INTERACTIVE DESIGN OF RIGID-BODY SIMULATIONS FOR COMPUTER ANIMATION

Jovan Popović

July 2001

CMU-CS-01-140

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation (NSF), the Microsoft Corporation, and the Siebel Scholars Program.

Keywords: Computer Animation, Motion Design, Physically Based Animation, Animation with Constraints, Rigid-Body Simulation

Abstract

Physical simulation has become commonplace in computer animation because it produces realistic motion automatically. The animator specifies simulation parameters such as the initial positions and velocities of objects, and the simulator computes the corresponding physical motion. The resulting motion, however, is difficult to design; even a small adjustment of the simulation parameters can drastically change the subsequent motion.

Two semi-automatic techniques are introduced for designing the physical motion of few passive rigid bodies: without any self-propelling forces or linkages and with simple mathematical models of frictionless collision. Interaction is an integral component of the new approaches. The interactive editing technique allows the animator to design the entire physical motion by dragging a body, at any point in time, to a desired position. The sketching technique allows the animator to design the physical motion of bodies by acting out their motion with hand gestures. Both design tools transform a description of how bodies should move into a physical motion that matches the description as closely as possible.

Dedication

To my parents, Branko and Ljiljana, and my brother Zoran

ACKNOWLEDGMENTS

The work described in this dissertation was completed with an abundance of help and guidance from my advisors, friends, and family. Prof. Tony DeRose guided me through my first steps in graduate school at the University of Washington. At Microsoft Research, Dr. Hugues Hoppe helped me with my transition to Carnegie Mellon University. I continue to be inspired by Hugues's detailed approach to research. Dr. Andy Witkin was instrumental in identifying the motion design problem in computer animation. Some of Andy's earlier research with Dr. Will Welch, Dr. Mikako Harada, and Prof. Michael Gleicher served as the main inspiration for techniques described in this dissertation. I also benefited from Andy's guidance during the initial development of interactive design tools. Dr. David Baraff kindly provided the rigid-body simulator. The new techniques are successful because of the efficiency, speed, and robustness of David's physical simulator.

Much of this work would not have been completed without support from my advisors, Prof. Steve Seitz and Prof. Michael Erdmann. I am fortunate to have had the opportunity to be their student. Their technical contributions permeate this dissertation. I am especially grateful to them for sensing my frustrations and lifting my spirits with words of encouragement. Prof. Paul Heckbert read my drafts, attended my talks, and followed my research. His detailed comments and insights improved my research, my writing, and my teaching. During my last year at Carnegie Mellon, I had an additional benefit of discussing my research with Prof. Jessica Hodgins. Her feedback and support helped me stay on top of my work.

I would like to thank the members of the Carnegie Mellon graphics lab for research discussions, debugging, and help with everyday problems. In addition, Sebastian Grassia contributed with his own robust code for parts of the user interface and exponential map derivations. Dennis Cosgrove, from Prof. Randy Pausch's Stage3 lab, created the interface for sketching with motion sensors. Ivan Sokić and Elly Winner created several figures in this dissertation. Elly has also edited and revised most of my writing in the last three years. I will always cherish her companionship and affection.

My family has taken care of me throughout my life. My parents, Branko and Ljiljana, gave me the freedom to be a kid and a grown-up. I am indebted to them for their unconditional love and selfless support. My older brother Zoran has been my playing buddy, my role model, my mentor, and my colleague. He has been my remarkably insightful guide in both work and life.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

Animation is a compelling and effective form of expression; it engages viewers and makes difficult concepts easier to grasp. Creating an animation, however, is a complex process. Traditional techniques require artistry, skill, and careful attention to detail. Ultimately, an animator must design each frame of the animation.

Computer animation improves upon traditional techniques by automating parts of the animation process. For example, keyframing techniques fill-in animation frames from a sparse set of hand-drawn "key-frames." The animator need not create each frame explicitly; instead, she only creates a sufficient subset. Keyframing creates the in-between frames using mathematical interpolation.

Keyframing accelerates the animation process, but the animator must still skillfully create key-frames. In contrast, procedural techniques generate animation automatically; the entire motion is computed by algorithmic methods. Simulation is an appealing procedural technique that solves physics equations to generate realistic motion. The simulated motion depends on simulation parameters, which consist of physical coefficients and other free parameters of physics equations. For example, physical motion of a rigid body is determined by its mass, inertial properties, external forces and other physical coefficients. To create an animation, the animator specifies the simulation parameters and the simulation computes realistic behaviors such as bouncing balls [Moore 88], breaking windows [Terzopoulos 88b, O'Brien 99], folding cloth [Terzopoulos 88a, Baraff 98] and flowing water [Kass 90, Foster 96, Stam 99].

In today's animation industry, simulation methods are essential for creating visually rich and detailed computer animations [Robertson 98, Robertson 99, Robertson 01]. The primary drawback of physical simulation, however, is lack of intuitive control. Although the animator can tune the simulation parameters, in many instances, adjusting the parameters is an ineffective technique for motion design. For example, simulation parameters such as initial position and velocity determine the motion of a die as it bounces across the table.

Just as the slightest difference in a die toss can affect the outcome, a small adjustment of the initial position and velocity can drastically change any simulated motion. Altering these parameters to achieve a desired outcome is tedious, cumbersome, and counterintuitive. An automatic solution to this parameter estimation problem—given a description of the motion, compute the simulation parameters that yield the desired motion—is one of the main challenges in computer animation [Terzopoulos 89].

This dissertation addresses the parameter estimation problem for rigid-body simulations. It describes two novel tools, MOTIONEDITOR and MOTIONSKETCHER, for computing the simulation parameters that yield the desired motion. These methods are not complete solutions to the general parameter estimation problem for rigid-body motion. Instead, they apply to an interesting class of motion-design problems: simulations with a small number of colliding rigid bodies. The bodies are assumed to be passive, without any self-propelling forces. The collisions and contacts are resolved with simple mathematical models of frictionless collision.

Even with simplifying assumptions, the design of rigid-body motion is a tremendously difficult problem. For example, computing the time-optimal motion of rigid bodies in 3-D is NP-hard [Canny 87, Canny 88]. Nonlinearity of motion results in a parameter estimation problem with many local solutions. The parameter estimation problem for rigid-body motion is also not continuous. As a result, efficient optimization techniques have to be adapted to model discontinuities. Sampling approaches could search around discontinuities, but would not scale to the high-dimensional parameter space.

This dissertation introduces a semi-automatic approach to the parameter estimation of designing physical rigid-body motion. Instead of solving the design problem automatically, new methods are introduced to allow the animator to design the motion and, in the process, incorporate her physical intuition about the solution. The methods permit intuitive and rapid design of rigid-body motion. The animator directly controls the bodies at any point in time with assurances that the resulting motion will remain physical. Instead of adjusting the simulation parameters, the animator can use a mouse-based interface to directly manipulate the position and velocity of objects at any point in time. The animator can also design an animation with hand gestures that act out the desired motion of objects. The motion-design methods compute the simulation parameters and the corresponding physical motion that matches the animator's design.

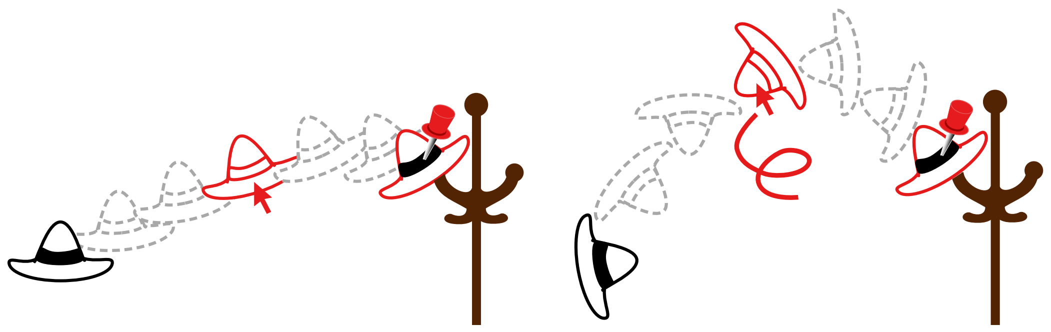


FIGURE 1.1. With MOTIONEDITOR, the animator designs the motion by first fixing the hat's landing position on the coatrack with a "nail" constraint. While the animator spins the hat at an earlier time to achieve the desired spin, the constraint maintains the desired landing location.

MOTIONEDITOR implements an interactive technique for editing physical simulations of rigid-body motion. Throughout the interaction, MOTIONEDITOR displays the entire trajectory of all objects in the scene. The animator is free to manipulate the *entire* motion by correcting the state of the object (position, velocity, etc.) at any time. For example, suppose the animator wants to design a scene in which an actor successfully tosses his hat onto a nearby coatrack, but instead has an animation of the hat falling to the floor. With MOTIONEDITOR, the animator first selects the hat at its landing position and simply drags it onto the coatrack. There are many ways in which the hat can land on the coatrack and the current motion may not have the desired style. As illustrated in Figure 1.1, the animator can adjust the style by first fixing the landing position on the coatrack and then spinning the hat at an earlier time until the hat motion achieves the desired spin.

MOTIONEDITOR expects the simulation to complete before each step of the interaction. When physical simulation is computationally expensive, interaction is not possible. MOTIONSKETCHER is an off-line design tool for motions that cannot be designed interactively. The animator sketches the rough motion of objects with a mouse or 3-D input device, and MOTIONSKETCHER automatically refines the sketch conforming it to physical laws. For instance, an animation of a hat tumbling in the air and landing on the coatrack could be sketched by simply picking up a real hat with an attached motion sensor and moving it in the desired fashion. Based on this sketch, MOTIONSKETCHER automatically constructs a similar hat movement but with physically correct timing and motion (Figure 1.2).

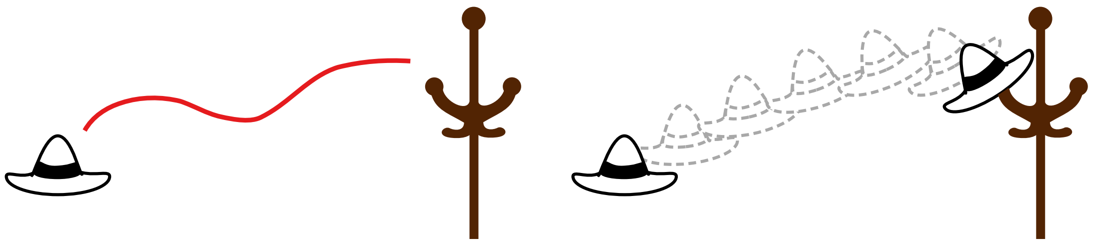


FIGURE 1.2. MOTIONSKETCHER converts the animator's sketch of the desired motion into a similar animation, but with physically correct timing and motion.

Many motion-design tasks are too complex to be solved with the MOTIONEDITOR and MOTIONSKETCHER tools. The success of both paradigms relies on the animator's intuition about the physical world, yet scenarios with complex dynamics can inhibit any intuition about the resulting behavior. For example, the animator may be hard pressed to chart out a physically meaningful sequence of collisions that will lead *all* billiard balls into pockets. In general, this billiard problem combines difficulties that this dissertation does not address: no correct physical intuition, many colliding bodies, frictional collisions and contacts with slip/slide frictional transitions. Without the physically meaningful guidance, MOTIONEDITOR and MOTIONSKETCHER will not converge to a physically realistic motion that matches the intended design.

Nevertheless, as the many examples in this dissertation attest, the animator's intuition can be a valuable guide for the design task. MOTIONEDITOR and MOTIONSKETCHER solve difficult problems by employing the animator's intuition, expressed through interaction, to formulate simpler, more tractable problems. These methods are described in the remaining chapters. Chapter 2 reviews the simulation framework and control methods for rigid-body simulation. Chapter 3 and Chapter 4 describe the MOTIONEDITOR and MOTIONSKETCHER tools in detail. Chapter 5 concludes and discusses future work.

Chapter 2

BACKGROUND

MOTIONEDITOR and MOTIONSKETCHER formulate motion design as a problem of controlling rigid-body simulations. Simulation creates a motion, and the control method ensures the motion reflects the animator's intent. Robotics, mechanical engineering and applied mathematics are few of many fields that study simulation and control of mechanical systems extensively. This chapter summarizes related work, and discusses theoretical and practical underpinnings of simulation and control methods.

2.1 Rigid-Body Simulation

Rigid-body simulation has become commonplace in computer graphics because it produces highly realistic animations. The animator specifies simulation parameters such as the objects' initial positions and velocities, and the simulator automatically generates realistic motions. The simulation integrates equations of motion, detects collisions, and applies impulses to prevent interpenetration. Mathematical models for rigid-body motion are described extensively in many books on classical dynamics [Arnold 89, Symon 71]. The details of rigid-body simulation for computer animation are described in a book [Barzel 92] and two dissertations [Baraff 92, Mirtich 96].

Section 2.1.1 gives an overview of the simulation framework and defines the simulation function that maps simulation parameters into rigid-body motion. Simulation parameters and their effect on the resulting motion are discussed in Section 2.1.2. The following sections describe the details of rigid-body simulation: Section 2.1.3 reviews the equations of motion for rigid bodies, Section 2.1.4 describes the simple collision model that prevents interpenetration, and Section 2.1.5 illustrates main concepts in this section with a simple particle example.

2.1.1 Simulation Function

In the Lagrangian approach, mechanical systems are described in terms of their generalized coordinates and velocities. A system of one or more rigid bodies is described by a *generalized state* vector $\mathbf{q}$ whose components are the generalized coordinates and velocities of the bodies in the system. For example, the generalized state of a single rigid body consists of body's position $\mathbf{x}(t) \in \mathbf{R}^3$, orientation $\mathbf{r}(t) \in \mathrm{SO}(3)$, linear $\mathbf{v}(t) \in \mathbf{R}^3$ and angular $\boldsymbol{\omega}(t) \in \mathbf{R}^3$ velocity:

$$\mathbf{q}(t) = \begin{pmatrix} \mathbf{x}(t) \\ \mathbf{r}(t) \\ \mathbf{v}(t) \\ \boldsymbol{\omega}(t) \end{pmatrix}.$$

The rigid-body simulator computes the *simulation function* $\mathcal{S}$, which specifies the state of the bodies in the world at every point in time:

$$\mathbf{q}(t) = \mathcal{S}(t, \mathbf{u}). \tag{2.1}$$

The control vector $\mathbf{u}$ consists of the simulation parameters such as the parameters describing the initial positions and velocities of simulated bodies. The simulation function maps the control vector $\mathbf{u}$ into the motion $\mathbf{q}(t)$ of rigid bodies. For notational convenience, the simulation function $\mathcal{S}_{t_f}(\mathbf{u})$ also denotes a function mapping the control vector $\mathbf{u}$ into the generalized state at a specific time t_f:

$$\mathcal{S}_{t_f}(\mathbf{u}) = \mathcal{S}(t_f, \mathbf{u}). \tag{2.2}$$

In principle, the animator could manipulate the motion $\mathbf{q}(t)$ by adjusting the control vector $\mathbf{u}$. However, such a form of control would be tedious because the relation between $\mathbf{u}$ and $\mathbf{q}(t)$ can be complex and nonintuitive. Instead, the animator should be able to specify desired body states $\mathbf{q}(t_i)$ at specific times $t_i = t_0, \ldots, t_n$, and automatically compute the control vector $\mathbf{u}$ that produces the desired motion. This problem is difficult for three reasons. First, the domain of the simulation function $\mathcal{S}$ is high-dimensional: for a *single* 3-D body, the components of the generalized state $\mathbf{q}$ are the body's position, orientation, linear, and angular velocity (i.e. $\mathbf{q} \in \mathbf{R}^3 \times \mathrm{SO}(3) \times \mathbf{R}^3 \times \mathbf{R}^3$). Second, the simulation function is highly nonlinear because the simulation is a solution to nonlinear equations of motion (Section 2.1.3) and applied collision impulses (Section 2.1.4). Third, the simulation function is not continuous. Each collision event (e.g., different vertices of an object colliding with the ground) bifurcates the simulation function.

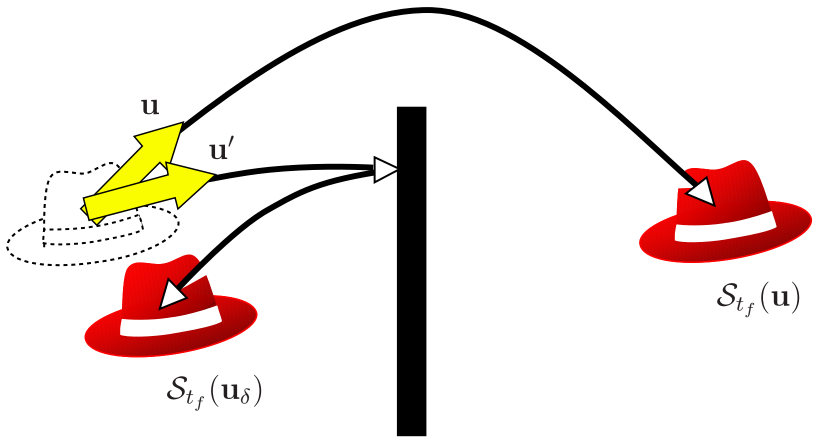

$\mathbf{u}$
$\mathbf{u}'$
$\mathcal{S}_{t_f}(\mathbf{u})$
$\mathcal{S}_{t_f}(\mathbf{u}_\delta)$

FIGURE 2.1. Simulation function discontinuity

Figure 2.1 illustrates a discontinuity in the simulation function. Suppose that the control vector represents the initial velocity of the hat. For some initial velocity $\mathbf{u}$, the hat flies over the fence. In this case, the hat is behind the fence at some final time t_f. For an infinitesimal adjustment and a new initial velocity $\mathbf{u}'$, the hat may collide with the fence. At the same final time t_f, the hat is now in front of the fence.

A connected set of control vectors for which the simulation function is continuous defines a connected component in the control space. A *smooth component* is defined to be a connected component of the control space on which the simulation function is continuously differentiable. For the example in Figure 2.1, control vectors $\mathbf{u}$ and $\mathbf{u}'$ are in different smooth components of the control space.

2.1.2 Simulation Parameters and Motion Realism

Passive rigid bodies, defined as bodies without any self-propelling forces, cannot regulate their own motion; the motion is fully determined by simulation parameters. The initial position and velocity are the only controllable simulation parameters of a passive real-world object. In a simulated environment, however, the animator may also choose to vary other parameters, such as gravitational forces, body shapes, masses, or elasticity coefficients.

The new control parameters add to the available degrees of freedom and can improve the controllability of motion.

The design requirements are sometimes in disagreement with the laws of physics. In computer animation for entertainment, small physical inaccuracies are acceptable as long as the perceived realism is unaffected. Because a viewer does not have complete information about the simulation environment, the animator may introduce additional, physically inaccurate, simulation parameters and increase the degrees of freedom. For example, even a seemingly flat real-life surface has small surface variations. A simulation that perturbs surface normals at each collision may still produce realistic motion [Barzel 96]. Physically inaccurate motions that remain realistic are called *physically plausible*.

The tradeoff between physically plausible and physically accurate motions is not well understood, but some qualitative results exist [Barzel 96, Chenney 00]. The design techniques developed in this dissertation let the animator decide whether a motion is physically plausible. If physical accuracy inhibits the design, the animator can add more simulation parameters. If physical plausibility is compromised, the animator must relax some design requirements.

2.1.3 Equations of Motion

The motion of rigid bodies in free-flight is described by a set of ordinary second order differential equations[1], which we write in vector form as a coupled first order differential equation,

$$\frac{d}{dt}\mathbf{q}(t) = \mathbf{f}(t, \mathbf{q}(t)), \tag{2.3}$$

where the generalized force $\mathbf{f}(t, \mathbf{q}(t))$ is derived from Newton's law. For a single 3-D rigid body with mass m and inertia tensor $\mathbf{I}(t)$, the equations of motion are

$$\frac{d}{dt}\begin{pmatrix}\mathbf{x}(t)\\ \mathbf{r}(t)\\ \mathbf{v}(t)\\ \boldsymbol{\omega}(t)\end{pmatrix} = \begin{pmatrix}\mathbf{v}(t)\\ \frac{1}{2}\boldsymbol{\omega}(t) * \mathbf{r}(t)\\ m^{-1}\mathbf{f}_e(t)\\ \mathbf{I}(t)^{-1}\big(\boldsymbol{\tau}(t) - \boldsymbol{\omega}\times\mathbf{I}(t)\boldsymbol{\omega}\big)\end{pmatrix}. \tag{2.4}$$

[1]Detailed descriptions and derivations can be found in many books on classical dynamics [Arnold 89, Symon 71].

