

INTERACTIVE CONTROL OF LINKED RIGID BODY SIMULATIONS

by

JONATHAN JONG-HO LEE

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL
ENGINEERING AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF ENGINEERING IN COMPUTER SCIENCE AND
ENGINEERING

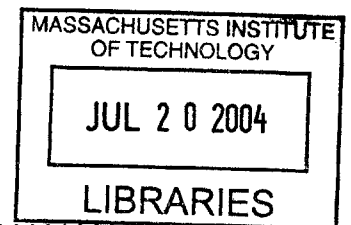
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2003

© Jonathan Jong-ho Lee, MMIII. All rights reserved.

The author hereby grants to MIT permission to reproduce and
distribute publicly paper and electronic copies of this thesis and to
grant others the right to do so.



Author

Department of Electrical Engineering and Computer Science

August 21, 2003

Certified by...


Assistant Professor, Computer Graphics Group
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Interactive Control of Linked Rigid Body Simulations

by

Jonathan Jong-ho Lee

Submitted to the Department of Electrical Engineering and Computer Science
on August 21, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Numerical simulation of linked rigid-body motion generates realistic motion automatically. Simulation alone, however, prevents intuitive direction. In the worst case, an artist repeatedly tweaks a variety of simulation parameters until she achieves the desired effect. Because of the complexity of the dynamics of linked rigid bodies, the tweaking becomes tedious, even for the simplest animation goals.

This thesis describes an interactive technique for direct control of linked rigid-body simulation with a click-and-drag interface. It uses numerical simulation to preserve the physical realism of the motion. Differential control, an interactive form of gradient descent, provides the basis of the interface. This thesis extends differential control to edit the motion of linked rigid bodies connected with passive joints in an open-loop topology. The linked object can collide for an instantaneous or sustained time period at a single point of contact.

Thesis Supervisor: Jovan Popović

Title: Assistant Professor, Computer Graphics Group

Acknowledgments

This thesis would not have been possible without the generosity and kindness of my advisor, Professor Jovan Popović, to whom I am thankful for providing me the exciting opportunity to work with him on this project. I must thank him for his patience and willingness to help me when the math just did not make any sense, and for helping me understand the big difference between graduate and undergraduate life.

I also thank my brothers at Sigma Nu who have given me support throughout my undergraduate and graduate years. Thanks to Stephen Larson for convincing me to take more time. Pavel Gorelik and Eric Konopka were kind enough to read my thesis and provide useful revisions and comments. Special thanks to Doug Quattrochi and Farid Jahanmir for their extensive feedback and interest.

Thank you to Adnan Sulejmanpašić from the Graphics Group for his thoughts and feedback.

I thank my parents, Jean and David, for giving me the opportunity to attend this school. I thank them and my sisters Allison and Christine for their support.

Contents

1	Introduction	13
1.1	Animation Issues and Overview	15
1.2	Problem Statement	17
1.3	Example: Shooting a Basketball	19
2	Previous Work	23
3	Simulating Motion	27
3.1	Generalized State and Control Vectors	29
3.2	Simulation	31
3.3	Changing Object Dynamics	33
4	Interface	37
4.1	Layout and Camera Controls	39
4.2	Object Selection	41
4.3	Key Frame Constraints	41
4.3.1	Basic Algorithm	43
4.3.2	Point Constraints	44
4.3.3	State Constraints	47
4.4	Adjusting the Control Vector	49
5	Editing Motion	55
5.1	Jacobian of Motion	56
5.2	Scaling Attributes	57

5.3	Finding Optimal Change	59
5.4	Changing Control Vector Format	60
5.4.1	Masking	61
5.4.2	Re-parameterizing Quaternions	62
5.5	Residual Check	63
5.6	Editing Across Configuration Changes	63
5.7	Creating Plausible Motion	67
5.8	Re-Simulation	68
6	Results	71
6.1	Example Sessions	71
6.2	Benchmarks	78
7	Conclusion	81
A	Re-parameterization of Quaternions	85
A.1	Basic Quaternion Notation	85
A.2	Quaternion Manipulation	87
A.3	Exponential and Logarithmic Maps	88
A.4	Calculation of Derivatives	89
A.5	Dynamic Re-parameterization	90
B	Camera Controls	91

List of Figures

1-1	A ball’s motion determined by key frames	14
1-2	Estimating the initial velocity of a ball	16
1-3	Diagram of process flow	18
1-4	Motion of basketball created by initial control vector	20
1-5	Artist-directed gesture moving the basketball to desired location . . .	20
1-6	Calculation of needed change to initial control vector	21
1-7	New control vector fulfills artist’s intentions	21
3-1	Two configurations of a double pendulum, with sample trajectories .	28
3-2	Chart with Illustrations of Joints	30
3-3	Changing dynamics from free-flight to clamped	34
4-1	Screenshot of LRBME	40
4-2	Key Frame Constraint Types	44
4-3	Constraint Strength Group Box	45
4-4	Point constraint	46
4-5	Translational constraint	48
4-6	Arcball constraint	50
4-7	Control Vector Attribute Mass Group Box	52
4-8	Toggling Control Vector Parameters	53
5-1	Comparison of system behavior with residual calculation	58
5-2	Residual check prevents editing	64
6-1	Example 1 — Free-falling three-link object	72

6-2	Example 2 — Clamped 4-link chain	74
6-3	Example 3 — Clamping Collision	76
6-4	Graph of time taken per editing loop during re-simulation	78
6-5	Graph of time taken per editing loop by the solver module	80
6-6	Graph depicting overhead of editing	80
B-1	Camera Properties	92

