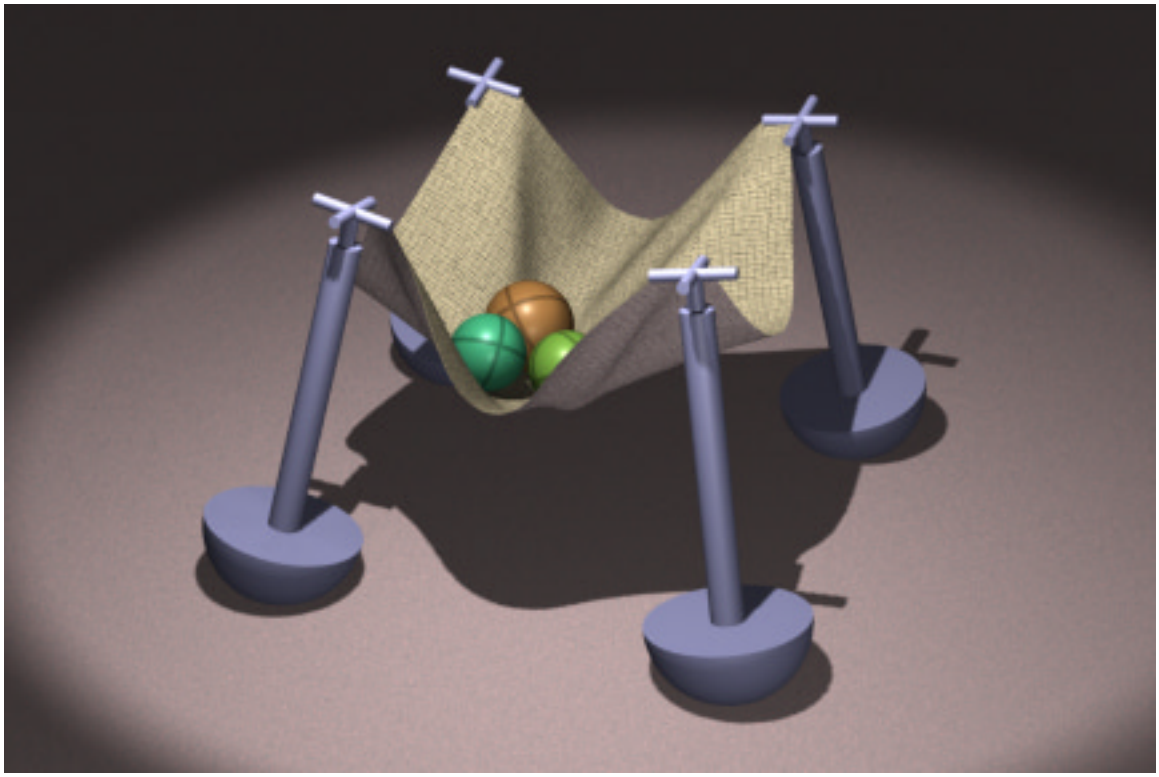

Figure 4: Complex interactions between moveable support posts, a cloth surface, and some spheres.