In this example, the generalized state consists of the body's position $\mathbf{x}(t) \in \mathbf{R}^3$, orientation $\mathbf{r}(t) \in \mathrm{SO}(3)$, linear velocity $\mathbf{v}(t) \in \mathbf{R}^3$, and angular velocity $\boldsymbol{\omega}(t) \in \mathbf{R}^3$ describing the body's position, orientation, linear and angular velocity. The generalized force vector includes the external force $\mathbf{f}_e(t) \in \mathbf{R}^3$ and torque $\boldsymbol{\tau}(t) \in \mathbf{R}^3$. Note that the second component of the generalized force, the expression $\boldsymbol{\omega}(t) * \mathbf{r}(t)$, is shorthand notation for a regular quaternion product between the quaternion $[0, \boldsymbol{\omega}(t)]$ and the orientation quaternion $\mathbf{r}(t) = [r_s(t), \mathbf{r}_v(t)]$:[2]

$$\boldsymbol{\omega}(t) * \mathbf{r}(t) = [-\boldsymbol{\omega}(t) \cdot \mathbf{r}_v(t),\ r_s(t)\boldsymbol{\omega}(t) + \boldsymbol{\omega}(t) \times \mathbf{r}_v(t)]$$

Simulation parameters control the simulated motion. If the external force $\mathbf{f}_e(t)$ depends on a parameter in the control vector $\mathbf{u}$, we extend the differential equation appropriately:

$$\frac{d}{dt}\mathbf{q}(t) = \mathbf{f}(t, \mathbf{q}(t), \mathbf{u}). \tag{2.5}$$

This equation of motion completely describes the system in free flight (i.e. when there are no collisions): integrating Equation 2.5 yields the motion $\mathbf{q}(t)$

$$\mathbf{q}(t) = \mathbf{q}_0 + \int_{t_0}^{t} \mathbf{f}(t, \mathbf{q}(t), \mathbf{u})\, dt, \tag{2.6}$$

where $\mathbf{q}_0$ is the generalized state representing the initial state, positions and velocities. The initial state $\mathbf{q}_0$ and other controllable simulation parameters constitute the control vector $\mathbf{u}$:

$$\mathbf{u} = \begin{pmatrix} \mathbf{q}_0 \\ \vdots \end{pmatrix}.$$

2.1.4 Collisions

For computer animation the simple Poisson model of collisions produces acceptable motions [Moore 88]. This collision model can represent elastic and inelastic impacts by applying instantaneous impulses to the colliding bodies. The system simulates the motion during free flight by numerically integrating Equation 2.5. At collision times, impulses are applied to colliding bodies. Impulses prevent interpenetration between the bodies. In a frictionless collision between two bodies, the Poisson collision model describes the relationship between their velocities before and after the collision.

[2]Unit quaternion parametrization of body orientation is a well-established technique [Murray 94].

Suppose that a vertex of body A collides with a facet of body B. At the collision, the simulator applies the impulse $\mathbf{j}$ to instantaneously change the pre-collision generalized state $\mathbf{q}^-$ into the new post-collision generalized state $\mathbf{q}^+$. Recall that the generalized state encodes the positions and velocities of both bodies. The Poisson collision model expresses the relationship between the two states through velocities of colliding points:

$$\mathbf{n}_B \cdot \big(\mathbf{v}_A(\mathbf{q}^+) - \mathbf{v}_B(\mathbf{q}^+)\big) = -\epsilon\Big(\mathbf{n}_B \cdot \big(\mathbf{v}_A(\mathbf{q}^-) - \mathbf{v}_B(\mathbf{q}^-)\big)\Big), \tag{2.7}$$

where the vector $\mathbf{n} \in \mathbf{R}^3$ is the facet normal at the collision and the scalar ϵ is the elasticity coefficient. In this expression, the velocity of the colliding vertex is a function $\mathbf{v}_A(\cdot)$ of a generalized state. Similarly, the function $\mathbf{v}_B(\cdot)$ expresses the velocity of a facet point that collides with the vertex. The simulator computes an impulse $\mathbf{j}$ to generate states $\mathbf{q}^+$ and $\mathbf{q}^-$ that satisfy Poisson Equation 2.7. Any generalized state an instant before the collision $\mathbf{q}^-$ is mapped into the state an instant after the collision $\mathbf{q}^+$:

$$\mathbf{q}^+ = \mathbf{q}^- + \mathbf{j}(\mathbf{q}^-, \mathbf{u}). \tag{2.8}$$

Because the control vector $\mathbf{u}$ comprises simulation parameters such as collision normals and elasticity coefficients, the impulse $\mathbf{j}$ explicitly depends on the control vector $\mathbf{u}$.

2.1.5 2-D Particle Example

A simple particle example is useful to illustrate the concepts described in this section. Suppose that a single 2-D particle moves under the action of gravity. The generalized state $\mathbf{q} \in \mathbf{R}^4$ encodes the particle's position $\mathbf{x} \in \mathbf{R}^2$ and velocity $\mathbf{v} \in \mathbf{R}^2$. If g is the acceleration of gravity, the equations of motion,

$$\frac{d}{dt}\begin{pmatrix}\mathbf{x}(t)\\ \mathbf{v}(t)\end{pmatrix} = \begin{pmatrix}\mathbf{v}(t)\\ \begin{pmatrix}0\\ -g\end{pmatrix}\end{pmatrix}, \tag{2.9}$$

describe the particle's path in free flight. The solution to this differential equation yields the simulation function:

$$\mathcal{S}(t, \mathbf{q}_0) = \begin{pmatrix}\mathbf{x}(t)\\ \mathbf{v}(t)\end{pmatrix} = \begin{pmatrix}\mathbf{x}(0) + \mathbf{v}(0)t + \begin{pmatrix}0\\ -\frac{1}{2}gt^2\end{pmatrix}\\ \mathbf{v}(0) + \begin{pmatrix}0\\ -gt\end{pmatrix}\end{pmatrix}. \tag{2.10}$$

If the particle collides with an immovable obstacle, the Poisson collision model applies an impulse to change the particle's velocity. For frictionless collisions, the impulse acts in the direction of the surface normal $\mathbf{n}$ at the point of collision. Assuming a perfectly elastic bounce ($\epsilon = 1$), the equation,

$$\mathbf{v}^+ = \mathbf{v}^- - 2(\mathbf{n} \cdot \mathbf{v}^-)\mathbf{n}, \tag{2.11}$$

applies an impulse to instantaneously change the particle's velocity $\mathbf{v}^-$ before the collision into its velocity $\mathbf{v}^+$ after the collision.

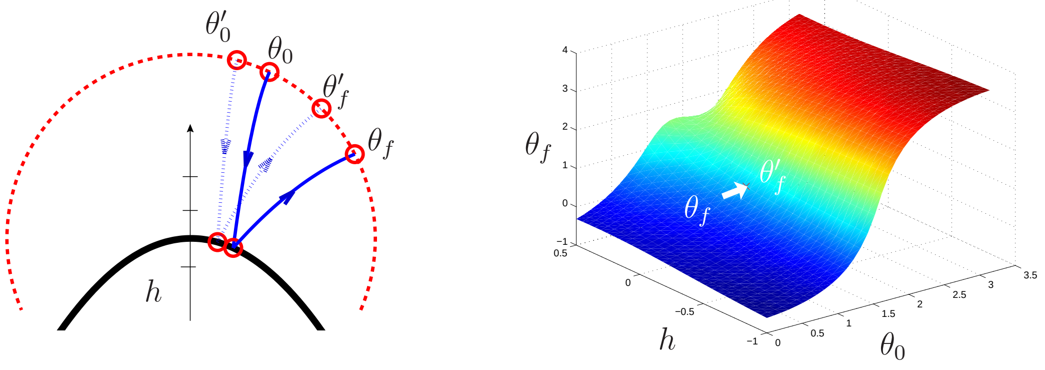


FIGURE 2.2. The simulation function for a particle bounce against a smooth obstacle.

Given these analytical expressions for the particle's motion, we can plot the space of all possible trajectories for the particle as a function of the initial conditions and the environment. For concreteness, suppose the particle collides with a single parabolic obstacle. The dimensions of the range and the domain of the simulation function can be reduced by introducing a unit circle in the scene: the particle enters the circle at some angle θ_0 with unit velocity vector directed towards a point on the vertical axis h, above or below the tip of the obstacle. The particle bounces off the obstacle and exits the circle at another angle θ_f. The simulation function $\mathcal{S}_f(\cdot) : \mathbf{R}^2 \to \mathbf{R}$ maps the control vector $\mathbf{u} = (\theta_0, h)$ into the particle's final, exit position θ_f. The simulation function is smooth and continuous in a region of the control space shown in Figure 2.2.

In general, as discussed in Section 2.1.1, the simulation function is not smooth and continuous everywhere. For example, a particle could be propelled to miss the obstacle. Even with a single bounce the simulation function becomes discontinuous when the obstacle is

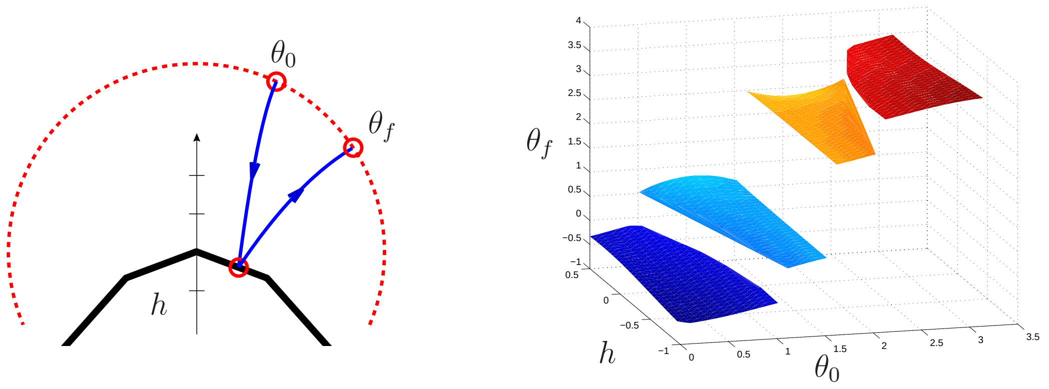


FIGURE 2.3. The simulation function for a particle bounce against non-smooth obstacle.

approximated by polygonal (piecewise linear) meshes. Suppose that the smooth obstacle in the particle example is replaced by a piecewise linear polygonal curve. As long as the particle collides with the same edge, the simulation function remains continuous. When the particle collides with a different edge, however, the surface normal on the obstacle changes abruptly. Thus, the resulting collision impulse (Equation 2.11) is discontinuous. The abrupt change carries over to the subsequent particle motion and corresponds to a discontinuity in the simulation function. In this example the four smooth components, shown in Figure 2.3, correspond to motions of the particle colliding with each of the four edges.

The general motion of many rigid bodies is much like this simple particle example: simulating the motion of multiple rigid body is equivalent to simulating a motion of a single particle in a high dimensional space. To describe the state of a single 3-D rigid body, the dimensions of the generalized state increases to include the components of orientation, angular velocity, and to extend the position and linear velocity vectors to 3-D. Two or more rigid bodies are modeled by adding additional components to the generalized state. Although the equations of motion are more complicated, a careful implementation can enable complex multi-object simulations with the same computational techniques and data structures used to implement single particle simulations.

2.2 Control of Rigid Body Simulation

Simulation computes physical motion automatically, but it is not effective for motion design. In a computer animation, the motion must also attain specific goals: a hat must land on the coatrack and the hat must spin before the landing. To design motion easily, the animator needs tools that control rigid-body simulation.

2.2.1 Boundary-Value Problem

Rigid-body simulation is usually thought of as an initial-value problem: given the initial state of rigid-bodies $\mathbf{q}_0$, the simulator computes the motion $\mathbf{q}(t)$ such that $\mathbf{q}(t_0) = \mathbf{q}_0$ at an initial time t_0. If the animation requires a particular state at an initial time then formulating rigid-body simulation as an initial-value problem is sufficient. In general, however, the simulated motion may also have to achieve specific states at other times $t_1, \ldots, t_n$. For example, in the hat animation from Chapter 1, the hat is tossed from the left of the screen and must land on the coatrack. For these animation problems a boundary-value formulation is more appropriate. Simulation formulated as a boundary-value problem integrates equations of motion, applies impulses to compute the motion $\mathbf{q}(t)$ and also satisfies animation constraints:

$$\begin{pmatrix} c_0\big(t_0, \mathbf{q}(t)\big) \\ \vdots \\ c_n\big(t_n, \mathbf{q}(t)\big) \end{pmatrix} = \mathbf{c}\big(t_0, \ldots, t_n, \mathbf{q}(t)\big) = 0.$$

Boundary-value problems are harder to solve than initial-value problems. In contrast to initial-value problems, for which uniqueness and existence of solutions can be guaranteed under fairly general conditions, boundary-value problems can have many solutions or none at all. Intuitively the additional complexity stems from global dependencies in boundary-value problems; boundary values describe global features, which are defined throughout the entire motion.

Shooting and collocation are the two most common numerical methods for solving non-linear boundary-value problems. Shooting solves boundary-value problems by computing the parameters $\mathbf{u}$ of the simulation function $\mathcal{S}(t, \mathbf{u})$ for which the resulting motion satisfies the boundary values $\mathbf{c}\big(t_0, \ldots, t_n, \mathbf{q}(t)\big)$. Collocation computes a piecewise polynomial that satisfies the boundary values and approximates the simulated motion. A detailed treatment of boundary-value problems and numerical solutions appears in several books [Stoer 80, Ascher 88].

The existing numerical methods for solving boundary-value problems are not effective for the design of rigid-body simulations. Most numerical methods assume smooth and continuous functions. As discussed in Section 2.1.1, these assumptions are not valid for motion of colliding rigid bodies. Furthermore, boundary-value formulations require that animation constraints are correctly formulated. The animation constraints have to be succinct, sufficient, and feasible. Too many constraints would unnecessarily complicate a boundary-value problem. Too few constraints would not describe the intended motion uniquely. Infeasible constraints would result in a boundary-value problem with no solution at all.

2.2.2 Optimal Control Theory

Optimal control theory provides the foundation for maximizing the performance of evolving dynamic systems. A detailed treatment of optimal control theory appears in several books [Pontryagin 62, Stengel 94, Bertsekas 95a]. In motion design, the performance criteria would measure how well the motion fits the animator's intent. For example, the animator may indicate that a flying hat must spin. Optimal control computes the time-varying controller $\mathbf{u}(t)$ that optimizes the cost function, $\int_{t_0}^{t_f} L\big(t, \mathbf{q}(t), \mathbf{u}(t)\big)\, dt$, where the motion $\mathbf{q}(t)$ of the dynamic system is controlled by the controller $\mathbf{u}(t)$. In the literature, an optimal control problem with an integral cost function is called a Lagrange type control problem[3] and the integrand is called the Lagrangian.

Classical, or indirect, approaches to problems of the Lagrange type derive the necessary conditions using the Euler-Lagrange equations, the Minimum Principle, or the Hamilton-Jacobi-Bellman equation. The Euler-Lagrange and the Minimum Principle conditions convert optimal control problems into boundary-value problems: the initial dynamic state and their adjoint variables form boundary values at opposite ends of the time interval. The Hamilton-Jacobi-Bellman equation formulates an optimal control problem as a partial differential equation, which is then solved with dynamic programming. Section 2.2.3 discusses this approach in more detail and provides an overview of dynamic-programming techniques for efficiently solving optimal control problems.

[3]Lagrange type problems can also be transformed into Mayer type or Bolza type control problems. The three forms are equivalent and chosen purely for convenience.

Direct approaches to problems of the Lagrange type discretize the control function $\mathbf{u}(t)$ and the integral cost function to formulate a finite dimensional optimization problem. The optimization can be solved efficiently with one of many techniques for general optimization [Gill 89, Bertsekas 95b]. In computer graphics, spacetime methods are the best example of direct solutions to optimal control problems.

Spacetime methods compute the motion of dynamic systems by solving a constrained optimization problem. The solution to the optimization problem is a motion that meets the animation goals, which are specified as optimization constraints. For example, the original spacetime method [Witkin 88] minimized the power usage of a virtual actor (Luxo lamp) subject to the dynamics and pose constraints. The optimization produces realistic and natural motion.

Recent spacetime methods propose different optimization objectives and different constraint formulations. Similar to the goal of constructing the motion that best matches the sketch, three recent spacetime methods [Gleicher 97, Gleicher 98, Popović 99] transform motion-capture data to construct new, distinctly different, motion that preserves the style of the original. These methods rely on motion-capture data that is both realistic and accurate. The realism in the data permits the methods to ignore the laws of dynamics and still yield convincing realistic motions.

Spacetime methods use the finite difference method [Witkin 88] or the collocation technique [Cohen 92, Liu 94, Gleicher 97, Gleicher 98, Popović 99] to discretize equations of motion. Both techniques result in enormous optimization problems because they require optimization constraints that enforce the equations of motion at each discrete time.

Like boundary-value formulations, all spacetime implementations assume smooth and continuous optimization problems. Although this assumption drastically simplifies optimization problems, it also complicates the design task. The animator must specify the sequence of collisions and their collision times. For example, the animator must specify die orientations and the timing of each bounce.

Optimal control computes the controller $\mathbf{u}(t)$ that is active at all times. In contrast, this dissertation addresses the control of passive rigid bodies with controllers that must act only at specific, discrete times: at initial time or at each collision. Although optimal control theory does not address this problem directly, parameter estimation is a similar formulation that relies on techniques for solving optimal control problems. Section 2.2.6 describes the parameter estimation formulation and related work.

2.2.3 Dynamic Programming

As described in the previous section, dynamic-programming formulation of optimal control solves the Hamilton-Jacobi-Bellman equation to compute the control function $\mathbf{u}(t)$. For continuous optimal control, the Hamilton-Jacobi-Bellman equation is a partial differential equation that is in most cases harder to solve than the Euler-Lagrange formulation. Instead, dynamic programming is best suited for discrete optimal control as done with reinforcement-learning methods developed in the artificial intelligence community [Kaelbling 96].

For most practical problems, when dynamic programming is applied to continuous problems, the state-space $\mathbf{q}(t)$ and the control space $\mathbf{u}(t)$ are discretized. This approach is often practical, but for high-dimensional problems computational requirements can be overwhelming [Bertsekas 95a]. Because the state of each rigid body is a 12-dimensional vector, the design problems in this dissertation are high-dimensional. For example, designing an animation of two passive rigid bodies results in a 24-dimensional control space (12 degrees of freedom for initial position and velocity of each rigid body). Although several approaches improve the scalability of dynamic programming by approximating the cost function [Bertsekas 96] or by employing variable-resolution grids [Munos 99], existing dynamic-programming techniques may still not be computationally efficient for designing rigid-body simulations.

2.2.4 Inverse Dynamics

Inverse dynamics computes the forces that produce the desired motion of a dynamic system. For example, the differential Equation 2.3 describes the relationship between the state of the system $\mathbf{q}(t)$ and the forces $\mathbf{f}(t)$. In contrast to forward dynamics, which computes the state given the forces, inverse dynamics computes the forces $\mathbf{f}(t)$ given the state $\mathbf{q}(t)$.

Inverse dynamics is a simple and efficient framework for some control problems [Isaacs 87, Barzel 88]. If the bodies initially violate a desired constraint, then the inverse dynamics can compute the forces required to guide the bodies into a configuration that complies with the constraint [Barzel 88]. For example, a deck of cards can be automatically assembled into a house of cards because inverse dynamics can compute the forces to assemble the cards and enforce constraints to prevent them from falling apart.

However, inverse dynamics is not an appropriate solution for designing the motion of passive rigid bodies. To meet the constraints, inverse dynamics applies forces of arbitrary direction and strength. As a result each body moves like it has a jet engine. For the motion of passive rigid bodies this is not the desired behavior.

2.2.5 Robot Control

The robotics community explores the methods for controlling rigid bodies to design the motion and behavior of a robot. Path planning methods compute collision-free paths to guide robots around obstacles in real environments. Control algorithms design actuators that exert forces and produce the robot motion. Many of these methods have been successfully applied to the animation of virtual actors.

Path planning methods frequently assume kinematic models of robot motion [Latombe 91]. This assumption is justified when robots can move slowly to reduce the effects of motion dynamics. Kinodynamic planning addresses the more general path planning problem with kinematic and dynamic constraints. Because the general kinodynamic problem is NP-hard [Canny 87, Canny 88], polynomial-time approximation algorithms discretize the state space to reduce the kinodynamic planning into finding a shortest-path in a directed graph [Donald 93]. In practice, probabilistic techniques can explore the state space efficiently to find the path between desired start and goal locations [LaValle 99].

Control algorithms construct convincing motions of active rigid body systems, most notably human and animal locomotion [Raibert 91, Hodgins 95, Grzeszczuk 95]. Active rigid-body systems can approximate human and animal skeletons which rely on their own muscles for locomotion. Control algorithms compute the time-varying actuators that produce the desired motion. Although progress has been made in generating control algorithms automatically [van de Panne 90, van de Panne 93, Grzeszczuk 95] controlling a general active dynamic system to meet arbitrary motion constraints remains an unsolved problem.

2.2.6 Parameter Estimation

The control methods described in previous sections assume active dynamic systems: systems with actuating controls $\mathbf{u}(t)$ that can exert forces at any time. In contrast, the motion of passive bodies is determined instead by a finite number of simulation parameters $\mathbf{u}$. Parameter estimation is a more natural formulation for designing the motion of passive rigid bodies.

The general parameter estimation framework identifies unknown parameters that fit a mathematical model of physical behavior to observed real-world measurements [Stengel 94]. In the context of motion design, parameter estimation computes the simulation parameters $\mathbf{u}$ that fit the resulting motion, $\mathbf{q}(t) = \mathcal{S}(t, \mathbf{u})$, to the animator's desired specification.

The distinction between optimal control and parameter estimation is not clear-cut. For example, direct methods for optimal control parametrize the actuating control $\mathbf{u}(t)$ with polynomial basis functions. If the coefficients of the basis functions are also added to the finite-dimensional control vector $\mathbf{u}$ then solving the parameter estimation problem is equivalent to solving an optimal control problem. This section makes the distinction to emphasize the dimensionality of the problem unknowns: regardless of the solution approach, parameter estimation solves for a finite-dimensional vector of control parameters $\mathbf{u}$ and optimal control solves for control trajectories $\mathbf{u}(t)$.

Several search techniques have been applied to the parameter estimation problem for motion design. A genetic algorithm computed solutions for a simple class of 2-D N-body problems [Tang 95]. A backwards search from desired ball locations in a 2-D billiards simulation discovered successful trick shots that, for example, end with balls in the desired table pockets [Barzel 96]. The Markov chain Monte Carlo method excelled at computing the motions for rigid-body systems with chaotic behavior such as the motion of bowling pins after the impact with a bowling ball [Chenney 00]. All of these approaches have long running times. For example, the Monte Carlo method, the most general of the three methods, could require several hours of computation. Furthermore, the efficiency of the Monte Carlo method depends on a proposal mechanism, which generates new control vectors $\mathbf{u}$ for the simulation [Chenney 00]. If the computed motion is not appropriate or the method fails to find any motion, the animator must re-adjust the animation constraints or the proposal mechanism and start again. In short, these methods do not permit interactive exploration, which is essential for design when aesthetics is a primary concern.