List of Algorithms

4.1	General Editing Process	39
4.2	Ray Intersection Algorithm	42
4.3	Arcball Constraint Algorithm	51
B.1	Camera Rotation	93
B.2	WINDOWTOSPHERE(\mathbf{m}_{win})	94
B.3	Camera Panning	94
B.4	Camera Zoom	95

Chapter 1

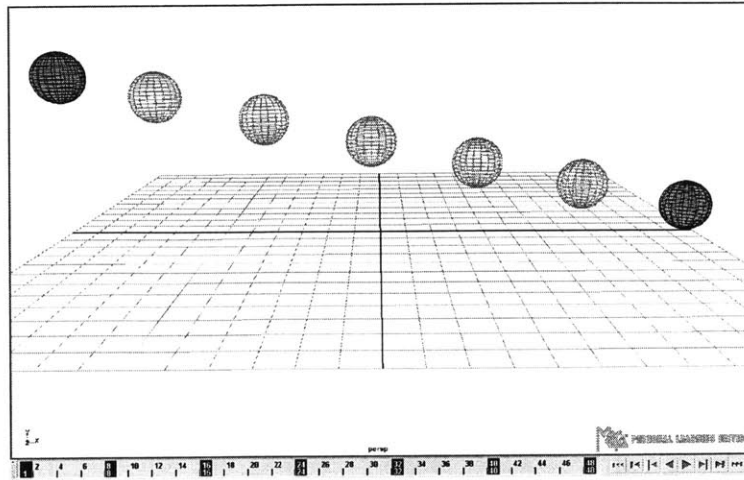
Introduction

Animation is an expressive, visually compelling art form. Its creation, however, is complex and tedious. Traditional techniques require every frame of the animation to be meticulously drawn by a skilled animator [30]. Fortunately, with progressing technology, more sophisticated tools and animation creation techniques are available for artists to use. Now they can draw a sparse set of *key frames* and have a computer generate the remaining frames in the animation. This technique gives the artist the greatest control over the object's motion, while saving her the tedium of drawing the many in-between frames.

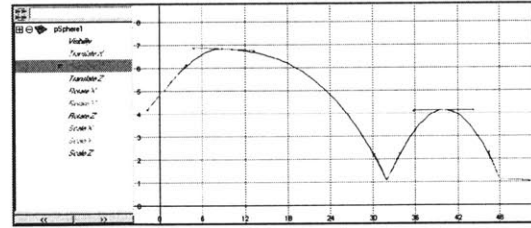
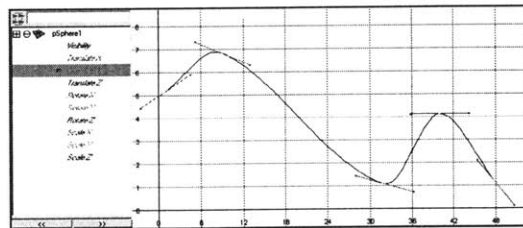
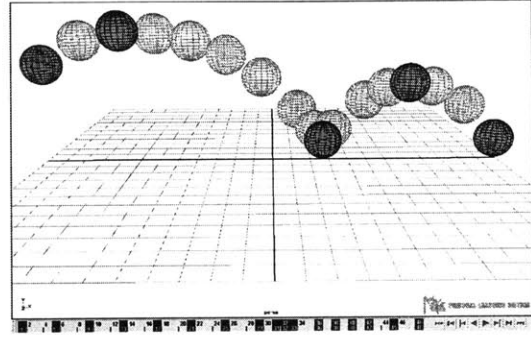
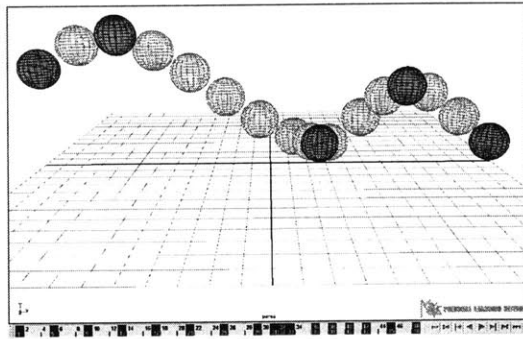
Although a quicker process, key framing does have its disadvantages, especially for non-artists. Firstly, the technique requires a talented artist to draw the important key frames. Secondly, the initial key frame set may not provide enough detail to produce an appropriate motion. Additional key frames may be needed to clarify the motion's overall trajectory or the transitions from one gesture to another. Often this requires a precise intuition of timing.

Figure 1-1 illustrates key framing using a popular graphics production tool.¹ The artist intends to animate a ball bouncing and arriving at a desired location. In Figure 1-1[a] the artist naïvely sets up two key frames—frame 1, where the ball begins, and frame 48, where the ball should arrive. The key frame set is insufficient because interpolation generates an animated ball which does not bounce. To correct this, the

¹Maya 4.5 Personal Learning Edition. <http://www.aliaswavefront.com>.