Parameter estimation problems in chemistry, biology and other sciences are formulated as continuous optimization problems [Bock 83, Bock 80]. In these contexts, dynamic processes such as chemical reactions are modeled with nonlinear ordinary differential equations. The optimization fits the parameters of the model to match the observed measurements. Frequently, the fitting process assumes independent, normally distributed measurement errors and optimizes a weighted least-squares function to obtain a maximum-likelihood estimate. Unlike the sampling and searching techniques, these numerical optimization methods compute the derivatives of a dynamic process to solve nonlinear estimation problems robustly and efficiently even in high-dimensional parameter spaces. The techniques in this dissertation extend this derivative-based estimation approach to the design of rigid-body simulations.

Chapter 3

MOTION EDITOR

Interaction is essential for exploring any design space because design goals are often subjective and concerned with aesthetics of the result. An intuitive design tool for rigid-body simulations should allow the animator to design the motion directly without manipulating the underlying simulation parameters. Ideally, the animator should immediately see the results of her adjustments and quickly explore the space of motions that achieve the desired behavior.

Interactive design techniques have been devised for geometric modeling [Witkin 90], drawing applications [Gleicher 91], interactive camera control [Gleicher 92], architectural design [Harada 95], and others. At the core of these approaches is an interactive derivative-based technique. MOTIONEDITOR, the interactive editing technique described in this chapter, extends this approach to the design of rigid-body simulations. The animator can select bodies at any time and simply drag them to desired locations. In response, MOTIONEDITOR computes the required physical parameters and simulates the resulting motion. Surface characteristics such as normals and elasticity coefficients can also be automatically adjusted to provide a greater range of feasible motions, if the animator so desires. Because the entire simulation editing process runs at interactive speeds, the animator can rapidly design complex physical animations that would be difficult to create with existing rigid body simulators.

Internally, MOTIONEDITOR represents the entire motion of bodies by the simulation parameters that control the simulation (i.e., initial positions and velocities, surface normal variations and other parameters included by the animator). As the animator interactively manipulates the motion, MOTIONEDITOR computes the new physical parameters that achieve the desired motion update. This is achieved in real time using a fast differential update procedure in concert with a rigid body simulator. Motion discontinuities pose an additional challenge (e.g. when a point of collision changes to a different facet on a body's polyhedral mesh) because the motion changes abruptly. When this happens, MOTIONED-

ITOR performs a local discrete search in physical parameter space to compute the motion that most closely complies with the desired adjustments. This sampling approach is inspired by the continuous and discrete optimization method for applications in architectural design [Harada 95].

The organization of this chapter follows the description of this work published earlier in conference proceedings [Popović 00]. This chapter includes additional details and clarifications. Section 3.1 presents the top-level description of the MOTIONEDITOR algorithm. Section 3.2 derives the equations for computing the derivatives of the motion efficiently. The details of the differential update during the editing interaction are described in Section 3.3. Section 3.4 describes MOTIONEDITOR constraint formulations. Section 3.5 describes a discrete search approach to resolving discontinuities in the simulation function. Section 3.6 evaluates the editing approach with several motion-design tasks.

3.1 Basic Algorithm

MOTIONEDITOR adapts a differential approach for manipulating rigid-body simulations. The animator adjusts the motion by specifying a desired motion modification, which is expressed as a differential change $\delta \mathbf{q}_i$ in the generalized state $\mathbf{q}(t_i)$ at time t_i. As shown in Figure 3.1, MOTIONEDITOR computes the required differential change $\delta \mathbf{u}$ in the control vector $\mathbf{u}$ that reshapes the current motion to comply with the differential adjustments $\delta \mathbf{q}_i$. Recalling the notation from Section 2.1, the simulation function $\mathcal{S}_{t_i}$ maps the control vector

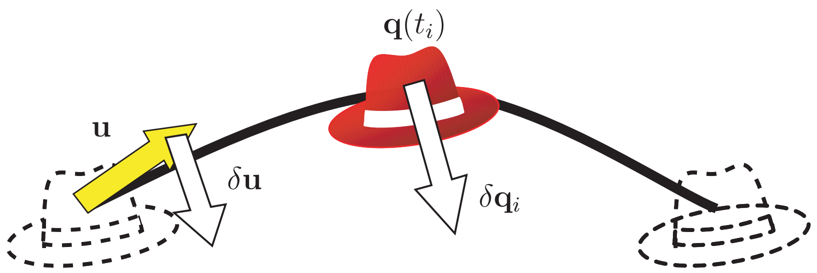


FIGURE 3.1. MOTIONEDITOR computes a differential adjustment $\delta \mathbf{u}$ in the control vector $\mathbf{u}$ that complies with a desired differential adjustment $\delta \mathbf{q}_i$ in the generalized state $\mathbf{q}(t_i)$.

$\mathbf{u}$ into the generalized state $\mathbf{q}(t_i)$, as described in Equation 2.2. Locally linearizing the simulation function yields a linear relationship between the desired motion adjustment $\delta \mathbf{q}_i$

and the required control vector adjustment $\delta\mathbf{u}$:

$$\delta\mathbf{q}_i = \frac{\partial\mathcal{S}_{t_i}(\mathbf{u})}{\partial\mathbf{u}}\delta\mathbf{u}. \tag{3.1}$$

The Jacobian matrix $\partial\mathcal{S}_{t_i}(\mathbf{u})/\partial\mathbf{u}$ represents the linear transformation and encodes an editing constraint. MOTIONEDITOR combines all editing constraints into linear optimization constraints. As described in Section 3.3, the optimization may compute the differential control vector $\delta\mathbf{u}$ with either a conjugate gradient technique or singular value decomposition.

The differential vector $\delta\mathbf{u}$ describes the direction in which to change the current control vector $\mathbf{u}$ to obtain the desired motion change $\delta\mathbf{q}_i$. The differential update is a small step in the computed direction,

$$\mathbf{u}' = \mathbf{u} + \epsilon\,\delta\mathbf{u}, \tag{3.2}$$

where ϵ is the length of the step. Because Equation 3.1 is only a linear approximation of the nonlinear simulation function, the new control vector $\mathbf{u}'$ will not acquire desired motion change in a single step. Instead, given the new, updated control vector $\mathbf{u}'$, a rigid body simulator computes the new motion, displays the result, and repeats the differential update. By repeating this interactive method, the animator guides MOTIONEDITOR toward the desired solution.

Figure 3.2 describes the interactive editing algorithm. MOTIONEDITOR consists of three parts: (1) a differential-control module, (2) a rigid-body simulator, and (3) a user interface for motion display and editing. In response to an editing operation from the user interface, the control module recomputes the simulation parameters $\mathbf{u}$ needed to accomplish the desired motion adjustments. The new simulation parameters are supplied to the rigid-body simulator, which recomputes the motion and updates the display.

The implementation relies on the general-purpose rigid-body simulator developed by David Baraff [Baraff 92, Baraff 94]. MOTIONEDITOR interacts with the simulator to provide the control vector $\mathbf{u}$ for the simulation and to modify the collision impulses with adjusted surface normals and elasticity coefficients. The simulator, in turn, computes the new motion. The user interface displays the new motion and reevaluates editing constraints. The differential-control module processes the new motion to extract collision events and

```
/* User Interface */
animator specifies desired state $\mathbf{q}_i$ at time $t_i$
repeat
    $\delta\mathbf{q}_i = \mathbf{q}_i - \mathbf{q}(t_i)$
    /* Differential-Control Module */
    compute the Jacobian matrix $\frac{\partial \mathcal{S}_{t_i}(\mathbf{u})}{\partial \mathbf{u}}$
    solve $\delta\mathbf{q}_i = \frac{\partial \mathcal{S}_{t_i}(\mathbf{u})}{\partial \mathbf{u}} \delta\mathbf{u}$ for $\delta\mathbf{u}$
    $\mathbf{u} = \mathbf{u} + \epsilon\, \delta\mathbf{u}$
    /* Rigid-Body Simulator */
    update motion $\mathbf{q}(t)$ by computing the simulation function $\mathcal{S}(t, \mathbf{u})$
    redisplay $\mathbf{q}(t)$
until $\delta\mathbf{q}_i = 0$
```

FIGURE 3.2. The interactive editing algorithm.

to decompose the equations of motion as described in Section 3.2. A change in the control parameters may introduce another bounce, which the simulator detects as an additional collision. In response, the differential-control module updates the equations of motion to include the collision.

The user interface displays the entire simulation by tracing out the trajectories of one or two points on the moving objects. This display minimizes clutter, yet it provides the animator with a sense of the cumulative motion that is sufficient for most interaction tasks. The animator can choose to view the complete motion as a traditional frame sequence at any time during the interaction.

3.2 Jacobian Computation

MOTIONEDITOR relies on the efficient computation of the Jacobian matrix $\partial \mathcal{S}_{t_i}(\mathbf{u})/\partial \mathbf{u}$ in Equation 3.1. Computing the Jacobian matrix with finite differences is expensive because of the need to perform at least one additional simulation for each simulation parameter. For example, a forward difference formula,

$$\frac{d\mathcal{S}_{t_i}(u)}{du} \approx \frac{\mathcal{S}_{t_i}(u+h) - \mathcal{S}_{t_i}(u)}{h},$$

approximates the derivative of the simulation function $\mathcal{S}_{t_i}(u)$ with respect to the scalar simulation parameter u. This formula requires two simulations: one to evaluate the simulation for the current simulation parameter u and one to evaluate the simulation for the perturbed parameter $u + h$. Similarly, each additional parameter would require one more simulation to compute its directional derivative. Furthermore, the accuracy of the forward difference formula is only $O(h)$ with the additional caveat that the integration accuracy for computing the simulation function $\mathcal{S}_{t_i}$ must be $O(h^2)$ to achieve this derivative accuracy [Bock 83]. In short, finite-difference approach is inefficient and inaccurate for the editing approach.

Instead of numerically approximating the derivatives, MOTIONEDITOR uses a specialized automatic differentiation technique [Griewank 91]. The simulation function $\mathcal{S}$ is decomposed into analytically differentiable functions and the Jacobian is numerically composed from the subcomponents using the chain rule. For example, suppose that a single collision occurs at time t_c and the simulation function $\mathcal{S}_{t_f}(\mathbf{u})$ describes the body's state at some time $t_f > t_c$ after the collision. The function $\mathcal{S}_{t_f}(\mathbf{u})$ is the composition of three sub-functions:[1]

$$\mathcal{S}_{t_f}(\mathbf{u}) = \mathcal{F}_{t_f}(\mathbf{u}, \cdot) \circ \mathcal{C}_{t_c}(\mathbf{u}, \cdot) \circ \mathcal{F}_{t_c}(\mathbf{u}, \mathbf{q}_0), \tag{3.3}$$

$\mathbf{q}(t_c^-) = \mathcal{F}_{t_c}\big(\mathbf{u}, \mathbf{q}_0\big)$: pre-collision free-flight function, which maps the initial conditions $\mathbf{q}_0$ and perhaps additional elements of the control vector $\mathbf{u}$ into the body's state at t_c^-, an instant before collision (e.g. , Equation 2.10 for 2-D particles);

$\mathbf{q}(t_c^+) = \mathcal{C}_{t_c}\big(\mathbf{u}, \mathbf{q}(t_c^-)\big)$: collision function, which applies the impulse and maps the body's state an instant before collision into the body's state at t_c, an instant after collision (e.g. , Equation 2.11 for 2-D particles);

$\mathbf{q}(t_f) = \mathcal{F}_{t_f}\big(\mathbf{u}, \mathbf{q}(t_c^+)\big)$: post-collision free-flight function, which maps the body's state an instant after the collision into the body's state at t_f.

MOTIONEDITOR computes the Jacobian of the simulation function by computing Jacobians of the sub-functions and evaluating the chain rule:

$$\frac{\partial \mathcal{S}_{t_f}(\mathbf{u})}{\partial \mathbf{u}} = \frac{\partial \mathcal{F}_{t_f}}{\partial \mathcal{C}_{t_c}} \left(\frac{\partial \mathcal{C}_{t_c}}{\partial \mathcal{F}_{t_c}} \frac{\partial \mathcal{F}_{t_c}}{\partial \mathbf{u}} + \frac{\partial \mathcal{C}_{t_c}}{\partial \mathbf{u}} \right) + \frac{\partial \mathcal{F}_{t_f}}{\partial \mathbf{u}}$$

[1]The Equation 3.3 is written in this form for notational convenience. Alternatively, this equation is written as $\mathcal{S}_{t_f}(\mathbf{u}) = \mathcal{F}_{t_f}(\mathbf{u}, \mathcal{C}_{t_c}(\mathbf{u}, \mathcal{F}_{t_c}(\mathbf{u}, \mathbf{q}_0)))$.

This derivation describes the Jacobian computation for a motion with a single collision. Motions with an arbitrary number of collisions are composed in an analogous manner by chaining together free-flight and collision functions for each segment. Section 3.2.1 derives the Jacobian for free-flight functions and Section 3.2.2 derives the Jacobian for collision functions.

3.2.1 Jacobian of Free-Flight Functions

Although the free-flight motion of the particle in Section 2.1.5 has a closed form and is analytically differentiable, motion of a rigid body in 3-D does not have an analytic solution in the general case.[2] The simulator must numerically integrate equations of motion (Equation 2.5) to compute the free-flight motion.

We begin deriving the Jacobian $\partial \mathcal{F}_{t_c}(\mathbf{u}, \mathbf{q}_0)/\partial \mathbf{u}$, by integrating Equation 2.3 which encapsulates the equations of motion,

$$\mathcal{F}_{t_c}(\mathbf{u}, \mathbf{q}_0) = \mathbf{q}_0 + \int_{t_0}^{t_c(\mathbf{u})} \mathbf{f}(t, \mathbf{q}, \mathbf{u})\, dt,$$

where the generalized force $\mathbf{f}(t, \mathbf{q}, \mathbf{u})$ is the right-hand side of the differential Equation 2.5 with initial condition $\mathbf{q}_0$. Differentiating both sides with respect to control vector $\mathbf{u}$ yields a new integral equation:

$$\frac{\partial \mathcal{F}_{t_c}(\mathbf{u}, \mathbf{q}_0)}{\partial \mathbf{u}} = \frac{\partial}{\partial \mathbf{u}} \left(\mathbf{q}_0 + \int_{t_0}^{t_c(\mathbf{u})} \mathbf{f}(t, \mathbf{q}, \mathbf{u})\, dt \right).$$

Note that the time of collision depends on the control vector $\mathbf{u}$. We evaluate this integral equation by applying the Leibnitz rule [Kaplan 84], which interchanges the integration and differentiation:[3]

$$\frac{\partial \mathcal{F}_{t_c}(\mathbf{u}, \mathbf{q}_0)}{\partial \mathbf{u}} = \mathbf{f}(t_c(\mathbf{u}), \mathbf{q}, \mathbf{u}) \frac{dt_c(\mathbf{u})}{d\mathbf{u}} + \frac{\partial \mathbf{q}_0}{\partial \mathbf{u}} + \int_{t_0}^{t_c(\mathbf{u})} \frac{\partial \mathbf{f}(t, \mathbf{q}, \mathbf{u})}{\partial \mathbf{u}}\, dt. \tag{3.4}$$

The term $\mathbf{f}(t_c(\mathbf{u}), \mathbf{q}, \mathbf{u})$ is the right-hand side of the equation of motion (Equation 2.5) evaluated at the collision time. The simulator computes the value of this term from external

[2]For the special case of freely rotating 3-D rigid body (no torques), there is an analytic Poinsot's solution [Symon 71].

[3]The conditions for applying the Leibnitz rule require that $\mathbf{f}$ is continuous and has a continuous derivative $\partial \mathbf{f}/\partial \mathbf{u}$. These conditions are met under reasonable assumptions about external forces.

forces, torques, and the generalized state of the colliding bodies. The collision time derivative $dt_c(\mathbf{u})/d\mathbf{u}$ depends on the type of collision; we will derive this term in Section 3.2.2. The second and third term, the expression $\frac{\partial \mathbf{q}_0}{\partial \mathbf{u}} + \int_{t_0}^{t_c(\mathbf{u})} \frac{\partial \mathbf{f}(t,\mathbf{q},\mathbf{u})}{\partial \mathbf{u}}\, dt$, define an integral expression, which MOTIONEDITOR computes by numerically integrating the differential equation over time $t \in [t_0, t_c(\mathbf{u})]$:

$$\frac{d}{dt}\frac{\partial \mathbf{q}}{\partial \mathbf{u}}(t) = \frac{\partial \mathbf{f}(t,\mathbf{q},\mathbf{u})}{\partial \mathbf{u}}, \quad \text{with initial condition} \quad \frac{\partial \mathbf{q}}{\partial \mathbf{u}}(t_0) = \frac{\partial \mathbf{q}_0}{\partial \mathbf{u}}. \tag{3.5}$$

Note that the integration computes a matrix $\frac{\partial \mathbf{q}}{\partial \mathbf{u}}$ instead of a vector.

The computation of $\partial \mathcal{F}_{t_f}(\mathbf{u}, \mathbf{q}(t_c^+))/\partial \mathbf{u}$ is similar. Applying the Leibnitz rule yields an integral equation:

$$\frac{\partial \mathcal{F}_{t_f}(\mathbf{u}, \mathbf{q}(t_c^+))}{\partial \mathbf{u}} = -\mathbf{f}(t_c(\mathbf{u}), \mathbf{q}, \mathbf{u})\frac{dt_c(\mathbf{u})}{d\mathbf{u}} + \frac{\partial \mathbf{q}(t_c^+)}{\partial \mathbf{u}} + \int_{t_c(\mathbf{u})}^{t_f} \frac{\partial \mathbf{f}(t,\mathbf{q},\mathbf{u})}{\partial \mathbf{u}}\, dt, \tag{3.6}$$

with the right-hand side terms evaluated as before. In this case, the second and third term, the expression $\frac{\partial \mathbf{q}(t_c^+)}{\partial \mathbf{u}} + \int_{t_c(\mathbf{u})}^{t_f} \frac{\partial \mathbf{f}(t,\mathbf{q},\mathbf{u})}{\partial \mathbf{u}}\, dt$, result in the same differential equation with different initial condition:

$$\frac{d}{dt}\frac{\partial \mathbf{q}}{\partial \mathbf{u}}(t) = \frac{\partial \mathbf{f}(t,\mathbf{q},\mathbf{u})}{\partial \mathbf{u}}, \quad \text{with initial condition} \quad \frac{\partial \mathbf{q}}{\partial \mathbf{u}}(t_c(\mathbf{u})) = \frac{\partial \mathbf{q}(t_c^+)}{\partial \mathbf{u}}.$$

Note that the initial condition $\mathbf{q}(t_c^+)$ of the post-collision free-flight function $\mathcal{F}_{t_f}$ is the output of the collision function $\mathcal{C}_{t_c}$.

3.2.2 Jacobian of Collision Functions

To compute the Jacobian of the collision function $\partial \mathcal{C}_{t_c}(\mathbf{u}, \mathbf{q}(t_c^-))/\partial \mathbf{u}$ we differentiate the impulse equation (Equation 2.8):

$$\frac{\partial \mathcal{C}_{t_c}(\mathbf{u}, \mathbf{q}(t_c^-))}{\partial \mathbf{u}} = \frac{\partial \mathbf{q}(t_c^-)}{\partial \mathbf{u}} + \frac{\partial \mathbf{j}(\mathbf{q}(t_c^-), \mathbf{u})}{\partial \mathbf{u}}.$$

Applying the chain rule on the second term yields

$$\frac{\partial \mathcal{C}_{t_c}(\mathbf{u}, \mathbf{q}(t_c^-))}{\partial \mathbf{u}} = \frac{\partial \mathbf{q}(t_c^-)}{\partial \mathbf{u}} + \frac{\partial \mathbf{j}}{\partial \mathbf{q}(t_c^-)}\frac{\partial \mathbf{q}(t_c^-)}{\partial \mathbf{u}} + \frac{\partial \mathbf{j}}{\partial \mathbf{u}}.$$

Because the pre-collision free-flight function outputs the state an instant before the collision, $\mathbf{q}(t_c^-) = \mathcal{F}_{t_c}$. Section 3.2.1 derives the Jacobian $\partial \mathbf{q}(t_c^-)/\partial \mathbf{u}$ from the Jacobian of the pre-collision free-flight function $\partial\mathcal{F}_{t_c}/\partial\mathbf{u}$. Assuming the impulse $\mathbf{j}(\mathbf{q}(t_c^-),\mathbf{u})$ is a smooth function of the pre-collision state $\mathbf{q}(t_c^-)$ and the control vector $\mathbf{u}$, we can evaluate the appropriate derivatives of the impulse function analytically and compute the Jacobian of the collision function.

As mentioned in Section 3.2.1 the pre-collision state $\mathbf{q}(t_c^-)$ depends on the derivative of collision time $dt_c(\mathbf{u})/d\mathbf{u}$. We derive this term by defining a smooth collision event function $E(t,\mathbf{q}(\mathbf{u}))$, which is zero at the collision time $t_c(\mathbf{u})$:

$$E(t,\mathbf{q}(\mathbf{u})) \overset{\text{def}}{=} \begin{cases} > 0, & \text{bodies do not collide} \\ = 0, & \text{bodies collide at time } t \overset{\text{def}}{=} t_c(\mathbf{u}) \\ < 0, & \text{bodies interpenetrate} \end{cases} . \tag{3.7}$$

For the 2-D particle, for example, the collision event function E can be defined as the signed-distance function between the particle and the obstacle. Differentiating Equation 3.7 and solving for the collision time derivative we obtain

$$\frac{dt_c(\mathbf{u})}{d\mathbf{u}} = -\frac{\left(\dfrac{\partial E}{\partial \mathbf{q}}\right)^T \dfrac{\partial \mathbf{q}}{\partial \mathbf{u}}}{\dfrac{\partial E}{\partial t}}. \tag{3.8}$$

Because the event function $E(t,\mathbf{q}(\mathbf{u}))$ is defined to be smooth and analytically differentiable, MOTIONEDITOR evaluates derivatives $\partial E/\partial\mathbf{q}$ and $\partial E/\partial t$ analytically. We defined the Jacobian of the pre-collision state $\partial\mathbf{q}/\partial\mathbf{u}$ by the second and third term in the integral expression in Equation 3.4. MOTIONEDITOR computes the value of this Jacobian by numerically integrating Equation 3.5.