[a]



[b]

[c]

Figure 1-1: **A ball's motion determined by key frames.** For simplicity, most of the controls are removed from the image and individual frames of animation are composited into one image. The darker balls refer to the key frames, whereas the lighter balls refer to interpolated frames. Frames shown in the scene have their corresponding frame indices displayed as a black bar in the timeline below. [a] The two key frames provided by the artist is insufficient because interpolation generates an animated ball which does not bounce. [b] The artist provides more detail with the addition of three new key frames. The horizontal axis on the position graph below represents the key frames, and the vertical axis represents the ball's y-axis coordinate. The ball's position at each key frame acts as a control point on an interpolated Bézier curve; the handles on each key frame denote the tangent of the curve. The curvature clearly does not mimic the expected trajectory of a bouncing ball. [c] Here the tangents of the control points have been altered on the position graph to make the ball's behavior more realistic.

artist adds three key frames as shown in Figure 1-1[b], two for the peaks of each of the free-flight trajectories (frames 9 and 40) and one for the location of the bounce (frame 32). Even with the additional key frames the motion is still incorrect. The system assumes the key frames are control points of a Bézier curve and interpolates the position of the ball on the curve. For the ball to exhibit correct behavior, the artist needs to tweak the tangents of the control points (Figure 1-1[c]).

Physical simulation generates the frames of the animation automatically. The simulator uses the parameters, which are specified by the artist, to compute the motion. The automation of the animation process makes this technique particularly appealing. However, when the artist has particular goals for the motion, she must manually tweak the simulation parameters, a process that can become tedious. This reliance on tweaking the parameters is inadequate because a small change in one parameter can produce a drastically different animation.

In Figure 1-2 the artist wants the ball to depart from its first location and land on the ‘X’ after a specified period of time. If the physical simulator computes the motion from any initial value (Figure 1-2[a]), the ball will not reach the location of the ‘X’. The artist must make many adjustments before an acceptable animation is found (Figure 1-2[d]).

In this particular example the correct velocity can be derived analytically. However, finding the correct initial parameters for more complex motions would not be as simple.

1.1 Animation Issues and Overview

The previous examples reflect several challenges to creating easy-to-use animation tools. Firstly, the artist must animate both the fine details and the overall gesture. Instead of letting the artist add levels of details as necessary, she must pay close attention to every aspect of the motion. Although physical simulation can create a motion quickly, simulation lacks any type of control that allows the artist to refine the motion. Only the simulation parameters are available for the artist to adjust.

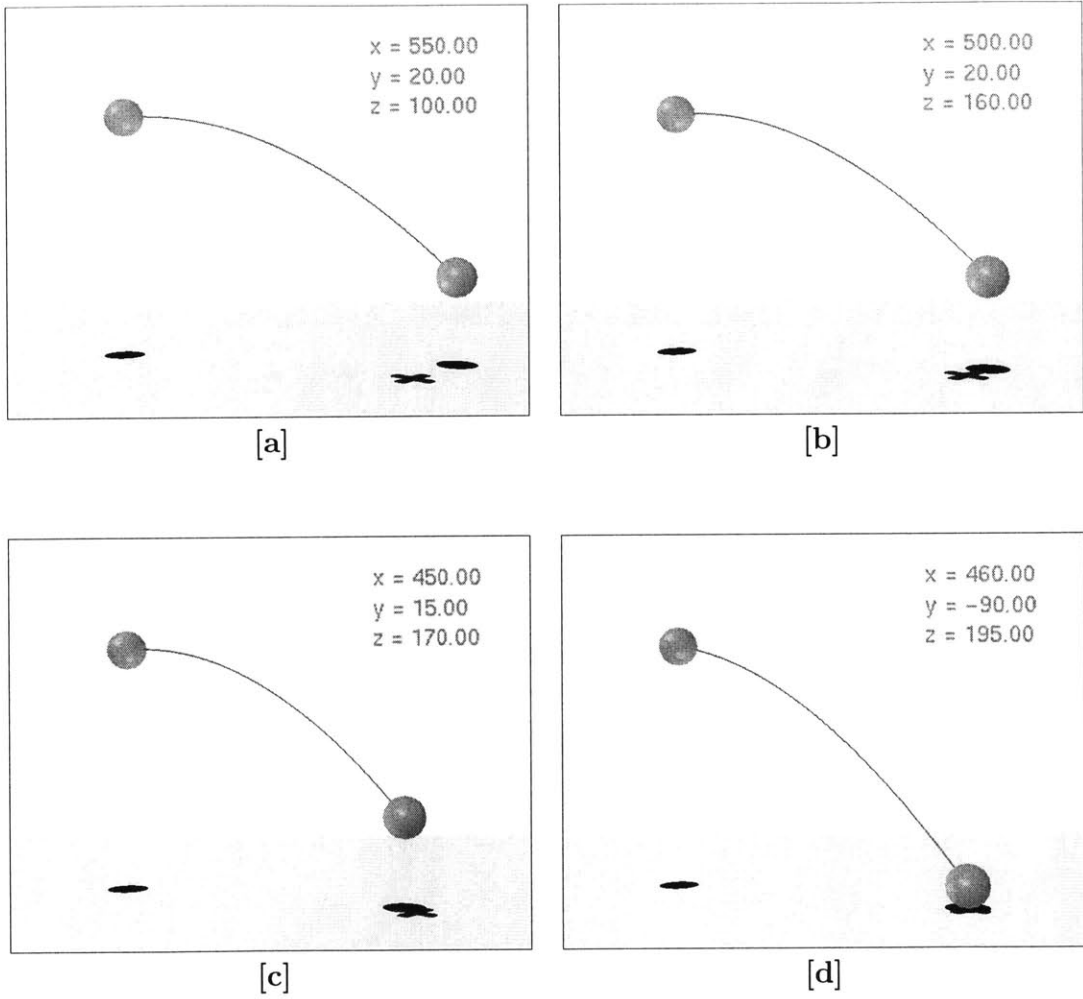


Figure 1-2: **Estimating the initial velocity of a ball.** The upper right-hand corner of each frame denotes the ball's initial velocity along each of the coordinate axes. The entire trajectory and the first and last frames of the simulation are shown.

Secondly, animation techniques often lack immediate feedback. An artist drawing key frames needs to draw all if not most of the key frames before she can judge the quality of the motion. Likewise, an artist simulating an object must tweak parameters and wait for the simulator to finish re-calculating the motion every time she is unsatisfied with the result. To be interactive, a procedure must provide immediate feedback to the artist. To be intuitive, the procedure must behave predictably, giving results that the artist expects. Thirdly, animating natural movement requires a good sense of timing and, in many cases, physics. Even when making cartoon-style movement where physics is exaggerated or completely discarded [20], the “squash and stretch” of an object must be timed well to convey the appropriate message to the viewing audience.

This thesis describes a semi-automatic technique that combines the benefits of key framing with that of physical simulation by addressing several important issues. The technique employs physical simulation to generate the motion, and provides the artist an interface that allows her to edit the motion by essentially key framing the important time instants. Consequently, the artist can add as much detail as she deems necessary to make a satisfactory animation. She may adjust in real-time the paths of all the objects by manipulating the object directly. The objects in the model are rigid bodies connected by joints in a tree structure. Collisions with the external environment are allowed.

1.2 Problem Statement

The artist first determines whether a physically-based simulation, initialized by a *control vector*, satisfies her animation goals. If not, she scrubs through the simulation and finds the time instants where important poses or aspects of the motion must be preserved or changed. At these instants she may place *key frame constraints* that dictate how the control vector should be altered. The ideal control vector initializes a simulation which satisfies all the artist-specified key frame constraints.

Calculation of this ideal control vector is difficult to solve. In fact, a solution

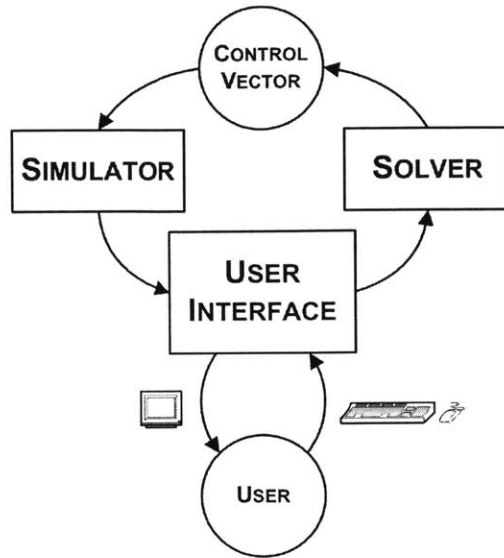


Figure 1-3: Diagram of process flow.

may not necessarily exist if too many constraints are placed. The domain of possible simulations is high-dimensional, and the simulation itself is non-linear and not necessarily continuous. To cope with these difficulties, a differential approach is taken to manipulate the control vector.

The key frame constraints placed by the artist represent desired aspects of the motion. The difference between the desired and the current aspects denote wanted changes to the motion. For example, a certain point on the object must be at a specific location in the surrounding environment at a specific time. The difference between the desired location and current location of the point represents a wanted change to the motion. Such changes help determine an incremental shift to the control vector. After the control vector takes a small step along the direction of the shift, a new simulation is calculated using the adjusted control vector. This simulation should better satisfy the artist's key frame constraints by reducing the differences between the desired and current aspects. The differences are re-calculated, and the process, or *editing loop*, repeats until the ideal control vector is found, or the artist is satisfied with the current simulation. The loop occurs very quickly, so that the artist may receive and in turn provide immediate feedback.

This interactive procedure is manifested in an application called Linked Rigid Body Motion Editor, or LRBME. The application contains three modules: the simulator module, the solver module, and the graphical user interface. Given an initial control vector, the simulator module calculates the object's motion and stores the data. The user interface relies on the stored data to correctly render the object and its behavior on the monitor. Since the initial resulting motion will most likely not satisfy the artist's intent, she can edit the motion by directly manipulating it. Any mouse or keyboard inputs are passed on to the solver. The solver computes the changes needed to accommodate the edits while still maintaining a physically correct motion. Once the simulator module incorporates the changes and re-simulates the motion, the user interface updates the motion displayed on the monitor. Figure 1-3 provides a graphical overview that will be referenced throughout the thesis.

1.3 Example: Shooting a Basketball

An illustration of the process is shown in Figures 1-4 through 1-7. Here the artist wants the basketball to begin at its initial location at time t_0 and land in the hoop at time t_f . The control vector is set to an initial guess and is given to the simulator module, which outputs the motion the artist can view through the user interface (Figure 1-4). Clearly the initial guess is insufficient since the basketball does not reach the hoop at the final frame. The artist grabs with her mouse the ball at time t_f and drags it to the hoop (Figure 1-5). The dragging motion creates a desired change in motion. From this change the solver module determines the proper shift to the control vector (Figure 1-6). The new control vector incorporates the shift with its previous value. It passes through the simulator module again, and the revised motion is displayed to the artist (Figure 1-7). This entire loop occurs very quickly so that the artist can immediately see how her dragging the basketball to the hoop affects the overall motion.