3.2.3 Example of the Jacobian Computation

To illustrate the Jacobian computation procedure, consider an animation of an object in ballistic flight through a gravitational field. Furthermore, assume that the initial position and velocity of the body $\mathbf{q}_0$ are the only controllable simulation parameters $\mathbf{u}$. For this example, Equation 2.4 describes the free-flight motion. As in Section 3.2.1, we differentiate

Equation 2.4 to obtain a new differential equation for the Jacobian of free-flight motion:

$$\frac{d}{dt}\begin{pmatrix} \frac{\partial \mathbf{x}}{\partial \mathbf{u}}(t) \\ \frac{\partial \mathbf{r}}{\partial \mathbf{u}}(t) \\ \frac{\partial \mathbf{v}}{\partial \mathbf{u}}(t) \\ \frac{\partial \boldsymbol{\omega}}{\partial \mathbf{u}}(t) \end{pmatrix} = \begin{pmatrix} \frac{\partial \mathbf{v}}{\partial \mathbf{u}}(t) \\ \frac{1}{2}\frac{\partial}{\partial \mathbf{u}}\Big(\boldsymbol{\omega}(t) * \mathbf{r}(t)\Big) \\ \frac{\partial}{\partial \mathbf{u}}\Big(m^{-1}\mathbf{f}_e(t)\Big) \\ \frac{\partial}{\partial \mathbf{u}}\Big(\mathbf{I}(t)^{-1}\big(\boldsymbol{\tau}(t) - \boldsymbol{\omega} \times \mathbf{I}(t)\boldsymbol{\omega}\big)\Big) \end{pmatrix}. \tag{3.9}$$

Because the object's initial state is the control vector $\mathbf{u} = \mathbf{q}_0$, the initial condition for the differential equation is the identity matrix:

$$\frac{\partial \mathbf{q}}{\partial \mathbf{u}}(0) = \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix}.$$

MOTIONEDITOR computes the Jacobian of free-flight motion by numerically integrating the differential Equation 3.9. At each step of the numeric integration, the derivatives on the right-hand side are evaluated analytically. In this example, we can first simplify the right-hand side,

$$\frac{d}{dt}\begin{pmatrix} \frac{\partial \mathbf{x}}{\partial \mathbf{u}}(t) \\ \frac{\partial \mathbf{r}}{\partial \mathbf{u}}(t) \\ \frac{\partial \mathbf{v}}{\partial \mathbf{u}}(t) \\ \frac{\partial \boldsymbol{\omega}}{\partial \mathbf{u}}(t) \end{pmatrix} = \begin{pmatrix} \frac{\partial \mathbf{v}}{\partial \mathbf{u}}(t) \\ \frac{1}{2}\frac{\partial}{\partial \mathbf{u}}\Big(\boldsymbol{\omega}(t) * \mathbf{r}(t)\Big) \\ \mathbf{0} \\ -\frac{\partial}{\partial \mathbf{u}}\Big(\mathbf{I}(t)^{-1}\big(\boldsymbol{\omega} \times \mathbf{I}(t)\boldsymbol{\omega}\big)\Big) \end{pmatrix},$$

prior to differentiating the quaternion product $\boldsymbol{\omega}(t) * \mathbf{r}(t)$ and the term $\mathbf{I}(t)^{-1}[\boldsymbol{\omega} \times \mathbf{I}(t)\boldsymbol{\omega}]$. The simplification is possible because there is no external torque on the object and because the gravitational field and the mass of an object do not depend on the control vector $\mathbf{u}$.

In case of a collision, the computed Jacobian must be corrected with a Leibnitz correction term (Equations 3.4 and 3.6):

$$\mathbf{f}(t_c(\mathbf{u}), \mathbf{q}, \mathbf{u})\frac{dt_c(\mathbf{u})}{d\mathbf{u}} = \begin{pmatrix} \mathbf{v}(t_c(\mathbf{u})) \\ \frac{1}{2}\boldsymbol{\omega}(t_c(\mathbf{u})) * \mathbf{r}(t_c(\mathbf{u})) \\ m^{-1}\mathbf{f}_e(t_c(\mathbf{u})) \\ \mathbf{I}(t_c(\mathbf{u}))^{-1}\Big(\boldsymbol{\tau}(t_c(\mathbf{u})) - \big(\boldsymbol{\omega} \times \mathbf{I}(t_c(\mathbf{u}))\boldsymbol{\omega}\big)\Big) \end{pmatrix}\frac{dt_c(\mathbf{u})}{d\mathbf{u}}.$$

MotionEditor computes the collision-time derivative by defining a collision event function (Equation 3.8), which for a vertex-face collision becomes

$$E(t, \mathbf{q}(\mathbf{u})) = \mathbf{n}(\mathbf{q}(\mathbf{u})) \cdot \Big(\mathbf{p}_A(\mathbf{q}(\mathbf{u})) - \mathbf{p}_B(\mathbf{q}(\mathbf{u}))\Big),$$

where the position of the vertex is defined by $\mathbf{p}_A$ and where the planar face is defined by a point $\mathbf{p}_B$ on the plane and its surface normal $\mathbf{n}$.

MotionEditor completes the Jacobian computation by differentiating the collision impulse equation. For a frictionless vertex-face collision the simulator applies the impulse in the direction of the face normal:

$$\mathbf{j}(\mathbf{q}^-, \mathbf{u}) = j_s(\mathbf{q}^-, \mathbf{u})\ \mathbf{n}(\mathbf{q}^-).$$

The impulse magnitude j_s, derived from the Poisson collision model described in Section 2.1.4, depends on the inertial properties, relative velocities, and the two moment arms $\mathbf{r}_A$ and $\mathbf{r}_B$ of colliding bodies:

$$j_s(\mathbf{q}^-, \mathbf{u}) = \frac{-(1+\epsilon)\big(\mathbf{n} \cdot (\mathbf{v}_A^- - \mathbf{v}_B^-)\big)}{m_A^{-1} + \mathbf{n} \cdot \Big(\mathbf{I}_A^{-1}(t)\big(\mathbf{r}_A \times \mathbf{n}\big)\Big) \times \mathbf{r}_A + m_B^{-1} + \mathbf{n} \cdot \Big(\mathbf{I}_B^{-1}(t)\big(\mathbf{r}_B \times \mathbf{n}\big)\Big) \times \mathbf{r}_B}.$$

Although differentiating the collision impulse $\mathbf{j}$ is tedious, analytic computation is possible.

3.3 Differential Update

The editing technique is a form of gradient descent: MotionEditor continuously linearizes the problem and moves in the gradient direction $\delta\mathbf{u}$. For a large gradient stepsize ϵ, the gradient descent method may diverge. Line minimization is the preferred method for choosing the stepsize in a gradient method, but it requires considerable computation. In practice, for all problems described Section 3.6, a small fixed stepsize has good convergence properties while also enabling interactive update rates. Although gradient descent converges only to a local optimum [Bertsekas 95b], local convergence is sufficient and effective for the interactive setting. The animator drags a body towards the intended position—guiding MotionEditor out of undesirable local minima—and MotionEditor quickly reshapes the motion to comply with the change.

After computing the Jacobian of the motion, MOTIONEDITOR solves a minimization problem to compute a differential update $\delta\mathbf{u}$ for the control vector $\mathbf{u}$. The equations linearizing the simulation function (Equation 3.1) form the linear constraints for the optimization. MOTIONEDITOR assumes that the resulting problems are under-constrained, and solves the optimization problem:

$$\min_{\delta\mathbf{u}} \quad \left(\delta\mathbf{u}^T\mathbf{D}\,\delta\mathbf{u} + \mathbf{d}^T\delta\mathbf{u}\right) \tag{3.10}$$

$$\text{subject to} \quad \begin{cases} \delta\mathbf{q}_1 = \dfrac{\partial\mathcal{S}_{t_1}(\mathbf{u})}{\partial\mathbf{u}}\,\delta\mathbf{u} \\ \quad\vdots \\ \delta\mathbf{q}_n = \dfrac{\partial\mathcal{S}_{t_n}(\mathbf{u})}{\partial\mathbf{u}}\,\delta\mathbf{u}. \end{cases} \tag{3.11}$$

The optimization does not find a feasible update $\delta\mathbf{u}$ only when the design problem is over-constrained. In this case, MOTIONEDITOR does not update the control vector and the interaction halts. The animator must either add additional simulation parameters to increase the degrees of freedom or reduce the design requirements. When either is not possible the animator can use MOTIONSKETCHER described in the next chapter.

The quadratic objective function has a dual purpose: it seeks the smallest change from the current state of the simulation and the smallest deviation from the desired values of the simulation parameters. The diagonal matrix $\mathbf{D}$ describes the relative scale between parameters in the control vector $\mathbf{u}$. The animator can describe the desired scaling to specify how MOTIONEDITOR should change the parameters. For example, the animator may instruct MOTIONEDITOR to favor changing the initial position rather than the initial velocity of a body. The vector $\mathbf{d}$ defines desired values for physical parameters. For example, if MOTIONEDITOR varies the surface normal at a collision the animator can specify the true geometric normal as the desired value. MOTIONEDITOR will attempt to stay as close as possible to the true surface normal while satisfying the constraints. Specifically, if $\delta\mathbf{u}_d$ is the desired change in the control vector $\mathbf{u}$ then setting $\mathbf{d} = -2\delta\mathbf{u}_d$ and optimizing Equation 3.10 minimizes $(\delta\mathbf{u} - \delta\mathbf{u}_d)^T(\delta\mathbf{u} - \delta\mathbf{u}_d)$. This result follows from expanding the multiplication,

$$\begin{aligned}(\delta\mathbf{u} - \delta\mathbf{u}_d)^T(\delta\mathbf{u} - \delta\mathbf{u}_d) &= \delta\mathbf{u}^T\,\delta\mathbf{u} - 2\delta\mathbf{u}_d^T\delta\mathbf{u} + \delta\mathbf{u}^T\delta\mathbf{u} \\ &= \delta\mathbf{u}^T\,\delta\mathbf{u} + \mathbf{d}^T\delta\mathbf{u} + \delta\mathbf{u}_d^T\delta\mathbf{u}_d,\end{aligned}$$

and recognizing that the last term is a constant which does not affect minimization. Because the minimization objective is quadratic and all constraints are linear, MotionEditor uses Lagrange multipliers to reformulate the minimization as a linear system and to solve for the vector $\delta\mathbf{u}$ [Gill 89]. MotionEditor solves the linear system with either singular value decomposition or conjugate gradients [Golub 96]. The linear system has to be solved quickly for interactive editing. Singular value decomposition is more robust than conjugate gradients for design problems described in Section 3.6. These problems result in optimizations of approximately 30 simulation parameters, and singular value decomposition is sufficiently fast. However, the computational cost of singular value decomposition is a cubic function of the number of simulation parameters and this technique would not scale as well to larger optimization problems. In these cases, conjugate gradients is a faster technique that can exploit the sparse structure of the Jacobian matrix: the entries in the Jacobian matrix are zero whenever the motion of a body is not affected by a simulation parameter. For example, if two rigid bodies do not collide then their motion is not affected by initial positions and velocities of the other body.

3.4 Editing Constraints

When the animator edits the motion, MotionEditor maps the constraints into the desired motion changes $\delta\mathbf{q}_i$. MotionEditor distinguishes three types of constraints: state constraints, expression constraints, and floating constraints.

State constraints occur when the animator drags the objects to desired locations or "nails down" objects by fixing their positions or velocities. Suppose that the animator wants an object to be at position $\mathbf{p}_d$ at time t_i. MotionEditor formulates the desired differential constraint as $\delta\mathbf{q}_i = \mathbf{p}_d - \mathbf{p}(\mathbf{q}(t_i))$. In this case the nail constraint is enforced at a specific time instant t_i.

Expression constraints are generalizations of state constraints. Any differentiable expression of the generalized state $\mathbf{q}$ can represent a constraint. For example, the animator can equate the velocities of two bodies with the differential constraint $\delta\mathbf{q}_i = \mathbf{v}_A(\mathbf{q}(t_i)) - \mathbf{v}_B(\mathbf{q}(t_i))$.

Both state and expression constraints can be specified without fixing the time of evaluation t_i. The animator can express a constraint at a particular event (e.g., the fifth collision in the simulation). Time of collision $t_c(\mathbf{u})$ is not fixed and thus the time of the constraint can "float" with the collision. For example, the constraint $\delta\mathbf{q}_i = -\boldsymbol{\omega}\big(\mathbf{q}(t_c(\mathbf{u}))\big)^T\boldsymbol{\omega}\big(\mathbf{q}(t_c(\mathbf{u}))\big)$ reduces the angular velocity $\boldsymbol{\omega}$ when the object collides. Subsequent modification of the simulation parameters will change the time at which the collision occurs, but MOTIONEDITOR will still enforce the floating constraint.

Floating constraints do not require any modifications of the existing MOTIONEDITOR framework. The differential constraints in Equation 3.11 can still be evaluated because floating constraints are attached to well-defined collision events: the desired adjustment $\delta\mathbf{q}_i$ is computed at the appropriate collision and the Jacobians $\partial\mathcal{S}_{t_i}(\mathbf{u})/\partial\mathbf{u}$ of the collision function are evaluated as described in Section 3.2.2.

3.5 Discontinuities

When the simulation function is continuous, the interactive-editing technique effectively converges to the desired motion. In general, the simulation function contains discontinuities that may cause MOTIONEDITOR to diverge. This section describes a method for expanding the smooth, continuous, components of the simulation function. Described approaches improve the convergence of the differential approach.

3.5.1 Physical Feasibility

Section 2.1.5 describes a motion of a single particle after colliding and bouncing off the obstacle. As shown in Figure 2.3, the polygonal approximation of the obstacle restricts the sections on the unit circle that a particle can reach after the bounce. Note that some values of θ_f are unattainable because the surface normal near the origin is discontinuous: the particle cannot exit at the section of the circle directly above the origin (θ_f near $\pi/2$). This restriction of feasible results becomes especially evident when the animator over-constrains MOTIONEDITOR with many desired body configurations. Finer polygonal approximations reduce the gaps in the piecewise smooth function because in the limit polygonal approximations converge to the underlying smooth surface. However, an overly fine approximation would increase the running time of a collision-detection algorithm in a rigid-body simulation, which would in turn reduce the interactivity of the editing algorithm [Baraff 92].

The approach to this problem is twofold. First, additional control parameters can vary the surface normals on a polygonal mesh and simulate a collision with a smooth obstacle. If the mesh approximates a smooth surface, the desired normal can be computed from a smooth local interpolant or, if available, from the true smooth surface. When the control vector $\mathbf{u}$ includes the parameters that offset true surface normals, normals can be adjusted dynamically during the design process. Figure 3.3 illustrates that the varying surface normals extend the range of smooth components to increase the physically feasible regions.

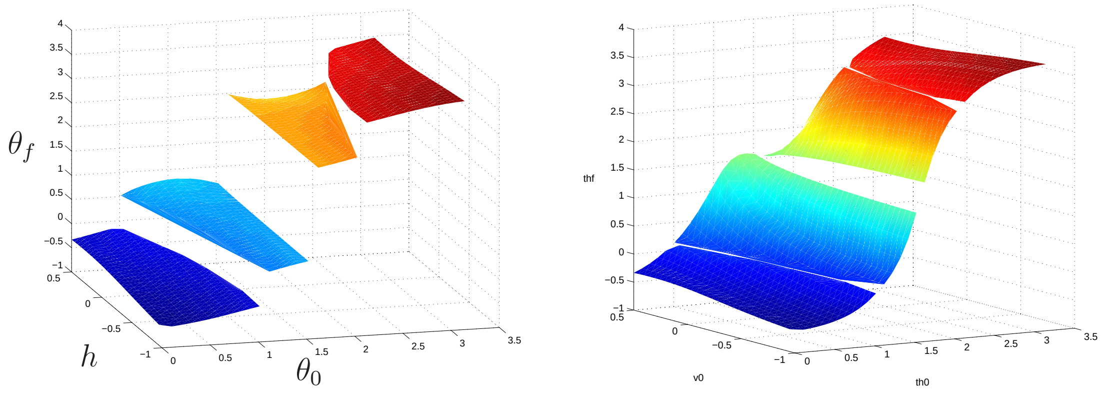


FIGURE 3.3. Simulation functions for a particle bounce against a non-smooth obstacle (left) is first describe in Section 2.1.5. If the surface normals are included in the control vector $\mathbf{u}$, the editing algorithm can vary the normals (right) to increase the size of smooth components, thus enlarging the physically feasible regions.

Second, MOTIONEDITOR uses curvature-dependent polygonal approximations in simulations because they keep the facet count low for fast collision detection and simulation, but also provide accurate approximations to the original surface [Garland 97]. Approximating smooth surfaces with polygonal meshes is well studied in computer graphics. In general, surface approximations allocate many facets to areas of high surface curvature and fewer facets to near-planar surface regions. For these approximating meshes, the colliding facet is an accurate first-order approximation to the underlying surface, despite the discontinuity in the surface normals. As a result, the differential change computed on one colliding facet is a first-order approximation to the differential change computed on the underlying surface. Because the differential update Equation 3.1 is also a linear approximation, the differential change $\delta\mathbf{u}$ continues to be a good predictor for the differential update.

3.5.2 Convergence

MotionEditor converges to the desired motion if there exists a connected path in the control space from the initial control vector to the desired control vector within a single smooth component. With discontinuities, such a path may not exist. To address this problem, a discrete search procedure must be introduced to guide the control vector between the appropriate components, piecing together a path that crosses discontinuities. Especially in higher dimensions, this is a daunting task for an interactive system. In general, the search must take into account physically feasible regions and jump to smooth components in possibly distant regions of a high-dimensional control space. The most important criterion for selecting smooth components is that they facilitate convergence to the desired motion. In addition, unless instructed otherwise, the components should preserve the "style" of the current motion by matching the current trajectories as closely as possible. For example, if an animator desires a successful "off-the-backboard" basketball shot, it is undesirable to jump to a smooth component corresponding to a direct, "nothing-but-net" motion. Lastly, the discrete search must complete quickly to maintain interactivity. MotionEditor addresses these problems with sampling and interaction techniques described in the following two sections.

3.5.3 Sampling

In the presence of discontinuities, the editing technique becomes more sensitive to the stepsize ϵ and the direction $\delta\mathbf{u}$ in the differential update in Equation 3.2. With a large stepsize ϵ, the gradient-descent method may diverge. The approximation errors in the differential vector $\delta\mathbf{u}$ also adversely affect convergence. MotionEditor improves convergence with a sampling procedure that finds the best values for the update stepsize ϵ and the direction $\delta\mathbf{u}$. A successive stepsize-reduction technique finds a reasonable stepsize ϵ, by reducing an initial stepsize until the motion matches the optimization constraints in Equation 3.10. Successive stepsize-reduction does not guarantee convergence, but performs well in practice [Bertsekas 95b].

Because polygonal meshes for curved surfaces lead to approximation errors in the gradient $\delta\mathbf{u}$, the differential update, a gradient-descent technique, may fail. Although convergence results for such gradients exist, there are no standard techniques for improving the convergence [Bertsekas 95b]. Recall from Section 3.5.1 that for discontinuities due to polygonal approximations, the update vector $\delta\mathbf{u}$ remains a reasonable gradient direction. Thus, when the simulation is directed off the boundary of a smooth component, MOTIONEDITOR samples the control space from the normal distribution centered around the suggested update $\delta\mathbf{u}$. Each such sample may produce a point on a new smooth component. The MOTIONEDITOR evaluates the corresponding motions and jumps to the most promising component. The animator perceives the jump as a minor “pop” in the resulting motion and typically, following the jump, the continuous manipulation continues. The sampling procedure also causes a momentary lag. While the lag could be reduced with a faster implementation, the visual pop is unavoidable in situations where the underlying motion is discontinuous. If sampling does not produce any reasonable smooth component, MOTIONEDITOR remains within the current smooth component. The animator is thus blocked from adjusting the motion in a particular way, but can continue to guide MOTIONEDITOR in a different manner.

3.5.4 Interaction

To guarantee convergence, MOTIONEDITOR would have to search through the entire control space. MOTIONEDITOR does not address this general problem—the high dimension of the control space makes the search especially difficult. Instead, MOTIONEDITOR relies on the animator to guide MOTIONEDITOR to a motion that satisfies given constraints. For example, a body that initially flies over a wall may have to bounce off the wall and fly in the opposite direction to accomplish the desired constraint. The editing technique will not make these transformations automatically. For a large class of motion design tasks, this behavior is desirable and sufficient. The interaction allows the animator to quickly experiment and guide MOTIONEDITOR toward the desired collision sequence. For example, to transform the motion of a basketball during a successful free throw, the animator may want to bounce the ball off the backboard before it goes through the hoop. In this case, the animator first guides the ball into a backboard collision, and then guides it through the hoop. Note that the single constraint specifying a successful shot does not uniquely determine the desired collision configurations: the ball may bounce off the backboard, off the floor or even off the scoreboard. An automatic system would have to choose the desired motion (or keep track of a possibly exponential number of motions) according to some objective criteria. Instead, MOTIONEDITOR provides the animator with interactive, direct control over the motion and allows her to guide MOTIONEDITOR to the appropriate solution.