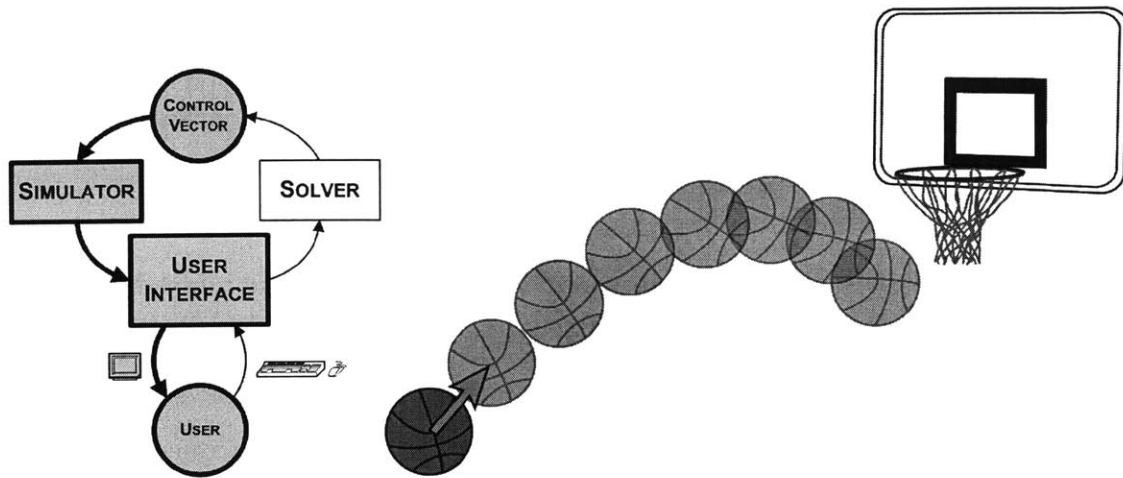


Figure 1-4: **Motion of basketball created by initial control vector.** The highlighted portions of the flow chart (from Figure 1-3) represent the parts of the process shown in the adjacent figure. The arrow on the initial frame of the ball represents the control vector. The initial guess is inadequate because the basketball does not reach the hoop in the final frame.

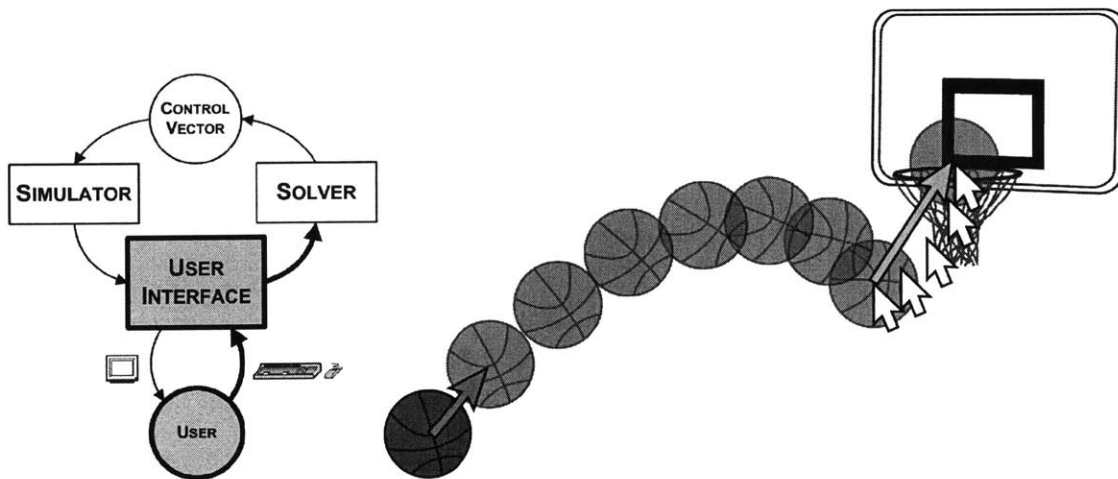


Figure 1-5: **Artist-directed gesture moving the basketball to desired location.** The artist drags the basketball using her mouse in the final frame toward the hoop. The difference of the basketball's desired location (in the hoop) and its current location represents an adjustment to the motion.

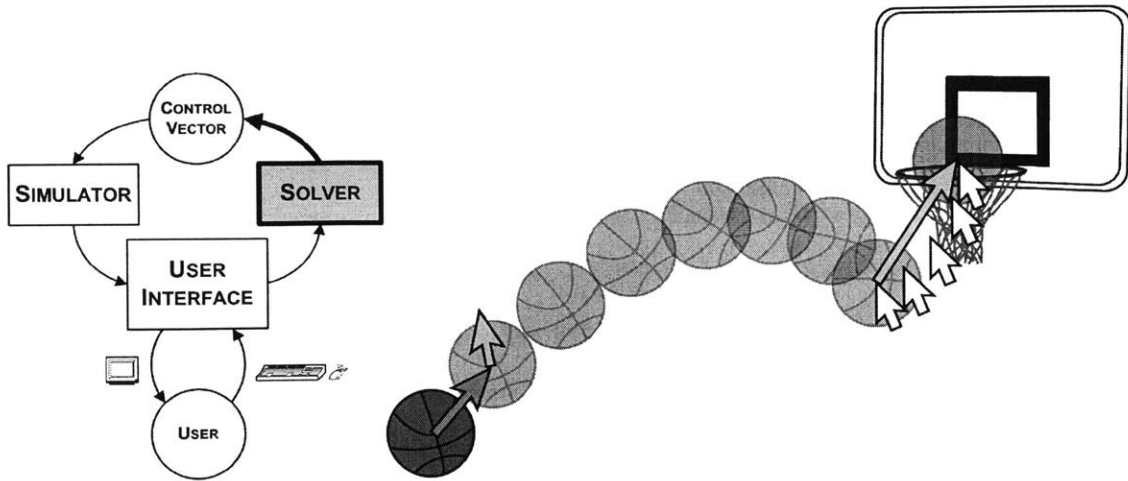


Figure 1-6: **Calculation of needed change to initial control vector.** The needed motion adjustment leads to a shift in the control vector, represented by the small arrow above the original control vector.

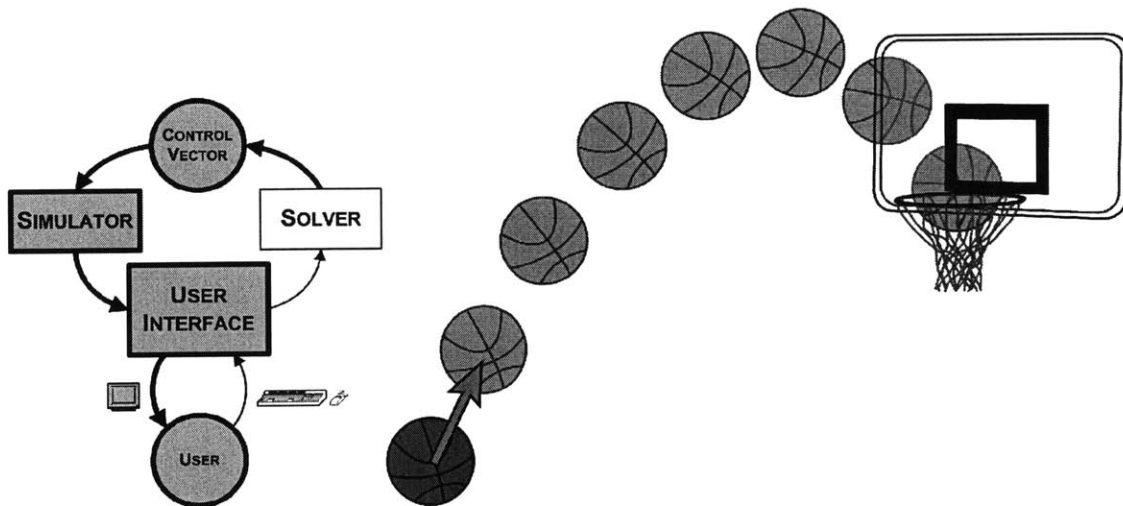


Figure 1-7: **New control vector fulfills artist's intentions.** The control vector takes a step in the direction of the shift, leading to a new control vector. The adjusted control vector initializes a new simulation that successfully fulfills the goal of the animation.

Chapter 2

Previous Work

Computer animation techniques rely heavily on physical simulation to generate animation for complex dynamics such as rigid bodies [15], deformable bodies [23, 1], fluids [8], smoke [7, 31], and cloth [29, 2]. These techniques aim to create mechanisms for simulating complex physical processes and to find effective means of controlling simulation.

Control of simulations has often led to optimization problems. The spacetime-constraints method formulates a finite-dimensional optimization problem that computes a character’s motion over a time period. The solution is constrained to meet the artistic animation goals, which are stated as constraints to the optimization problem. The original spacetime method successfully determined coordinated motions of a virtual actor [33]. Finding a suitable solution, however, took long periods of time. This thesis aims to provide the artist immediate feedback, necessitating a formulation of the spacetime problem that can be quickly solved.

Machine learning techniques have been used to compute controllers which generate desired motions. Neural networks can be trained to learn an approximate simulation function from simulated motions [14]. With the trained, analytically differentiable neural network, a fast gradient descent optimization algorithm computes the controller that generates the user-defined motions. Simulated annealing algorithms can also produce efficient low-level locomotion for animals whose skeletons contains many degrees of freedom [13]. The low-level motion controllers become abstract constructs

which the user concatenates to produce higher-level behaviors. The spacetime framework can employ genetic algorithms to globally sample the controller space [22]. To create active locomotion of a 2-D character, a set of functions represents instinctive reflexes for the character in response to environmental stimuli. Physical simulation along with the stimulus-response mechanism provides a motion that is evaluated quantitatively. The genetic algorithm searches through the stimulus-response parameter space to maximize the distance the object’s center of mass travels.

In the context of parameter estimation, Chenney and Forsyth proposed a Markov Chain Monte Carlo technique that iteratively samples the space of simulation parameters to determine the fitness of the generated animations by a set of artist-specified criteria [5]. A proposal function provided by the artist instructs the algorithm how to tweak the parameters of an animation sample. Although this technique is the most general solution to date, the process does not allow interactive exploration and depends heavily on the quality of the user-provided proposal function. A flawed proposal function could lead to slow convergence.

Off-line techniques provide the advantage of sampling the solution space globally, leading to optimal or near-optimal controllers that are capable of producing compelling and complex motion. Due to the iterative nature of these techniques, however, they cannot lead to interactive applications, and do not pertain directly to this thesis. In situations where artist creativity is a priority, researchers have developed more interactive techniques that provide interfaces artists can use to specify their intent. Cohen designed a process where the spacetime problem is solved with user assistance [6]. Using conjugate gradient methods, the system provides the user the opportunity to modify the functions and parameters of the optimization problem as the solution converges. Moreover, division of the animation into smaller spacetime “windows” helps focus the solution process. Gleicher simplified the spacetime formulation to achieve interactive performance for real-time editing [9]. The system provides an interface that hides the underlying equations, and constraints are specified by script or by direct manipulation of the character.