3.6 Examples

Creating each example in this section required between two and fifteen minutes of interaction. Example figures show the animation before interaction, at an intermediate point, and after the desired motion is obtained. The motion is illustrated with trajectories that track one or more points on the body.

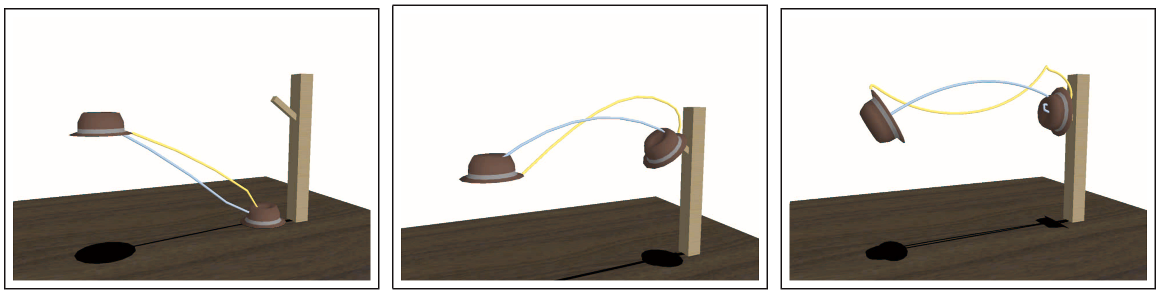


FIGURE 3.4. The animator drags the hat onto the coatrack to enforce the landing at the end of the animation. Once the hat lands safely, the animator creates a tumbling motion by spinning the hat at the starting time.

The first example, shown in Figure 3.4, recreates the example from Chapter 1. The hat misses the coatrack and lands on the ground. The animator selects the hat and simply drags it to the rack to enforce the successful landing. The dragging action enforces a 6-dimensional constraint that specifies the animator's adjustments to the hat's position and orientation. In response, MOTIONEDITOR computes the appropriate 6-dimensional control vector consisting of the hat's initial linear and angular velocity. Once the hat lands on the coatrack, the animator adds a 6-dimensional nail constraint that enforces the landing by fixing the hat's position and orientation at the end of the animation. The final adjustments spin the hat at the starting time, forcing it to tumble prior to the safe landing. The spin adjustments introduce an additional 3-dimensional constraint that specifies adjustments to the orientation of the hat. MotionEditor computes the appropriate 12-dimensional control vector, which describes the initial state (position, orientation, linear velocity, and angular velocity) of the hat that satisfies both the nail and the spin constraint.

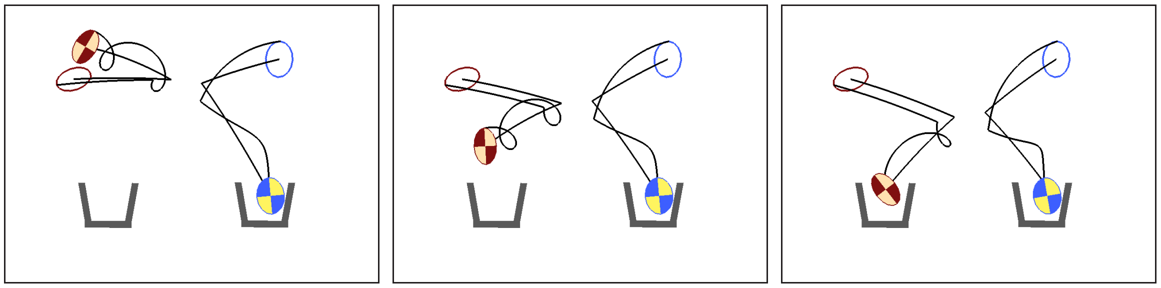


Figure 3.5. In less then two minutes, the animator creates an animation of two eggs colliding in mid-air and landing in the buckets.

The objective of the second example, as shown in Figure 3.5, is to have two 2-D eggs collide in the air and land successfully in two buckets on the ground. Creating this motion by simply adjusting initial positions and velocities of the objects would be difficult due to the complexity of the motion and the constraint that the buckets themselves cannot be moved. In contrast, the desired animation is easily created from scratch with MotionEditor. First, the starting positions of the eggs are fixed, and the velocities and orientations are assigned arbitrarily. By selecting an egg on its flight path, the animator interactively drags the first egg's trajectory towards the second egg until the two objects collide in the air. During this interaction, the constraint is a 3-dimensional vector consisting of the position and orientation adjustments. The 3-dimensional control vector consists of the egg's initial linear and angular velocity. Running at roughly 20 frames per second, MotionEditor computes the required changes in the initial orientation and velocity of both eggs to achieve the desired motion updates. Once one egg is in the bucket, the animator applies a nail constraint to fix its ending state and drags the second egg into the other bucket. In this case, the two constraints are two 3-dimensional vectors both describing the adjustments in position and orientation. The 6-dimensional control vector consists of the initial linear and angular velocities of both eggs.

In the third example, shown in Figure 3.6, the animator's goal is to drop a plank onto two supports to form a table. The problem is made more difficult by requiring the plank to

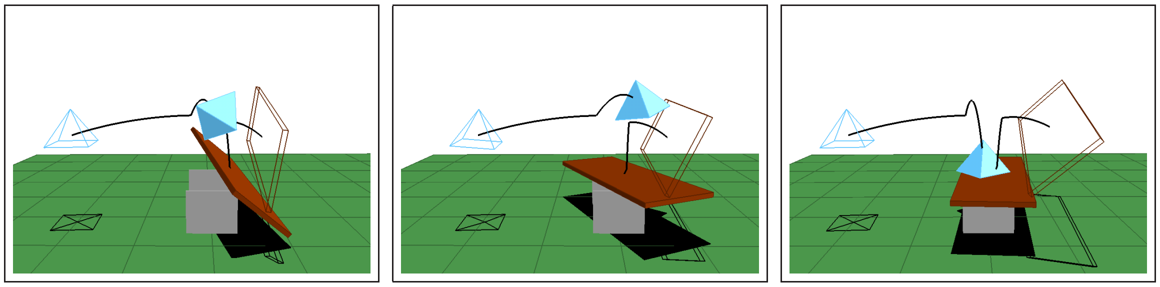


FIGURE 3.6. A table top lands on its legs after a mid-air collision with a pyramid.

collide with a pyramid, prior to landing centered on the supports. This example requires MOTIONEDITOR to solve for the initial plank position, orientation, and velocity (both linear and angular) in order to achieve the desired configuration after the collision. As in the previous example, the animator directly manipulates the plank's position and orientation while MOTIONEDITOR interactively computes the corresponding simulation parameters. This editing occurs in two steps. First, the animator selects the plank after its collision with the pyramid, and positions it above the supports. Second, the animator aligns the plank's orientation so that it lands squarely on the supports. In both instances, the constraint is a 6-dimensional vector describing adjustments to the position and orientation of the plank. The 24-dimensional control vector describes the initial generalized state of both bodies.

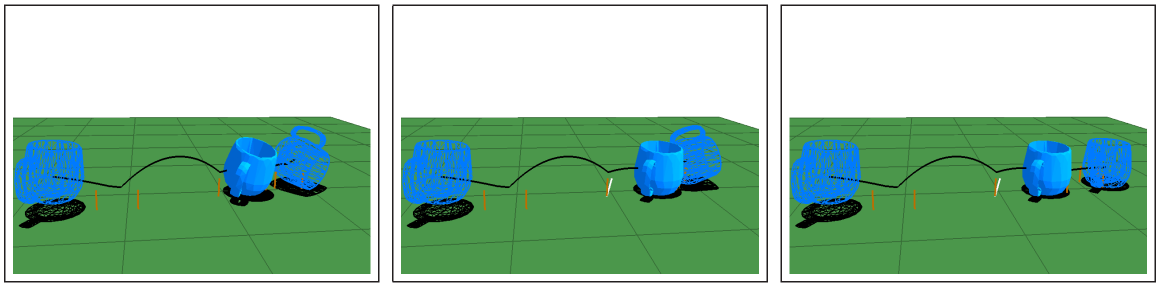


FIGURE 3.7. The animator adjusts an animation of a tumbling mug to prevent it from tipping over. The floating constraints adjust the mug's orientation and reduce its angular velocity.

The fourth example, shown in Figure 3.7, demonstrates the use of floating constraints and additional control parameters. Suppose the animator wishes to keep a falling mug from tipping over without changing its initial position, orientation, or velocity. This is accomplished by adding new control parameters that control the surface normals and elasticity coefficients at each collision. To keep the mug from tipping over, the animator first corrects the orientation, with a 3-dimensional orientation constraint, so that the mug is upright at the fourth bounce. MotionEditor accommodates the change by adjusting the surface normals, through a 12-dimensional control vector, at the four prior bounces. Note that these adjustments alter the time at which the fourth bounce occurs, requiring a floating time constraint (Section 3.4). Because of the angular velocity, however, the mug still tips over. The animator prevents the tipping by specifying a 3-dimensional velocity constraint that reduces the angular velocity after the fourth bounce to keep the mug upright after the bounce. The adjustments to surface normals and elasticity coefficients, through a 16-dimensional control vector, are perceived as changes in the floor texture but their effect on the motion allows the animator to create the desired animation.

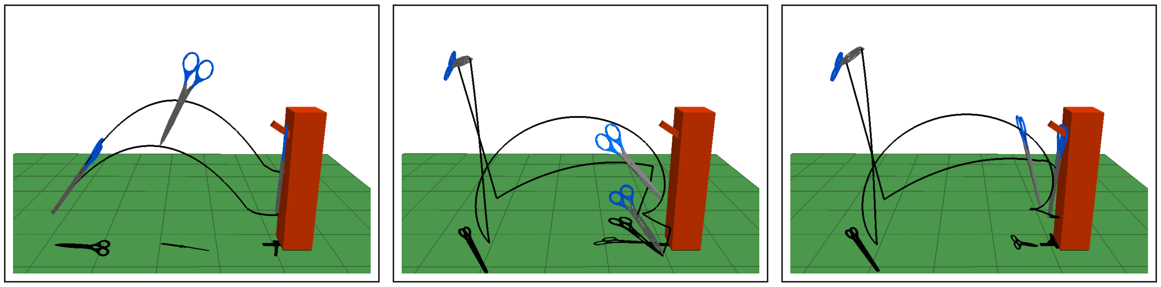


FIGURE 3.8. Scissors bounce and flip before landing on the hook.

In the fifth example, shown in Figure 3.8, an animator constructs a complex motion of the scissors bouncing off the floor, flipping in the air, and landing on the hook. First, the animator enforces the bounce by dragging the scissors towards the floor. Second, spinning the scissors after the bounce produces a somersault motion. Final tweaks ensure that the scissors land on the hook. The progressive interaction is typical of a MotionEditor session. A complex animation is a result of successive edits to an initially simple animation. In all edits, the 6-dimensional constraints adjust the position and orientation of the scissors and the computed 12-dimensional control vector specifies the initial generalized state of the scissors.

The last example, shown in Figure 3.9, illustrates the benefits of additional simulation parameters for controlling motion of chaotic systems. In this animation the die must land

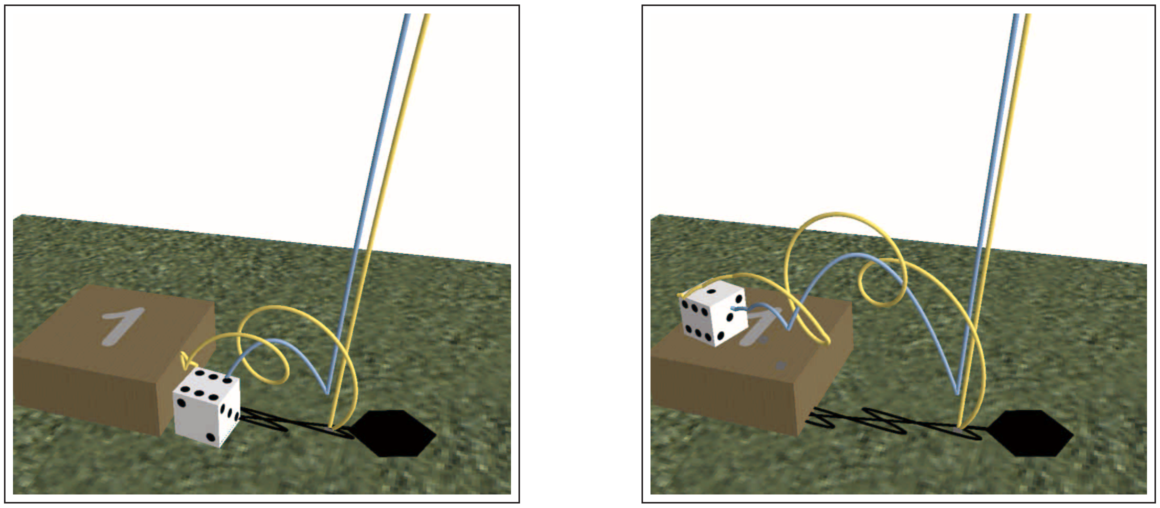


FIGURE 3.9. The outcome of a die toss changes as the animator adjusts the orientation at an earlier bounce.

on the stand with the desired outcome. The animator first drags the die on top of the stand, but the die lands with the wrong side up. The animator could tilt the die forward to obtain the desired outcome but the chaotic system is extremely sensitive to initial position and velocity of the die. Instead, the animator instructs MOTIONEDITOR to adjust surface normals, through a 9-dimensional control vector, at each of the three collisions. The outcome of a die toss is changed as the animator adjusts the orientation at an earlier bounce. Adjusting the surface normals improves the conditioning of the problem, and the animator proceeds to tilt the die with a floating 3-dimensional orientation constraint.

Chapter 4

MOTION SKETCHER

MOTIONEDITOR, the interactive editing technique described in the previous chapter, is an effective and intuitive tool for an important class of motion-design problems. The interaction is critical for this editing approach, without interaction the animator could not edit the motion as easily and effectively. Because the interactive performance hinges on the speed of physical simulation, this editing approach is ineffective when the motion of rigid bodies cannot be simulated interactively. Furthermore, despite many advantages, interactive editing may be tedious and time consuming for design of highly improbable motions. For example, designing the motion of scissors described in Section 3.6 required fifteen minutes of interaction because the animator has to guide the editor continuously until it converges to the intended motion.

MOTIONEDITOR, an offline design tool described in this chapter, takes a different approach: the animator designs the motion before any physical motion is computed. Off-line, MOTIONSKETCHER computes a physical rigid-body motion that closely matches the animator's design: a sketch of the desired motion. With this approach, the design process is not impeded even when simulating the physical motion is computationally slow. Furthermore, the animator can specify detailed and physically meaningful sketch, without carefully guiding the tool until its convergence to the desired motion.

MOTIONSKETCHER consists of a sketching interface and a parameter solver. The interface allows the animator to sketch the motion and to specify constraints that enforce key aspects of the motion. The parameter solver computes the simulation parameters, by solving a parameter estimation problem that yields the physical motion that closely matches the sketch.

The sketching approach introduces three new challenges. First, the motion implied by a sketch may be physically infeasible. Even for physically feasible goals, finding *any* motion that achieves the goals is a challenge. Second, motion sketches are imprecise. Except for the parts of a sketch that specify explicit constraints, sketches convey the intent but do not define precise behavior. Consequently, motion sketches may differ dramatically from the desired physical motion. Third, a sketch may have an incorrect, or non-physical, timing. For example, the motion may be sketched in slow motion and MOTIONSKETCHER must determine the correct timing using physical laws, such as the motion of rigid bodies in a gravitational field.

MOTIONSKETCHER employs a generalized multiple-shooting technique to solve the parameter estimation problem in its discrete form. In contrast to traditional formulations, which require that the animator specifies the timing of constraints, MOTIONSKETCHER computes both the motion and the timing that matches the sketched motion as closely as possible. Multiple shooting is designed to work effectively in a mixed, continuous and discrete, optimization domain. MOTIONSKETCHER aligns the discretization time grid with rigid-body collisions, which greatly simplifies the search in the discrete domain. The alignment, however, need not enforce the timing of collisions. The use of this sliding time grid allows the optimization procedure to adjust the timing of collisions while fitting the animator's sketch. In practice, this optimization procedure converges rapidly and effectively.

Section 4.1 presents the top-level description of the MOTIONSKETCHER algorithm. Section 4.2 formulates motion sketching as a parameter estimation problem. Section 4.3 introduces multiple shooting and reformulates the parameter estimation problem. Section 4.4 describes techniques for resolving discontinuities in the simulation function. Section 4.5 presents the two sketching interfaces and examples of animations constructed with the interfaces.

4.1 Basic Algorithm

MOTIONSKETCHER creates a rigid-body motion by solving a parameter estimation problem. The animator designs the motion of a rigid body by sketching its desired 3-D trajectories with either the acting interface or the editing interface. With the acting interface, the animator moves real objects with attached motion sensors to sketch a motion. With the editing interface, the animator creates the sketch by interactively manipulating rigid-body simulations. The resulting trajectories specify the desired motion of animated objects. For example, Figure 4.1 illustrates the sketching process with the acting interface. The animator moves the pencil to act out an animation of a pencil bouncing, twirling in the air, and landing in a mug.

The animator need not move the pencil in a physically correct manner; both the trajectories and the timing could be physically implausible. For example, the animator may move the pencil in slow motion. To allow for precise control over the animation, any part of the sketch may be designated as a hard kinematic constraint. For instance, the animator may choose to require that the pencil lands at a precise location, or that it collide with another object at a specific point in time.

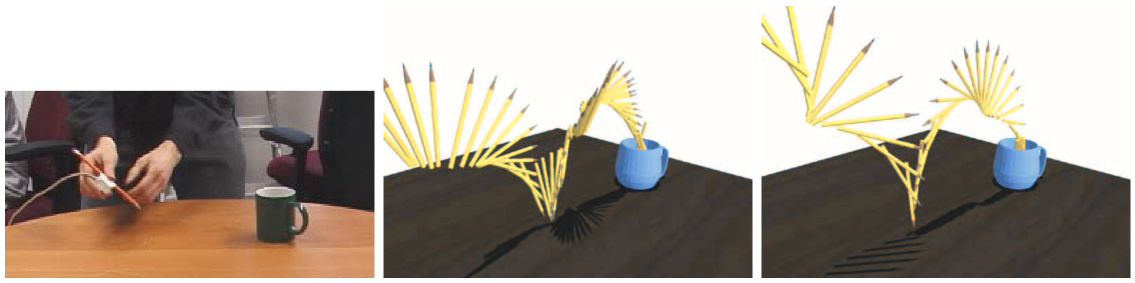


FIGURE 4.1. The animator moves an instrumented pencil to act out the desired behavior. A sampled representation of the sketched motion (middle figure) is transformed into a physical motion that closely matches the sketch.

Prior to solving the parameter estimation problem, MOTIONSKETCHER transforms the sketch into a canonical form that serves as the initial approximation for the subsequent optimization procedure. In this step, the animator annotates the sequence of collisions in the sketch. The collision sequence partitions the rigid-body motion into free-flight segments. MOTIONSKETCHER computes the initial approximation by fitting a motion to each free-flight segment independently.

In the last step, MOTIONSKETCHER solves a parameter estimation problem to compute the simulation parameters. A generalized multiple-shooting method fits the motion within each free-flight segment while ensuring the correct behavior at each collision. The result is physically correct motion that closely adheres to the sketch. If desired, strict adherence to physical laws may be relaxed at specified points in the animation. For instance, the animator may choose to allow small changes in energy at collisions or visually imperceptible discontinuities in position or orientation, yielding physically plausible motions that better fit the sketch.

4.2 Parameter Estimation

The goal of parameter estimation is to compute the control vector $\mathbf{u}$ for which the resulting motion $\mathbf{q}(t)$ matches the sketch. A sketch of the motion is represented by sketch samples $\mathbf{s}_1, \dots, \mathbf{s}_n$. Each sketch sample $\mathbf{s}_i$ specifies desired values for an arbitrary function of the generalized state $\mathbf{p}_i(\mathbf{q})$. For example, if a sketch sample specifies the desired orientation of a body at time t_i then the function $\mathbf{p}_i\big(\mathbf{q}(t_i)\big)$ projects the body's orientation components from the generalized state $\mathbf{q}(t_i)$. The control vector $\mathbf{u}$ typically includes initial positions and velocities of rigid bodies, but can also include collision elasticity and collision normals. The parameter estimation formulation is general and MOTIONSKETCHER could be generalized to optimize additional simulation parameters such as masses, moments of inertia, and the acceleration of gravity.

Any metric could be used to measure how well the motion $\mathbf{q}(t)$ matches the sketch samples $\mathbf{s}_1, \dots, \mathbf{s}_n$. The least-squares metric is a reasonable choice because it is a maximum-likelihood estimator[1] with nice numerical properties [Stengel 94]. The least-squares metric evaluates the motion, but it does not enforce required sketch goals. For example, an animator may sketch the motion of a successful basketball free-throw, but the best least-squares motion may result in a miss. MOTIONSKETCHER addresses this problem by allowing sketch samples that constrain the required motion goals. The animator could enforce a successful free-throw by constraining the basketball's center-of-mass to fall through the rim.

The aim of parameter estimation is to compute the motion $\mathbf{q}(t)$ and parameters $\mathbf{u}$ that minimize the least-squares distance to sketch samples, subject to complying with sketch constraints:

$$\begin{aligned} \min_{\mathbf{q}(t),\mathbf{u}} \quad & \sum_{i=1}^{n} \|\mathbf{p}_i\big(\mathbf{q}(t_i),\mathbf{u}\big) - \mathbf{s}_i\|^2 \\ \text{subject to} \quad & \underline{\mathbf{b}} \le \mathbf{c}_s\big(\mathbf{u}, \mathbf{q}(t_1), \dots, \mathbf{q}(t_n)\big) \le \overline{\mathbf{b}}, \end{aligned} \tag{4.1}$$

where the constant lower $\underline{\mathbf{b}}$ and upper $\overline{\mathbf{b}}$ bounds permit both equality and inequality sketch constraints. The formulation attempts to preserve the style and goal of a sketch. The

[1]Least-squares fitting is a maximum likelihood estimator if the sketch samples are independently distributed according to a normal (Gaussian) distribution of constant standard deviation.

least-squares objective function expresses the intended style (i.e., the motion shape) of the sketch, and the sketch constraints enforce specific goals of the sketch. Solving the parameter estimation problem yields a physical motion that meets the motion goals and preserves the motion style as much as possible. The formulation in Equation 4.1 is a variational form because the entire motion path $\mathbf{q}(t)$ is optimized. The next section describes a shooting technique for solving parameter estimation problems of this form.

4.3 Multiple Shooting

Shooting techniques solve a variational problem, such as the parameter estimation problem defined in Equation 4.1, by parameterizing the entire motion with a finite set of parameters. For example, a single-shooting method, illustrated in Figure 4.2, would parametrize the entire motion $\mathbf{q}(t)$ with the generalized state $\mathbf{q}_0$ at the initial time because the entire motion can be computed by numerically integrating equations of motion from the initial condition $\mathbf{q}_0$. MOTIONEDITOR is one example of this approach.

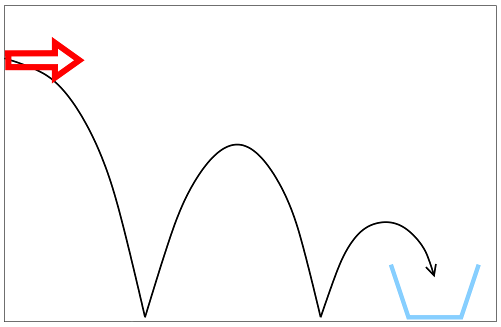


FIGURE 4.2. Single shooting parameterizes the entire motion with the state at the initial time. In this illustration, the motion of a particle is parameterized by its initial velocity (arrow).

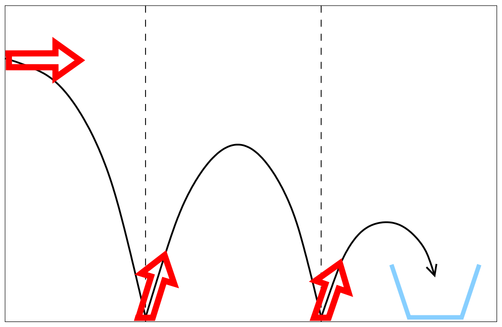


FIGURE 4.3. Multiple shooting subdivides the integration interval before parameterizing the motion. In this illustration, the motion of a particle within each segment is parameterized by its initial velocity at the beginning of each free-flight segment (arrows).

Although single shooting is intuitive and easy to implement it has two well-known drawbacks: long integration interval and implicit motion representation [Ascher 88]. The long integration interval results in numerical methods with poor stability. Although numerical instability can be compensated for with good initial approximations, the implicit motion representation prevents the animator from easily describing good initial approximations. For example, an animator cannot directly specify the number of bounces a die takes as it bounces across the table. Instead, the animator must deduce the initial state $\mathbf{q}_0$ that results in the motion with the correct number of bounces.

Multiple shooting, illustrated in Figure 4.3, is a numerically stable and robust technique for solving the parameter estimation problem described in Section 4.2. By integrating equations of motion within a series of shorter integration intervals, multiple shooting eliminates the main drawback to single shooting. Furthermore, multiple shooting exposes the generalized state of the body at more than just the initial time allowing the animator to provide better initial approximations. Section 4.3.1 describes multiple shooting in a general setting, rewriting parameter estimation in discrete form. Section 4.3.2 illustrates the multiple-shooting technique with a concrete example.

4.3.1 General Formulation

Shooting techniques for the parameter estimation problem compute the motion $\mathbf{q}(t)$ of a rigid body system by evaluating the simulation function $\mathcal{S}(t, \mathbf{u})$ [Bock 83, Bock 80]. The simulation function $\mathcal{S}(t, \mathbf{u})$, defined in Section 2.1, integrates equations of motion to map the simulation parameters $\mathbf{u}$ into the rigid-body motion $\mathbf{q}(t)$. Multiple shooting subdivides the interval of integration $[t_0, t_n]$ to compute the motion on each subinterval separately. A time grid,

$$t_0 = t_0^G < t_1^G < \cdots < t_m^G = t_n,$$

splits the interval of integration into m subintervals. The subdivision can be arbitrary. Fine-grained subdivision improves stability but increases the size of the optimization problem. After subdivision, the motion $\mathbf{q}(t)$ is defined by m functions:

$$\mathbf{q}(t) \overset{\text{def}}{=} \begin{cases} \mathcal{S}(t, \mathbf{u}, \mathbf{q}_0) & \text{if } t \in [t_0^G, t_1^G), \\ & \vdots \\ \mathcal{S}(t, \mathbf{u}, \mathbf{q}_{m-1}) & \text{if } t \in [t_{m-1}^G, t_m^G]. \end{cases}$$

Each function is a solution of an initial value problem, completely determined by simulation parameters $\mathbf{u}$ and initial states $\mathbf{q}_i = \mathbf{q}(t_i^G)$. Conceptually, given the control vector $\mathbf{u}$

and m initial states $\mathbf{q}_0, \ldots, \mathbf{q}_{m-1}$, the rigid-body simulator computes the simulation function, which specifies the state of the bodies in the world at every point in time:

$$\mathbf{q}(t) = \mathcal{S}(t, \mathbf{u}, \mathbf{q}_0, \ldots, \mathbf{q}_{m-1}), \quad t \in [t_0, t_n].$$

In general, the motion $\mathbf{q}(t)$ so computed is not continuous. The sequence $t_1^G, \ldots, t_{m-1}^G$ marks the times between the segments, where the neighboring motions may not match up. Continuity constraints $\mathbf{c}_c$ are added to enforce continuity between motion segments, ensuring that the motion $\mathbf{q}(t)$ is continuous in both position and velocity:[2]

$$\mathbf{c}_c(\mathbf{u}, \mathbf{q}_0, \ldots, \mathbf{q}_{m-1}) \overset{\text{def}}{=} \begin{pmatrix} \mathbf{q}_1 - \mathcal{S}_{t_1^G}(\mathbf{u}, \mathbf{q}_0) \\ \vdots \\ \mathbf{q}_{m-1} - \mathcal{S}_{t_{m-1}^G}(\mathbf{u}, \mathbf{q}_{m-2}) \end{pmatrix} = 0$$

In this formulation, the simulation parameters $\mathbf{u}$ and initial states $\mathbf{q}_0, \ldots, \mathbf{q}_{m-1}$ are the unknowns. The parameter estimation procedure computes their values to satisfy the sketch constraints $\mathbf{c}_s$ and to enforce the continuity constraints $\mathbf{c}_c$:

$$\begin{array}{ll} \min\limits_{\mathbf{u}, \mathbf{q}_0, \cdots, \mathbf{q}_{m-1}} & \sum_{i=1}^{n} \left\| \mathbf{p}_i\Big(\mathcal{S}_{t_i}(\mathbf{u}, \mathbf{q}_0, \ldots, \mathbf{q}_{m-1})\Big) - \mathbf{s}_i \right\|^2 \\ \text{subject to} & \begin{cases} \underline{\mathbf{b}} \leq \mathbf{c}_s\big(\mathbf{u}, \mathbf{q}(t_1), \ldots, \mathbf{q}(t_n)\big) \leq \overline{\mathbf{b}} \\ \mathbf{c}_c(\mathbf{u}, \mathbf{q}_0, \ldots, \mathbf{q}_{m-1}) = \mathbf{0}. \end{cases} \end{array} \tag{4.2}$$

Provided the simulation function $\mathcal{S}(t, \mathbf{u}, \mathbf{q}_0, \ldots, \mathbf{q}_{m-1})$ is continuous with respect to the control vector $\mathbf{u}$ and the initial states $\mathbf{q}_0, \ldots, \mathbf{q}_{m-1}$, the multiple-shooting formulation yields a constrained nonlinear optimization problem on a continuous domain which can be solved by a variety of efficient optimization methods [Gill 89].

MOTIONSKETCHER uses SNOPT optimization software [Gill 97], a sparse optimization solver based on sequential quadratic programming (SQP). SQP methods solve constrained nonlinear optimization problems by generating a sequence of iterates that converge to a solution point satisfying the Karush-Kuhn-Tucker conditions of optimality. Each iterate is the result of a quadratic programming subproblem, which is derived from the original nonlinear optimization.

[2]Velocity is not continuous for colliding bodies. In this case, the states must still match up through the impulse Equation 2.8.

For highly nonlinear optimization problems, a good initial approximation of the solution is a key requirement for convergence. SQP methods find solutions that are locally optimal, but with good initial approximations a locally optimal solution is often a global optimum. One of the primary benefits of multiple shooting is that prior knowledge about the desired solution, incorporated in the sketch, can be used to construct a good initial approximation. Section 4.5 describes the specifics of constructing an initial approximation from sketching interfaces.

SQP methods rely on accurate derivatives of the least-squares objective function and the constraints with respect to the optimization unknowns: control vector $\mathbf{u}$, and initial conditions $\mathbf{q}_0, \ldots, \mathbf{q}_{m-1}$. MOTIONSKETCHER uses a specialized automatic differentiation technique to compute these Jacobian matrices. The Jacobian matrices are numerically composed, using the chain rule, from analytically differentiable functions that comprise the simulation function. Section 3.2 describes this derivation in detail.

SNOPT and other sparse optimization solvers can exploit the structure in the Jacobian matrices to improve optimization speed and to scale to large optimization problems. In the parameter estimation problem, the Jacobian matrices of both the sketch $\mathbf{c}_s$ and the continuity constraints $\mathbf{c}_c$ are sparse. The Jacobian matrices have a sparse structure because initial states do not affect any constraint values outside the corresponding shooting segment. For example, the Jacobian matrix of the continuity constraints $\mathbf{c}_c$ with respect to initial conditions $\mathbf{q}_0, \ldots, \mathbf{q}_{m-1}$ is a banded matrix:

$$\frac{\partial \mathbf{c}_c}{\partial \mathbf{q}_p} = \begin{pmatrix} -\frac{\partial \mathcal{S}_{t_1^G}}{\partial \mathbf{q}_0} & \mathbf{1} & 0 & 0 & \cdots & 0 \\ 0 & -\frac{\partial \mathcal{S}_{t_2^G}}{\partial \mathbf{q}_1} & \mathbf{1} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & -\frac{\partial \mathcal{S}_{t_{m-1}^G}}{\partial \mathbf{q}_{m-2}} & \mathbf{1} \end{pmatrix}, \text{ where } \mathbf{q}_p \overset{\text{def}}{=} \begin{pmatrix} \mathbf{q}_0 \\ \mathbf{q}_1 \\ \vdots \\ \mathbf{q}_{m-1} \end{pmatrix},$$

and matrix $\mathbf{1}$ is an identity matrix. Section 4.4 describes a modified estimation technique for discontinuous optimization space. The new technique will result in denser, but still sparse, Jacobian matrices.

4.3.2 An Illustrative Example

Suppose the animator sketches an animation of a pencil bouncing, twirling in the air, and landing in a mug. Figure 4.1 illustrates the sketching process and the resulting sketch. In this example, the sketched trajectories indicate the position $\mathbf{x}_s(t)$ and orientation $\mathbf{r}_s(t)$ of the pencil. MOTIONSKETCHER discretizes the continuous sketch trajectories with sketch samples $\mathbf{s}_i \in \mathbf{R}^3 \times \mathrm{SO}(3)$:

$$\mathbf{s}_i \overset{\text{def}}{=} \begin{pmatrix} \mathbf{x}_s(t_i) \\ \mathbf{r}_s(t_i) \end{pmatrix}, \quad i \in [1, n].$$

Although in general the sketch samples do not contain the timing information t_i, this section assumes that the timing is provided. Section 4.3.3 describes the general setting, when the timing information is not available.

For each sketch sample, the projection functions $\mathbf{p}_i(\mathbf{q}(t_i))$ extract the position and orientation of the pencil from the generalized state:

$$\mathbf{p}_i\left(\begin{pmatrix} \mathbf{x}(t_i) \\ \mathbf{r}(t_i) \\ \mathbf{v}(t_i) \\ \boldsymbol{\omega}(t_i) \end{pmatrix} \right) \overset{\text{def}}{=} \begin{pmatrix} \mathbf{x}(t_i) \\ \mathbf{r}(t_i) \end{pmatrix}, \quad i \in [1, n].$$

Sketched trajectories indicate the preferred path for the pencil. The animator will also introduce an explicit motion goal to ensure that the pencil lands in the mug. Noting that the last sketch sample $\mathbf{s}_n$ locates the pencil inside the mug, one simple method would construct an equality sketch constraint $\mathbf{c}_s$ that matches the final position and orientation of the pencil $\mathbf{p}_n\big(\mathbf{q}(t_n)\big)$ with the last sketch sample:

$$\mathbf{c}_s = \mathbf{p}_n\big(\mathbf{q}(t_n)\big) - \mathbf{s}_n = 0. \tag{4.3}$$

In this example, the pencil bounces once. The time of this collision t_1^G is a natural segmentation point for multiple shooting. The time grid splits the motion into two segments: the pre-collision and the post-collision free-flight segment. As discussed in Section 4.3.1, the motion is computed within two shooting segments independently:

$$\mathbf{q}(t) \overset{\text{def}}{=} \begin{cases} \mathcal{S}(t, \mathbf{u}, \mathbf{q}_0) & \text{if } t \in [t_0^G, t_1^G], \\ \mathcal{S}(t, \mathbf{u}, \mathbf{q}_1) & \text{if } t \in [t_1^G, t_2^G]. \end{cases}$$

The initial states $\mathbf{q}_0$ and $\mathbf{q}_1$ denote the generalized state of the pencil at the beginning of each interval. Parameter estimation computes the values of the initial states and the control vector $\mathbf{u}$ that minimize the least-squares metric subject to satisfying the sketch constraint $\mathbf{c}_s$.

The continuity constraint $\mathbf{c}_c$ ensures that the initial state $\mathbf{q}_1$ matches the ending of the previous shooting segment:

$$\mathbf{c}_c(\mathbf{u}, \mathbf{q}_0, \mathbf{q}_1) \overset{\text{def}}{=} \mathbf{q}_1 - \mathcal{S}_{t_1^G}(\mathbf{u}, \mathbf{q}_0) = 0.$$

Because the generalized state describes the position and velocity of each body, this constraint enforces the continuity in both. If the initial state $\mathbf{q}_1$ occurs an instant after the bounce then the simulation function $\mathcal{S}_{t_1^G}(\mathbf{u}, \mathbf{q}_0)$ evaluates the pre-collision free-flight motion and applies the impulse through the collision function to compute the appropriate post-collision state.

Putting it all together, MOTIONSKETCHER formulates the parameter estimation problem as a constrained minimization:

$$\min_{\mathbf{u}, \mathbf{q}_0, \mathbf{q}_1} \quad \sum_{i=1}^{n} \left\| \mathbf{p}_i\Big(\mathcal{S}_{t_i}(\mathbf{u}, \mathbf{q}_0, \mathbf{q}_1)\Big) - \mathbf{s}_i \right\|^2 \tag{4.4}$$

$$\text{subject to} \quad \begin{cases} \mathbf{0} \le \mathbf{p}_n\big(\mathcal{S}_{t_n}(\mathbf{u}, \mathbf{q}_1)\big) - \mathbf{s}_n \le \mathbf{0} \\ \mathbf{q}_1 - \mathcal{S}_{t_1^G}(\mathbf{u}, \mathbf{q}_0) = \mathbf{0}. \end{cases} \tag{4.5}$$

The optimization computes the control vector $\mathbf{u}$ which consists of simulation parameters such as surface normals and elasticity coefficients at the bounce, and the two 12-dimensional initial conditions $\mathbf{q}_0, \mathbf{q}_1$.

4.3.3 Sketch Sample Timing Assignment

Although an animator typically knows what motion trajectories should look like, she need not know precisely when any motion event should occur. In such a case, sketch samples $\mathbf{s}_1, \ldots, \mathbf{s}_n$ encode the desired body trajectories, and MOTIONSKETCHER computes the timing information t_i from sketched trajectories alone.

The iterative algorithm, shown in Figure 4.4, alternates two steps: the assignment step and the optimization step. First, the assignment step estimates the timing t_i by pairing each

```
compute initial approximation $\mathbf{q}(t)$
repeat
   /* Assignment Step */
   for $i = 1$ to $n$ do
      $t_i = \arg\min_{t_i} \left\| \mathbf{p}_i\big(\mathbf{q}(t_i)\big) - \mathbf{s}_i \right\|^2$ such that $t_1 < \cdots < t_i$
   end for
   /* Optimization Step */
   solve parameter estimation problem in Equation 4.2
until timing $t_i$ changes little from previous iteration
```

FIGURE 4.4. The iterative algorithm assigns the timing information to each sketch sample before solving the parameter estimation problem.

sketch sample $\mathbf{s}_i$ with a closest state $\mathbf{q}(t_i)$. Second, given the sample timing t_i, the optimization step solves the parameter estimation problem described in Section 4.3.1. MOTIONSKETCHER alternates the assignment and the optimization steps until convergence. As described in Section 4.3.1, the optimization uses an iterative SQP procedure. For efficiency, MOTIONSKETCHER ends the optimization step after only a few iterations before repeating the assignment step to re-estimate the timing. After several repetitions the timing estimates stabilize in all of our examples, varying little with each repetition, and MOTIONSKETCHER completes the optimization until convergence.

A good initial timing estimate is critical for the success of the iterative algorithm. Both the parameter estimation described in Section 4.2 and the timing assignment are nonlinear and nonconvex optimizations with many local solutions. For convergence of both problems, the initial approximation must be sufficiently close to the desired solution. In this case, the initial approximation must yield an approximate timing estimate that can be locally optimized with the iterative procedure. Section 4.5 describes the specifics of constructing an initial motion from sketching interfaces.

The iterative assignment-optimization algorithm is similar in spirit to the iterative-closest-point (ICP) method for registering 3-D shapes from multiple views [Besl 92]. Registration aligns the shapes, initially expressed in multiple coordinate systems, to express the data in a single, object coordinate system. For registration, ICP alternates pairing the points in overlapping shape regions, and optimizing the coordinate transformations for the current point-pairing.

4.4 Discontinuities

With collisions, parameter estimation is a challenging optimization problem on a mixed, continuous and discrete domain. Section 4.3 described multiple shooting for the continuous optimization domain. This section describes a modified algorithm that converges in a continuous and discrete domain.

Section 4.4.1 describes a time-grid alignment that simplifies the discrete search by aligning rigid-body collisions with time-grid points t^G. In the standard multiple-shooting approach, the time-grid is fixed throughout optimization. After alignment with collisions the standard approach would also require fixed collision timing. Instead, MOTIONSKETCHER generalizes the standard multiple-shooting approach with a sliding time grid. Section 4.4.2 describes an approach that allows MOTIONSKETCHER to optimize the collision timing by adjusting the time grid.

4.4.1 Time-Grid Alignment

When the time grid is aligned with collisions, each shooting interval corresponds to a free-flight motion. Impacts are instantaneous events that occur between shooting intervals. Each initial state $\mathbf{q}_i$ specifies the generalized state of the system an instant after the impact; thus, the free-flight motion during each interval is continuous with respect to its initial state.

The grid alignment allows the animator to specify directly desired collisions, positions, orientations, and velocities of bodies. For example, initial states $\mathbf{q}_0, \ldots, \mathbf{q}_{m-1}$ directly encode intended collisions as each initial state, except for $\mathbf{q}_0$, maps to a collision. Each set of initial states identifies a smooth component in the simulation function. A simple discrete search could search through smooth components by selecting the number of initial states and sampling their values. For example, a motion with a single collision consists of two initial states: initial vector $\mathbf{q}_0$ at time t_0 and a state $\mathbf{q}_1$ at the time of collision. Sampling the initial state $\mathbf{q}_0$ and $\mathbf{q}_1$ and solving the appropriate parameter estimation problem would enumerate the smooth components of the simulation function that correspond to all motions with a single collision.

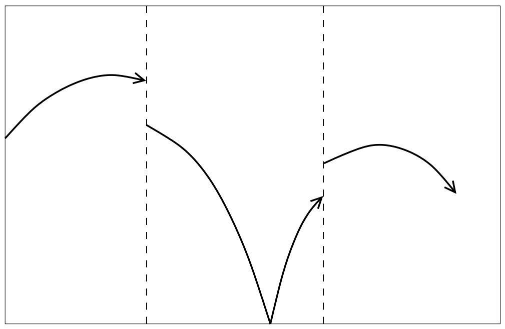


FIGURE 4.5. An unaligned time grid subdivides the motion arbitrarily. The motion in the second interval may not be continuous with respect to its initial state; the collision type may change at impact.

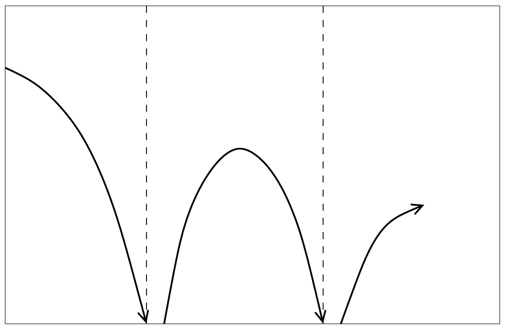


FIGURE 4.6. An aligned time grid subdivides the motion into free-flight segments. Subdivisions begin and end with each collision. The motion during each interval is continuous with respect to its initial state.