Witkin et al. provided a flexible framework of interactive dynamics which allows

constraints, added or removed by the user at will, to dynamically rephrase the linear system of equations [32]. The constrained dynamics formulation has led to a broad range of applications such as drawing applications [11], interactive camera control [10], and geometric modeling [32].

The interactive physically-based modeling framework has been expanded to permit both continuous and discrete changes in the spatial relationship between objects [16]. Mouse gestures are modeled as physical forces that modify state variables for the object. A mass matrix represents the relative difficulty of change among the parameters. When making a discrete change in the object’s configuration, the user triggers a search for a new object state whose parameters are minimally and locally altered from the current object state. This infrastructure has been successfully applied to the interactive designing of architectural, circuit board, and page layouts.

Popović et al. described a differential-based technique which relies on the computation of derivatives for faster convergence for parameter estimation methods [25]. Using the interactive dynamics framework proposed by Witkin and colleagues [32], this approach uses a differential control to manipulate passive rigid-body simulations with an interactive click-and-drag interface. Although unable to detect motion discontinuities automatically, the interface permits the user to guide the object to the desired collision sequence.

Differential control tries to find the best animation by taking the current control mechanism and changing it slightly with the hopes of finding a better animation. Since convergence depends on gradient information, the solution reaches only local optima. In an interactive setting finding local optima is acceptable—an artist’s edit, denoting the intended state, effectively guides the motion out of undesired optima.

The parameter estimation paradigm is difficult to control since the parameters only affect the system’s initial state. To find optimal simulation parameters, a relationship between all user-provided constraints and the parameters must be found, which can be difficult to develop. *Multiple shooting* methods provide better control by dividing the entire simulation into smaller and more manageable subproblems. Once the subproblems are solved, they are merged either by enforcement of continuity

constraints or by solving overlapping subproblems which implicitly seam boundaries. The extra control in these techniques stems from the artist’s ability to adjust parameters that affect the simulation at various times throughout its duration. This technique has been used successfully to control simulations of both rigid-body and smoke simulations [24, 31].

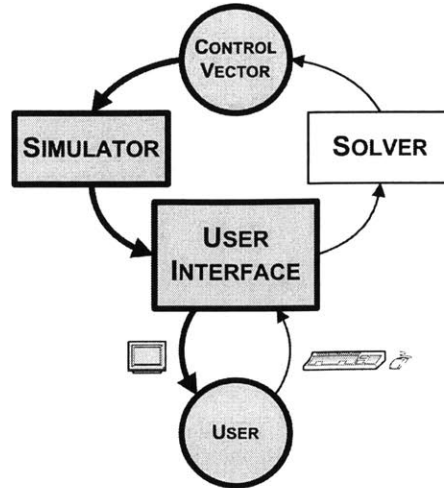
Since multiple shooting methods often lead to large objective functions and constraint matrices, the search for the ideal parameter values is done off-line. However, Popović provided an appealing production pipeline called *motion sketching* that provides the artist the benefit of specifying her intents interactively, then querying the system to optimize the controller [24]. Controlling simulations through use of multiple shooting techniques in an interactive setting still remains an unsolved problem.

This thesis, using single shooting methods, provides the artist a click-and-drag interface for interactive control of linked rigid-body simulations. The interface allows an animator to select the simulated object and drag it to the desired location at any point in the simulation. This direct and intuitive approach is superior to straightforward parameter tweaking. The only overhead comes from the computation of *sensitivity matrices*, which are the derivatives of the simulated state with respect to the simulation parameters.

A numerical simulator can detect collisions with the external environment that change the object’s dynamics [15]. At the time of collision, an impulse force may be applied [21]. Popović et al. showed that differential control is possible for a small number of instantaneous collisions [25] but a sustained contact may need many microcollisions. To solve this problem, LRBME switches the object’s dynamics at each transition to and from a sustained contact. This thesis does not address multi-point contacts or contacts with non-holonomic constraints. LRBME does not automatically detect collisions, and instead expects the user to define the appropriate time instants at which the object collides with the external environment. If the location of the collision is incorrect, the user may manipulate the object to correct the motion.

Chapter 3

Simulating Motion



A *linked rigid body* consists of several bodies connected together by joints. Each body is *rigid* because its shape does not deform. A joint can attach only two bodies together, and the total structure of the object must be in the form of a tree—that is, no loops are allowed. The body that is at the top of the joint hierarchy is called the *root body*. A joint connects an *outboard body* to an *inboard body*. Structurally an outboard body is a child of the inboard body. Any rigid body that is not the root body must be connected to one inboard body.

Each individual body of an object has several properties, all of which are assumed to be known prior to simulation: mass, inertia tensor, location of the inboard joint, type of joint, and inboard body. Mass is simply the mass of the individual body. The inertia tensor is a 3×3 matrix denoting the body's moment of inertia in its resting state. Each body knows its parent body, the type of the inboard joint that connects the body to its parent, the joint's location with respect to the body's local reference frame, and how the joint is oriented with respect to the body's reference frame.

An object is allowed to contain pin, universal, or ball joints. The *pin joint* has one degree of freedom (DOF), and is described by one parameter denoting the angle about the joint's axis of rotation. The two-DOF *universal joint* is described by two numbers denoting the angles about each of the joint's two known axes. Exhibiting three DOFs, the *ball joint* is described as a unit-normalized quaternion vector.