4.4.2 Sliding Time Grid

In the standard multiple-shooting approach, the time grid $t_0^G, \ldots, t_m^G$ remains fixed throughout the optimization. Because MOTIONSKETCHER aligns the time grid with collisions, this approach restricts the optimization by enforcing predetermined collision times. Instead, MOTIONSKETCHER generalizes the standard multiple shooting to allow a sliding time grid:

$$0 = t_0^G < t_1^G(\mathbf{q}_0) < \cdots < t_{m-1}^G(\mathbf{q}_{m-2}) < t_m^G = T. \tag{4.6}$$

The interior grid points $t_1^G(\mathbf{q}_0), \ldots, t_{m-1}^G(\mathbf{q}_{m-2})$ are functions of the generalized state at the time of the previous collision. For rigid bodies with smooth shapes, the functions are smooth, enabling efficient optimization with parameter estimation. In the general case the functions are only piecewise smooth, nevertheless the parameter solver converges because in practice discontinuities are not close to an optimum [Gill 89]. MOTIONSKETCHER uses a two-step optimization process.

The first optimization step discovers the correct smooth component—a region of the control space in which the simulation function is continuously differentiable—by identifying the types of collision in the sketch. As outlined in Section 4.4.1, a smooth component is identified by initial states $\mathbf{q}_0, \ldots, \mathbf{q}_{m-1}$. The first step computes the parameters that yield the motion closest to the sketch. As a result, MOTIONSKETCHER discovers the smooth

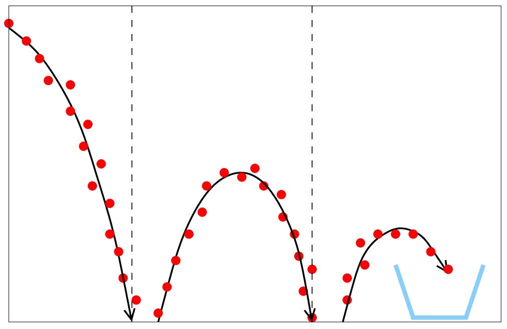


FIGURE 4.7. The first optimization step uses a fixed time grid to fit a physical motion within each segment to sketch samples (illustrated with dots). The aim of this step is to identify collision types.

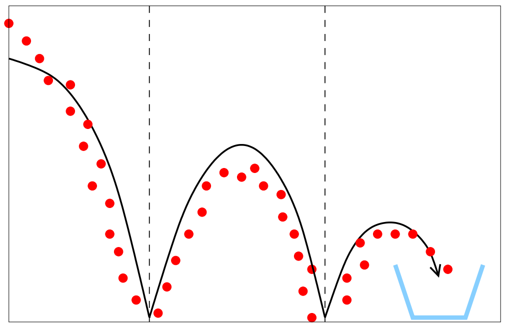


FIGURE 4.8. The second optimization step uses a sliding time grid to fit sketch samples (illustrated with dots), to eliminate inter-penetration, to adjust collision times, and to enforce physical behavior at each collision.

component that best corresponds to the sketched motion.

The first step uses a fixed time grid to solve the relaxed parameter estimation problem in Equation 4.2 without the continuity constraints $\mathbf{c}_c$:

$$\begin{array}{ll} \min\limits_{\mathbf{u},\mathbf{q}_0,\dots,\mathbf{q}_{m-1}} & \sum_{i=1}^{n} \left\| \mathbf{p}_i\Big(\mathcal{S}_{t_i}(\mathbf{u},\mathbf{q}_0,\dots,\mathbf{q}_{m-1})\Big) - \mathbf{s}_i \right\|^2 \\ \text{subject to} & \underline{\mathbf{b}} \le \mathbf{c}_s\big(\mathbf{u},\mathbf{q}(t_1),\dots,\mathbf{q}(t_n)\big) \le \overline{\mathbf{b}}. \end{array}$$

Collisions are ignored during the optimization, allowing inter-penetration between the bodies. The result is locally physically correct motion that within each segment obeys the equations of motion (Figure 4.7). Most important, the computed initial states $\mathbf{q}_0,\dots,\mathbf{q}_{m-1}$ identify the desired smooth component of the simulation function.

The second optimization step uses the sliding time grid to solve the full parameter estimation problem in Equation 4.2. The resulting motion eliminates inter-penetration, adjusts collision times, and fits the sketch while complying with the motion model (Figure 4.8). In contrast to the fixed time grid, the sliding time grid introduces two technical issues. First, as the optimization adjusts interior grid times, some sketch samples $\mathbf{s}_i$ will shift between shooting segments. These discrete events introduce small discontinuities into the least-squares objective function and sketch constraints $\mathbf{c}_s$. In practice, these discontinuities are removed by ignoring the samples near each collision. Because the sketches are sampled densely ignoring few samples does not disregard main aspects of the sketch. Second, the sliding time grid may violate the total ordering in Equation 4.6. This violation is a strong indication that, for the given collision sequence and collision types, the motion model cannot express a sketched motion. For example, a physical motion model cannot express physically infeasible motion. Possible solutions are to relax the equations that model the physics or to propose different collision configurations, either automatically in a heuristic search or interactively by the animator.

4.5 Sketching Interfaces

Sketching interfaces are animator tools for rapid design of rigid-body animations. The sketch procedure is intuitive, allowing the animator to express the key aspects of the desired motion and to encode her intuition for constructing the solution. A sketch interface encapsulates both the user interface, which abstracts the details of the parameter solver, and the optimization engine, which transforms the sketch into a canonical form. The canonical form serves as the initial approximation for the parameter estimation problem. Constructing the initial approximation is the key component of the interfaces because parameter estimation is a difficult nonlinear and nonconvex optimization problem with many locally optimal solutions.

This section describes two of many possible interfaces and evaluates each interface with several motion-design problems. Section 4.5.1 describes the editing interface, which relies on MotionEditor to design each free-flight segment of the sketch. Section 4.5.2 describes the acting interface, which captures animator's hand gestures as performed sketches of the desired motion.

4.5.1 Editing Interface

The editing interface generalizes the MotionEditor approach to editing rigid body simulations. With MotionEditor, the animator designs the *entire* motion by dragging a body, at any point in time, to a desired location. MotionEditor computes the required simulation parameters and simulates the corresponding motion. In contrast, the editing interface divides rigid-body motion into independent segments: the animator designs the motion of each free-flight segment independent of the prior segments. The animator adds and removes free-flight segments as desired. As described in Section 4.3, the segmentation improves the stability of the method because the simulation need only integrate over small integration intervals and need not compute the gradients of motion through collisions. As a result, the editing interface allows the animator to sketch complex motions quickly and robustly.

With the editing interface, sketched trajectories may contain gaps—discontinuities in position, orientation, or velocities—between each free-flight segment. In contrast to MOTIONEDITOR, the interface modifies only the simulation parameters affecting the single free-flight segment. For example, if the animator adjusts the free-flight trajectories during the second segment, the segment between the first and second collision, then the interface computes new values for initial state $\mathbf{q}_1$ only. Initial states $\mathbf{q}_0, \mathbf{q}_2, \ldots, \mathbf{q}_{m-1}$ are not adjusted as they are either before or after the current free-flight segment. The parameter estimation procedure removes these gaps by enforcing the continuity constraints.

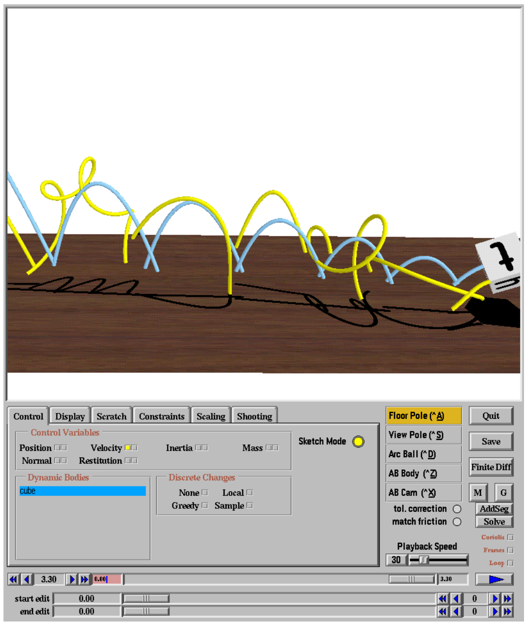


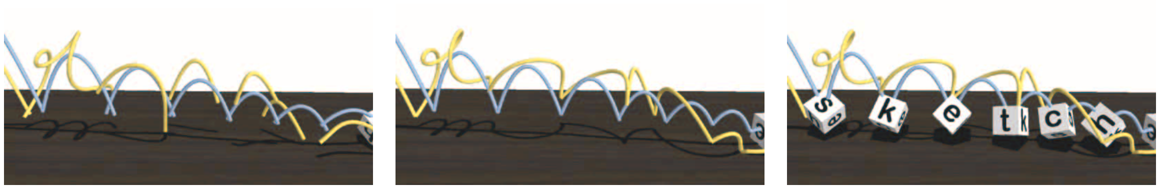
FIGURE 4.9. Snapshot of the editing interface. The animator designs the sketch by editing each free-flight segment independently.

Sketches designed with the editing interface consist of a set of free-flight segments. Each segment corresponds to an initial state $\mathbf{q}_i$. As the animator constructs the sketch by editing a free-flight segment, the interface adjust the initial states appropriately. In effect, the interface provides the animator with direct intuitive control of the initial approximation.

For the editing interface, the construction of the initial approximation from the sketch is straightforward. When the animator adds a segment, the interface creates the appropriate initial state; when the segment is removed, the interface deletes the initial state. Once the sketch is complete, the resulting initial states describe the initial approximation.

The die example illustrates an over-constrained motion-design problem, which MOTIONEDITOR alone cannot solve. The die must spell out a word, bounce in line with proper spacing, and have the correct orientation at each impact (Figure 4.10). With the editing interface, the animator sketches the motion in pieces, each free-flight segment is sketched independently. The resulting sketch describes the desired trajectories and encodes the sequence of spelled letters.

The sketch has gaps at each collision because continuity between free-flight segments is not enforced during sketch design. Figure 4.10 shows the gaps in position and orientation. Gaps in velocities are also present but are not illustrated in the figure. The parameter estimation procedure adjusts the initial position and velocity of the die, as well as surface normals at impacts to reduce the gaps as much as possible. As a result, MOTIONSKETCHER computes a physically plausible motion that achieves the desired goals, with imperceptible gaps in position and orientation, and small gaps in linear and angular velocities at collisions.



no. of constraints	no. of sim. parameters	optimization time (MIPS R10000)
36	30	182.45 seconds

FIGURE 4.10. In the left figure, sketched trajectories of the center of mass (blue curve) and the die corner (yellow curve) are discontinuous. This illustrates gaps in position and orientation at each collision. Gaps in velocities also exist but are not explicitly illustrated. In the middle figure, the same trajectories after parameter estimation are continuous. The parameter solver removes the gaps in position and orientation. Gaps in velocities are also reduced. The right figure displays the final motion.

4.5.2 Acting Interface

With the acting interface, the animator performs the motion by holding and moving instrumented real-world objects. The resulting sensor data encodes motion trajectories for each object. The trajectories are imprecise but sufficient to convey the essence of the motion. The timing may be arbitrary: slower or faster than real-time. The parameter estimation procedure transforms the sketch into a physical motion with physically correct timing.

The acting interface converts sketched trajectories into an initial approximation of the animation. The initial approximation is defined by initial state $\mathbf{q}_0, \ldots, \mathbf{q}_{m-1}$ at the beginning of each free-flight segment. The acting interface recovers the initial states from the sketch by solving a sequence of simplified parameter estimation problems.

The optimization transforms each free-flight segment independently. The animator identifies free-flight segments by recording the indices $l_1, \ldots, l_{m-1}$ of sketch samples near each collision. Because collisions denote the end and the beginning of a free-flight segment, the interface represents m free-flight segments with $m + 1$ indices:

$$1 = l_0 < l_1 < \ldots < l_{m-1} < l_m = n.$$

Recalling that there are n sketch samples. Indices $l_1, \ldots, l_{m-1}$ mark sketch samples for all sketch collisions. For example, samples $\{\mathbf{s}_{l_1}, \ldots, \mathbf{s}_{l_2} - 1\}$ discretize sketch trajectories of the second free-flight segment, between the first and second collision.

A new optimization procedure simultaneously computes the timing of each sketch sample within a free-flight segment, and the center-of-mass trajectories for all bodies. The center-of-mass trajectories within each free-flight segment are computed as functions $\mathbf{p}^{\mathbf{q}}(t, \mathbf{q}_i)$ of the appropriate initial state. The interface optimizes the least-squares fit to sketch samples within each free-flight segment:

$$\begin{aligned} \min_{\mathbf{q}_i,\, t_{l_i}, \ldots, t_{l_{i+1}-1}} \quad & \sum_{j=l_i}^{l_{i+1}-1} \|\mathbf{p}^{\mathbf{q}}(t_j, \mathbf{q}_i) - \mathbf{p}^{\mathbf{s}}(\mathbf{s}_j)\|^2 \\ \text{subject to} \quad & 0 \le t_j < t_{j+1}, \quad \forall j \in [l_i, l_{i+1} - 1], \end{aligned}$$

where $\mathbf{p}^{\mathbf{s}}$ is the projection operator that extracts the center-of-mass coordinates from a sketch sample. The optimization constraint ensures that the total order of sketch samples is preserved: the succeeding sample must be at a later time than its predecessor.[3] Note that the function $\mathbf{p}^{\mathbf{q}}(t, \mathbf{q}_i)$ is considerably simpler than the simulation function $\mathcal{S}(t, \mathbf{q}_i)$. If the gravitational force is aligned with a second coordinate axis, the center-of-mass trajectory of a single 3-D rigid body in free flight is described by a quadratic function and two linear functions:

$$\mathbf{p}^{\mathbf{q}}(t, \mathbf{q}_i) \overset{\text{def}}{=} \mathbf{x}(\mathbf{q}_i) + \mathbf{v}(\mathbf{q}_i)t + \begin{pmatrix} 0 \\ -\frac{1}{2}gt^2 \\ 0 \end{pmatrix},$$

where the projection functions $\mathbf{x}(\cdot)$ and $\mathbf{v}(\cdot)$ extract the position and the linear velocity of the center of mass.

After m optimizations, one for each free-flight segment, the interface computes m initial states $\mathbf{q}_0, \ldots, \mathbf{q}_{m-1}$. Although these initial states define an initial approximation, the approximation can be improved by matching the orientation of samples in the sketch. The first of two optimization steps described in Section 4.4.2 performs this task by solving a relaxed parameter-estimation problem:

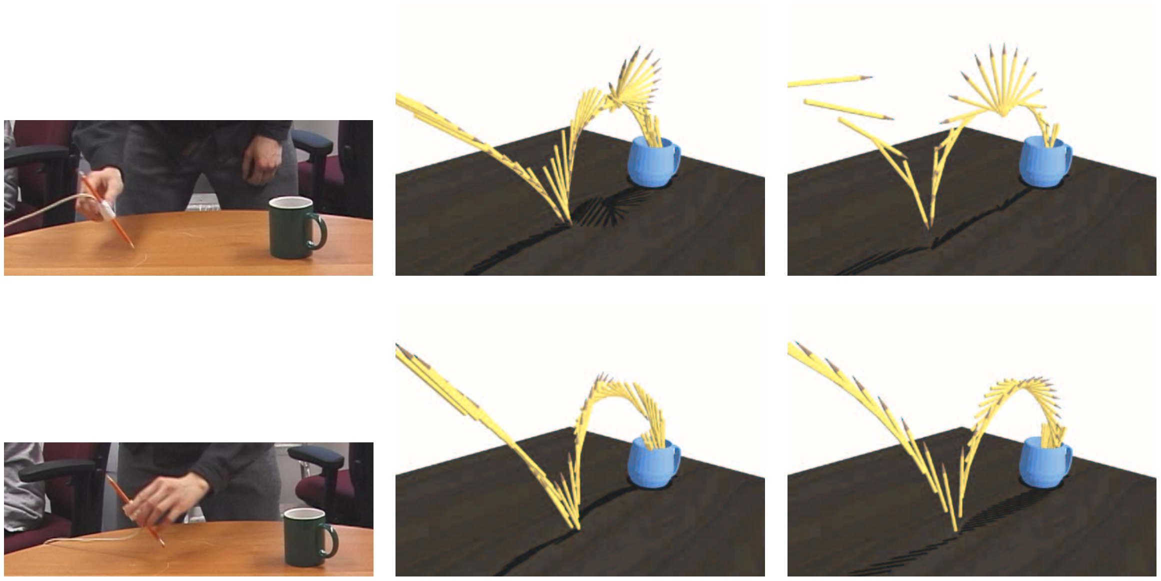
$$\begin{array}{ll} \displaystyle\min_{\mathbf{u}, \mathbf{q}_0, \cdots, \mathbf{q}_{m-1}} & \sum_{i=1}^{n} \left\| \mathbf{p}_i\Big(\mathcal{S}_{t_i}(\mathbf{u}, \mathbf{q}_0, \ldots, \mathbf{q}_{m-1})\Big) - \mathbf{s}_i \right\|^2 \\ \text{subject to} & \underline{\mathbf{b}} \le \mathbf{c}_s\big(\mathbf{u}, \mathbf{q}(t_1), \ldots, \mathbf{q}(t_n)\big) \le \overline{\mathbf{b}}. \end{array}$$

In the first example of motion design, the animator performs an animation of a pencil bouncing off a table and landing in a mug. This performance sketches the desired motion, by roughly describing desired trajectories. The animator also adds an explicit position constraint enforcing the pencil landing within the mug. The constraints are specified with the MOTIONSKETCHER user interface.

The annotated sketch sample near the collision with the table defines the two free-flight segments. The interface constructs the initial approximation, and the parameter estimation procedure refines the approximation to produce a physical motion that matches the sketch.

Figure 4.11 shows two physical animations with subtle variations in style. In the first animation, the pencil tumbles in the air after colliding, tip-first, with the table. In the

[3] The implementation reformulates the problem slightly to solve for inter-sample durations d_i instead of explicit times t_j. With this reformulation $t_j = \sum_i^j d_i$, and the linear constraints $0 \le t_j < t_{j+1}$ are simplified with constant bound constraints $0 < d_i$.



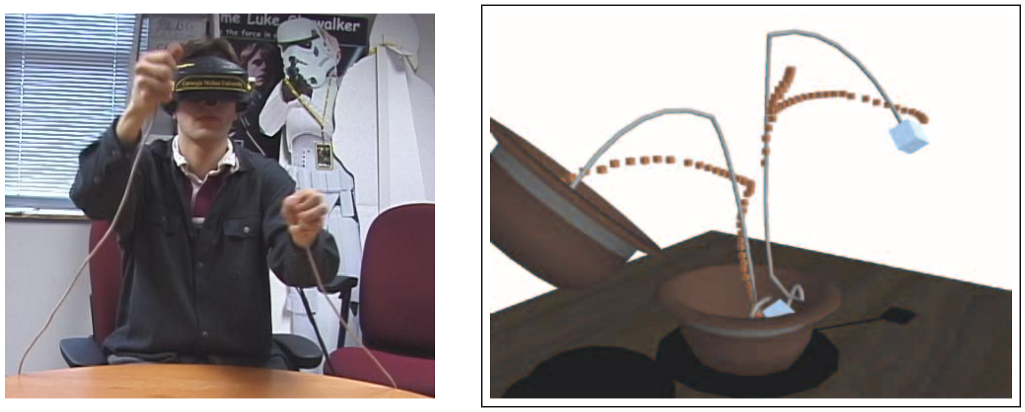
no. of constraints	no. of sim. parameters	avg. optimization time (MIPS R10000)
6	12	102.12 seconds

FIGURE 4.11. MOTIONSKETCHER converts a performed sketch (left and middle) into a physical motion (right). The two animations capture the subtle variations of performed sketches.

second, the pencil bounces with the eraser side and lands in the mug after only a half-tumble.

The second motion-design problem involves a sketch with two objects, a hat and a box that collide in mid-air. After the collision, the box should land inside the hat. Rather than use real-world objects, the animator employs a head-mounted display and a virtual world to view the motion of objects during the sketching performance. The motion itself is specified by gesturing with hand-held motion sensors. The animator enforces two position constraints: an inequality position constraint that enforces the mid-air collision and an equality position constraint that forces the box to land within the hat.

The parameter solver computes initial positions and velocities of both objects. In addition, the solver adjusts the surface normal at the impact and the elasticity of the objects. Figure 4.12 shows the resulting motion. In this example, the motion is physically plausible but not physically correct, due to the apparent infeasibility of the two constraints—in order for the box to land in the hat, the collision must cancel the opposite linear momentum of the



no. of constraints	no. of sim. parameters	optimization time (MIPS R10000)
15	29	300.35 seconds

FIGURE 4.12. An illustration of the physically plausible motion derived from a sketched motion. The computed center-of-mass trajectories for the hat and the box are shown in gray lines. The sketch samples are indicated with red dots.

two bodies. This is not possible without adjusting the angular momentum of the two bodies, which would, in turn, require violating the orientations specified in the sketch. Instead, the solver minimizes physical violations at the impact to construct a plausible motion. Note that the mid-air collision in the solution has been automatically raised to a higher point than was specified in the sketch. Raising the collision point has the effect of decreasing the velocities of the two objects when they collide, making it easier to satisfy the constraints.

Chapter 5
CONCLUSION AND FUTURE WORK

Using the semi-automatic techniques described in this dissertation, an animator can rapidly design complicated physical animations that would be difficult to create with keyframing techniques or rigid-body simulators. MOTIONEDITOR implements an interactive editing paradigm; instead of changing the simulation parameters, the animator can directly adjust positions and velocities of simulated bodies. MOTIONSKETCHER implements an off-line sketching paradigm; the animator sketches the motion and MOTIONSKETCHER generates the physical motion that best matches the sketch.

Without help from an animator, these motion-design tools are not effective. Neither MOTIONEDITOR nor MOTIONSKETCHER will converge to the desired motion unless the animator identifies the appropriate smooth component[1] through interaction or sketching. Discontinuities in the simulation function are the key obstacles to unguided convergence. Both tools would be improved with an automatic technique that searches for the appropriate smooth component. Section 5.1 discusses possible approaches.

Once a smooth component is identified, the design tools converge efficiently because both techniques rely on derivative information: the Jacobians of motion and constraints. The techniques compute the Jacobians analytically for all expressions except integral expressions, which require numeric integration. Analytic differentiation is more efficient and accurate than a finite difference method, but it prevents the current tools from designing motions with complicated collision and contact behavior: multiple-point frictional collision, sliding and static contact. The implemented design tools assume that mathematical models of physical behaviors are analytically differentiable. For some contacts and collisions, however, differentiable models may not exist. Section 5.2 discusses techniques for removing this limitation.

MOTIONSKETCHER is a general framework independent of the sketching interface. This dissertation explores two possible sketching interfaces: the acting and editing inter-

[1]Smooth components are defined and discussed in Section 2.1.1.

faces. Section 5.3 discusses other intuitive interfaces that could make animation a more accessible form of expression for non-professional animators.

All examples demonstrate motion design for passive and unconstrained rigid bodies. The motion of more complicated objects with joints and hinges, or constrained rigid bodies, cannot be designed with existing implementations. Although the described techniques should generalize to constrained rigid bodies and bodies with self-propelling forces, it is not clear whether the resulting implementations will remain practical. Section 5.4 discusses these issues in more detail.

5.1 Automated Discrete Search

MOTIONEDITOR cannot compute the desired motion without interaction. In the editing approach, the animator guides MOTIONEDITOR towards the motion. As described in Section 3.5.4, this guidance is critical for convergence in the continuous and discrete control space. When a simulation-function discontinuity forces MOTIONEDITOR away from the desired motion, the animator must correct the problem. In the hat example, shown in Figure 3.4, MOTIONEDITOR can enforce the landing constraint because the hat remains in free-flight throughout the adjustments. In contrast, if the animator wanted to introduce a bounce, as in the scissors example shown in Figure 3.8, MOTIONEDITOR could not maintain both the hooking constraint and the transition from free-flight to bouncing motion. Instead, the animator must introduce a bounce before hooking the scissors.

Similarly, without a good initial approximation, the MOTIONSKETCHER parameter estimation will not converge to a physically plausible motion. In the sketching approach, the animator creates a sketch that serves as the initial approximation to the desired motion. Although the sketch need not describe the desired motion precisely, it must yield a good initial approximation of the desired motion. In particular, the sketch must identify the correct collision sequence for the desired motion.

An automatic discrete search may alleviate these problems. Continuous optimization converges effectively once the appropriate smooth component is identified. A smooth component corresponds to a specific type of motion: free flight, one bounce, two bounce, and other motions. Note that two single-bounce motions correspond to the same smooth component only if the collision occurs between identical bodies and identical polygonal primitives (i.e., between the same vertex and the same face). Finding the correct smooth component requires searching through the discrete space to find the colliding primitives, colliding bodies, and the number of collisions. Changing the number of collisions, colliding bodies, or colliding primitives changes the smooth component.

A search could enumerate all the smooth components in the discrete space. Section 4.4.1 describes how this enumeration can be accomplished within the MOTIONSKETCHER framework. Because the discrete space grows exponentially with the number of bounces, the simple approach would not scale for many practical problems. Interaction could help prune the search space. For example, the animator could specify the number of bounces or identify the colliding bodies.

A heuristic sampling approach could be more effective for many practical problems. For example, a randomized search based on the Markov chain Monte Carlo method is effective for designing animations with complex interactions and many colliding bodies [Chenney 00]. Monte Carlo methods randomly sample the parameter space. The sequence of samples asymptotically converges to the simulation parameters that yield the desired animation. For quick convergence, the Monte Carlo approach requires a handcrafted proposal mechanism that generates a new sample in the sequence. For example, in the ideal case a proposal mechanism would only generate samples that yield the desired animations.

In current Monte Carlo methods, the animator designs a proposal mechanism that samples the simulation parameter space [Chenney 00]. A better approach would allow the animator to design a proposal mechanism that samples the state space directly. The animator can identify the desired motion in the state space by describing the desired states of the rigid bodies at any point in time or by identifying desired collision configuration. For example, in the state space the animator could directly express her intent to create a motion with only three bounces. Describing this intent by setting the simulation parameters would be difficult.

5.2 Collision Models

The techniques described in this dissertation rely on the assumption that the mathematical models of collision are analytically differentiable. This assumption does not always hold for the rigid-body simulator employed by MOTIONEDITOR and MOTIONSKETCHER. During a resting (i.e. sustained) contact or for multiple-point collisions the applied impulses are solutions to linear complementarity problems [Baraff 94]. In general, linear complementarity problems do not have closed-form, analytically differentiable solutions. As a result, the current implementations of MOTIONEDITOR and MOTIONSKETCHER cannot design motions with complex collision behaviors.

The Jacobian of an arbitrary collision model could be computed with a finite difference approximation. As described in Section 3.2, the finite-difference technique is inaccurate and inefficient for evaluating the integral expressions defining the Jacobian of free-flight motion. The collision models of instantaneous impact, however, do not require numeric integration, and the finite difference technique may yield sufficiently accurate derivatives of the collision model.

Alternative collision models may permit analytic differentiation. Simple and efficient models may result in better design tools because they could be easier to control. Developing such models is challenging, however, because the models must remain physically meaningful. Most research in simulation of physical processes develops sophisticated mathematical models that emphasize greater physical accuracy. If the primary concern is physical accuracy then controllable collision models may not exist. Statistical analysis of physical processes, derived either from current state-of-the-art simulation methods or real-world observations, may lead to simpler mathematical models that are physically plausible instead of physically accurate. Ideally, a simulation framework should balance the physical accuracy of a simulation method with its simplicity and efficiency.

5.3 Sketching Interfaces

Sketching interfaces are an appealing paradigm for specifying a desired motion. MOTIONSKETCHER supports the editing and acting interface but other interfaces are possible.

The acting interface described in Section 4.5.2 requires motion sensors that detect their positions and orientations in 3-D. The animator holds the motion sensors or attaches them to real-world objects. By moving the motion sensors, the animator acts out the desired motion. A camera-based interface could make the acting process less intrusive and even more natural. For example, the animator might simply record her hand gestures with a video camera. For this interface, a tracking procedure must extract the hand motion from the video. If this extraction can be done robustly then parameter-estimation techniques might be able to resolve the ambiguities of fitting a 3-D physical motion to the 2-D recorded gestures.

A drawing interface could allow the animator to sketch the desired body trajectories. A paper and pencil drawing might be sufficient to describe a simple motion. Multiple sketches from different viewpoints may be necessary to describe a more complicated motion; redundant sketches can resolve the ambiguities of a single 2-D sketch.

5.4 Constrained and Active Rigid Bodies

The equations of motion of rigid bodies constrained by links and joints can be derived by two different methods: a method of reduced coordinates and a method of maximal coordinates. The reduced-coordinate method, also known as the generalized-coordinate method, describes the state of the constrained rigid bodies with the minimal number of coordinates. This reduced set of coordinates has the same number of parameters as the constrained system has degrees of freedom. The maximal-coordinate method, also known as the Lagrange-multiplier method, does not reduce the number of coordinates to match the number of degrees of freedom. Instead, the Lagrange-multiplier method enforces the constraints explicitly. The benefits and drawbacks of the reduced-coordinate method and the maximal-coordinate method are documented [Baraff 96].

MotionEditor and MotionSketcher use the generalized-coordinate method to describe and design the motion of rigid bodies, as described in Section 2.1.3. Because the generalized state expresses only configurations that satisfy the constraints, joint and link constraints are enforced automatically. The physical simulator employed by MotionEditor and MotionSketcher, however, uses the Lagrange-multiplier method to compute the motion of constrained rigid bodies. Because the simulator enforces the constraints numerically, numerical drift may cause small constraint violations during the simulation. Although constraint violations are unnoticeable in an animation, they reduce the accuracy of the Jacobian computation described in Section 3.2, preventing MotionEditor and MotionSketcher from converging. As a result, the current implementations of

MotionEditor and MotionSketcher cannot design the motion of rigid bodies constrained by joints or linkages.

The maximal-coordinate method for simulation could still be used. The drift incurred by this method can be prevented with sophisticated techniques for solving differential-algebraic equations [von Schwerin 99]. Alternatively, the simulator could use a reduced-coordinate method to compute the motion of constrained rigid bodies. In this case, new implementations of MotionEditor and MotionSketcher must efficiently compute the Jacobians of more complicated equations of motion. Further work is required to determine which of these two approaches is more practical.

References

Vladimir Igorevich Arnold. *Mathematical Methods of Classical Mechanics*. Springer-Verlag, New York, second edition, 1989. (pp. 5, 8)

Uri M. Ascher, Robert M. M. Mattheij, and Robert D. Rusell. *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988. (pp. 13, 47)

David Baraff. *Dynamic Simulation of Non-Penetrating Rigid Bodies*. PhD thesis, Cornell University, March 1992. (pp. 5, 23, 33)

David Baraff. *Fast Contact Force Computation for Nonpenetrating Rigid Bodies*. In Computer Graphics (Proceedings of SIGGRAPH 94), Annual Conference Series, pages 23–34. ACM SIGGRAPH, July 1994. (pp. 23, 68)

David Baraff. *Linear-time dynamics using Lagrange multipliers*. In Computer Graphics (Proceedings of SIGGRAPH 96, Annual Conference Series, pages 137–146. ACM SIGGRAPH, August 1996. (p. 69)

David Baraff and Andrew Witkin. *Large Steps in Cloth Simulation*. In Computer Graphics (Proceedings of SIGGRAPH 98), Annual Conference Series, pages 43–54. ACM SIGGRAPH, July 1998. (p. 1)

Ronen Barzel and Alan H. Barr. *A Modeling System Based On Dynamic Constraints*. In Computer Graphics (Proceedings of SIGGRAPH 88), Annual Conference Series, pages 179–188. ACM SIGGRAPH, August 1988. (pp. 16, 17)

Ronen Barzel. *Physically-Based Modeling for Computer Graphics: A Structured Approach*. Academic Press, San Diego, CA, 1992. (p. 5)

Ronen Barzel, John F. Hughes, and Daniel N. Wood. *Plausible Motion Simulation for Computer Graphics Animation*. In Computer Animation and Simulation '96, Proceedings of the Eurographics Workshop, pages 184–197, Poitiers, France, September 1996. (pp. 8, 18)

Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*, volume I. Athena Scientific, Belmont, Massachusetts, 1995. (pp. 14, 16)

Dimitri P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, Massachusetts, 1995. (pp. 15, 30, 35, 36)

Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, Massachusetts, 1996. (p. 16)

Paul J. Besl and Neil D. McKay. *A Method for Registration of 3D Shapes*. IEEE Transactions on Pattern Analysis and Machine Intelligence, 14(2):239–256, 1992. (p. 53)

Hans Georg Bock. *Numerical Treatment of Inverse Problems in Chemical Reaction Kinetics*. In K. H. Ebert, P. Deuflhard, and W. Jäger, editors, *Modelling of Chemical Reaction Systems (Proceedings of an International Workshop, Heidelberg)*, pages 102–125. Springer-Verlag, September 1980. (pp. 19, 48)

Hans Georg Bock. *Recent Advances in Parameter Identification Techniques for Ordinary Differential Equations*. In P. Deuflhard and E. Hairer, editors, *Numerical Treatment of Inverse Problems in Differential and Integral Equations (Proceedings of an International Workshop, Heidelberg)*, pages 95–121. Birkhäuser, Boston, 1983. (pp. 19, 25, 48)

John Canny and John Reif. *New Lower Bound Techniques for Robot Motion Planning Problems*. In 28th Annual Symposium on the Foundations of Computer Science, pages 49–60, Los Angeles, California, October 1987. IEEE. (pp. 2, 17)

John Canny, Bruce Donald, John Reif, and Patrick Xavier. *On the Complexity of Kinodynamic Planning*. In 29th Annual Symposium on the Foundations of Computer Science, pages 305–316, White Plains, New York, October 1988. IEEE. (pp. 2, 17)

Stephen Chenney and D. A. Forsyth. *Sampling Plausible Solutions to Multi-body Constraint Problems*. In Computer Graphics (Proceedings of SIGGRAPH 2000), Annual Conference Series, pages 219–228. ACM SIGGRAPH, July 2000. (pp. 8, 18, 67)

Michael F. Cohen. *Interactive Spacetime Control for Animation*. In Computer Graphics (Proceedings of SIGGRAPH 92), Annual Conference Series, pages 293–302. ACM SIGGRAPH, July 1992. (p. 15)

Bruce Donald, Patrick Xavier, John Canny, and John Reif. *Kinodynamic Motion Planning*. Journal of the ACM, 40(5):1048–1066, November 1993. (p. 17)

Nick Foster and Dimitri Metaxas. *Realistic Animation of Liquids*. Graphical Models and Image Processing, 5(58):471–483, 1996. (p. 1)

Michael Garland and Paul S. Heckbert. *Surface Simplification Using Quadric Error Metrics*. In Computer Graphics (Proceedings of SIGGRAPH 97), Annual Conference Series, pages 209–216. ACM SIGGRAPH, August 1997. (p. 34)

Philip E Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*. Academic Press, London, 1989. (pp. 15, 32, 49, 55)

Philip E. Gill, Walter Murray, and Michael A. Saunders. *User's Guide for SNOPT 5.3: A Fortran Package for Large-Scale Nonlinear Programming*. Technical Report NA 97–5, University of California, San Diego, 1997. (p. 49)

Michael Gleicher and Andrew Witkin. *Differential Manipulation*. In Graphics Interface, pages 61–67, June 1991. (p. 21)

Michael Gleicher and Andrew Witkin. *Through-the-Lens Camera Control*. In Computer Graphics (Proceedings of SIGGRAPH 92), Annual Conference Series, pages 331–340. ACM SIGGRAPH, July 1992. (p. 21)

Michael Gleicher. *Motion Editing with Spacetime Constraints*. In 1997 Symposium on Interactive 3D Graphics, pages 139–148, April 1997. (p. 15)

Michael Gleicher. *Retargetting Motion to New Characters*. In Computer Graphics (Proceedings of SIGGRAPH 98), Annual Conference Series, pages 33–42. ACM SIGGRAPH, August 1998. (p. 15)

Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, Maryland, third edition, 1996. (p. 32)

Andreas Griewank and George Corliss, editors. *Automatic Differentiation of Algorithms*. SIAM, Philadelphia, 1991. (p. 25)

Radek Grzeszczuk and Demetri Terzopoulos. *Automated Learning of Muscle-Actuated Locomotion Through Control Abstraction*. In Computer Graphics (Proceedings of SIGGRAPH 95), Annual Conference Series, pages 63–70. ACM SIGGRAPH, August 1995. (p. 17)

Mikako Harada, Andrew Witkin, and David Baraff. *Interactive Physically-Based Manipulation of Discrete/Continuous Models*. In Computer Graphics (Proceedings of SIGGRAPH 95), Annual Conference Series, pages 199–208. ACM SIGGRAPH, August 1995. (pp. 21, 22)

Jessica K. Hodgins, Wayne L. Wooten, David C. Brogan, and James F. O'Brien. *Animating Human Athletics*. In Computer Graphics (Proceedings of SIGGRAPH 95), Annual Conference Series, pages 71–78. ACM SIGGRAPH, August 1995. (p. 17)

Paul M. Isaacs and Michael F. Cohen. *Controlling Dynamic Simulation with Kinematic Constraints, Behavior Functions and Iinverse Dynamics*. In Computer Graphics (Proceedings of SIGGRAPH 87), Annual Conference Series, pages 215–224. ACM SIGGRAPH, July 1987. (p. 16)

Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. *Reinforcement Learning: A Survey*. Journal of Artificial Intelligence Research, 4:237–285, 1996. (p. 16)

Wilfred Kaplan. *Advanced Calculus*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1984. (p. 26)

Michael Kass and Gavin Miller. *Rapid, Stable Fluid Dynamics for Computer Graphics*. In Computer Graphics (Proceedings of SIGGRAPH 90), Annual Conference Series, pages 49–57. ACM SIGGRAPH, August 1990. (p. 1)

Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, Massachusetts, 1991. (p. 17)

Steven M. LaValle and James J. Kuffner. *Randomized Kinodynamic Planning*. In Proceedings 1999 IEEE International Conference on Robotics and Automation, 1999. (p. 17)

Zicheng Liu, Steven J. Gortler, and Michael F. Cohen. *Hierarchical Spacetime Control*. In Computer Graphics (Proceedings of SIGGRAPH 94), Annual Conference Series, pages 35–42. ACM SIGGRAPH, July 1994. (p. 15)

Brian Mirtich. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD thesis, University of California, Berkeley, December 1996. (p. 5)

Matthew Moore and Jane Wilhelms. *Collision Detection and Response for Computer Animation*. In Computer Graphics (Proceedings of SIGGRAPH 88), Annual Conference Series, pages 289–298. ACM SIGGRAPH, August 1988. (pp. 1, 9)

Rémi Munos and Andrew W. Moore. *Varible resolution discretization for high-accuracy solutions of optimal control problems*. In International Joint Conference onf Artificial Intelligence, 1999. (p. 16)

Richard M. Murray, Zexiang Li, and S. Shankar Sastry. *A Mathematical Introduction to Robotic Manipulation*. CRC Press, Boca Raton, 1994. (p. 9)

James F. O'Brien and Jessica K. Hodgins. *Graphical Modeling and Animation of Brittle Fracture*. In Computer Graphics (Proceedings of SIGGRAPH 99), Annual Conference Series, pages 111–120. ACM SIGGRAPH, August 1999. (p. 1)

L. S. Pontryagin, V. G. Boltyansky, R. V. Gamkrelidze, and E. F. Mischenko. *Mathematical Theory of Opimal Processes*. Interscience Publishers, New York, 1962. (p. 14)

Jovan Popović, Steven M. Seitz, Michael Erdmann, Zoran Popović, and Andrew Witkin. *Interactive Manipulation of Rigid Body Simulations*. In Computer Graphics (Proceedings of SIGGRAPH 2000), Annual Conference Series, pages 209–218. ACM SIGGRAPH, July 2000. (p. 22)

Zoran Popović and Andrew Witkin. *Physically Based Motion Transformation*. In Computer Graphics (Proceedings of SIGGRAPH 99), Annual Conference Series, pages 11–20. ACM SIGGRAPH, August 1999. (p. 15)

Marc H. Raibert and Jessica K. Hodgins. *Animation of Dynamic Legged Locomotion*. In Computer Graphics (Proceedings of SIGGRAPH 91), Annual Conference Series, pages 349–358. ACM SIGGRAPH, July 1991. (p. 17)

Barbara Robertson. *Medieval Magic*. Computer Graphics World, April 2001. (p. 1)

Barbara Robertson. *Meet Geri: The New Face of Animation*. Computer Graphics World, February 1998. (p. 1)

Barbara Robertson. *Antz-piration*. Computer Graphics World, October 1999. (p. 1)

Jos Stam. *Stable Fluids*. In Computer Graphics (Proceedings of SIGGRAPH 99), Annual Conference Series, pages 121–128. ACM SIGGRAPH, August 1999. (p. 1)

Robert F. Stengel. *Optimal Control and Estimation*. Dover Books on Advanced Mathematics, New York, 1994. (pp. 14, 18, 46)

Josef Stoer and Roland Bulirsch. *Introduction to Numerical Analysis*. Springer Verlag, New York, second edition, 1980. (p. 13)

Keith R. Symon. *Mechanics, Third Edition*. Addison-Wesley Publishing Company, Reading, Massachussetts, 1971. (pp. 5, 8, 26)

Diane Tang, J. Thomas Ngo, and Joe Marks. *N-Body Spacetime Constraints*. Journal of Visualization and Computer Animation, 6:143–154, 1995. (p. 18)

Demetri Terzopoulos and Kurt Fleischer. *Deformable Models*. Visual Computer, 4(6):306–331, 1988. (p. 1)

Demetri Terzopoulos and Kurt Fleischer. *Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture*. In Computer Graphics (Proceedings of SIGGRAPH 88, Annual Conference Series, pages 269–278. ACM SIGGRAPH, August 1988. (p. 1)

Demetri Terzopoulos, John Platt, Alan Barr, David Zeltzer, Andrew Witkin, and Jim Blinn. *Physically-Based Modeling: Past, Present, and Future*. Computer Graphics, 23(5), December 1989. (p. 2)

Michiel van de Panne, Eugene Fiume, and Zvonko Vranesic. *Reusable Motion Synthesis using State-Space Controllers*. In Computer Graphics (Proceedings of SIGGRAPH 90), Annual Conference Series, pages 225–234. ACM SIGGRAPH, August 1990. (p. 17)

Michiel van de Panne and Eugene Fiume. *Sensor-Actuator Networks*. In Computer Graphics (Proceedings of SIGGRAPH 93), Annual Conference Series, pages 335–342. ACM SIGGRAPH, August 1993. (p. 17)

Reinhold von Schwerin. *MultiBody System Simulation: Numerical Methods, Algorithms, and Software*, volume 7 of *Lecture Notes in Computation Science and Engineering*. Springer-Verlag, 1999. (p. 70)

Andrew Witkin and Michael Kass. *Spacetime Constraints*. In Computer Graphics (Proceedings of SIGGRAPH 88), Annual Conference Series, pages 159–168. ACM SIGGRAPH, August 1988. (p. 15)

Andrew Witkin, Michael Gleicher, and William Welch. *Interactive Dynamics*. In Proceedings of the 1990 symposium on Interactive 3D graphics, pages 11–21, March 1990. (p. 21)