An object can use a *gimbal joint* instead of a ball joint. The gimbal joint also

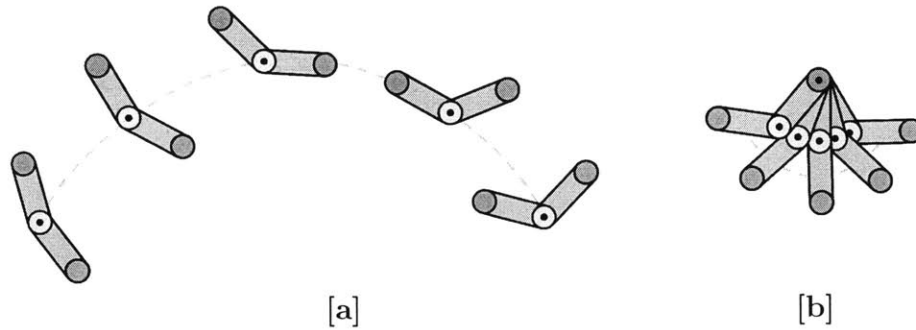


Figure 3-1: **Two configurations of a two-body arm, with sample trajectories.** A black dot in the middle of a circle represents a joint at that location. [a] The arm is in free-flight. [b] The first body of the arm has a point clamped in space. The arm has the dynamics of a double pendulum.

provides three DOFs and consists of three hinges, one for each rotation axis. Because the performance of the interactive editing process largely depends upon the size of the state and control vector (Chapter 6), using three-parameter gimbal joints to represent free rotations would appear to be a better choice over the four-parameter ball joint. Gimbal joints, however, suffer from *gimbal lock*, a singularity that occurs when the first and third axes are aligned, reducing the joint from three to two degrees of freedom. Since unrestricted rotation is desired, the ball joint is used.

An object may exhibit different *configurations* during an animation. A configuration represents the dynamics of the object as dictated by its relationship with the external environment. For example, a two-body arm consists of two cylinders connected at their ends by a pin joint (Figure 3-1). This arm can relate to the external environment in two main ways. Firstly, the arm's configuration can be *free-falling* as shown in Figure 3-1[a], where the arm flies through the air. Secondly, the arm can have any point on it fixed in space—in this case its configuration is *clamped*. Figure 3-1[b] illustrates the arm clamped at the end of one of its bodies. The dynamics of the arm differ between the two configurations. In the free-falling configuration, the arm has seven DOFs: the root body contains three rotational and three translational DOFs, and the joint has one rotational DOF. In the clamped configuration, the arm exhibits the dynamics of a double pendulum, losing its three translational DOFs.

3.1 Generalized State and Control Vectors

The state of a mechanical system of one or more rigid bodies linked by joints can be described by a generalized state vector \mathbf{y} which consists of two smaller vectors:

$$\mathbf{y} = \begin{bmatrix} \mathbf{q} \\ \mathbf{u} \end{bmatrix}$$

where vectors \mathbf{q} and \mathbf{u} ($= \dot{\mathbf{q}}$) represent the *generalized coordinates* and *velocities* of the object, respectively. More explicitly, the vector \mathbf{q} contains information regarding the positions and orientations, and the vector \mathbf{u} the linear and angular velocities of the object. Particular groups of elements in \mathbf{y} describe the parameters regarding each individual rigid body that comprise the entire object.

Since the root body is the ancestor of all the other bodies in the object, only one set of position and linear velocity parameters exist in the state vector. All other parameters are related to angles and angular velocities. A chart describing the joints and the number of parameters they represent in the state vector is shown in Figure 3-2 on the following page.

Regardless of the object's configuration, the root body is related to the external environment by a joint. If the object is free-falling, a *free joint* containing three translational and three rotational DOFs "connects" the root body to the environment. If the object is clamped, the root body is connected by the proper rotational joint.

The control vector \mathbf{p} consists of parameters that initialize the object, such as initial position and velocity. In most cases, the format of the control vector assimilates that of the object's state vector. Additional elements which parameterize other controllable aspects of the motion can be appended to the control vector. Examples include body masses, wind forces, or additional velocities added at important times during the animation (Section 5.7).

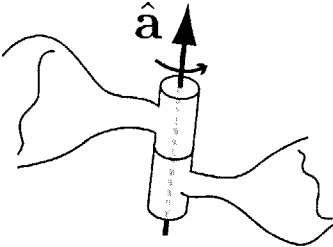
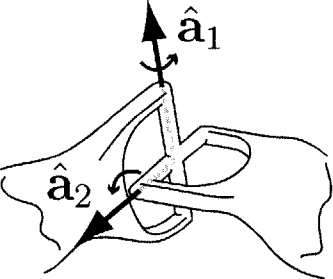
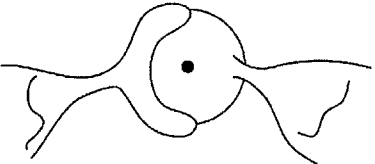
Type of Joint	Description	Rotation Axes	Number of Angle Parameters	Number of Angular Velocity Parameters
Pin		\hat{a}	1	1
Universal		\hat{a}_1, \hat{a}_2	2	2
Ball		quaternion	4	3 (along \hat{x} , \hat{y} , \hat{z} axes)

Figure 3-2: **Chart with Illustration of Joints.** The state vector does not encode the axis or axes about which the pin or universal joint revolve. That information is implicitly encoded in the physics equations used to simulate the object. The vectors \hat{x} , \hat{y} , and \hat{z} represent the unit vectors pointing along the positive x-, y-, and z-axes in the world frame, respectively.

3.2 Simulation

An object's behavior can be described by a set of ordinary second-order differential equations:

$$\frac{d}{dt}\mathbf{y}(t) = \mathcal{F}(t, \mathbf{y}(t), \mathbf{p}) \quad (3.1)$$

where \mathbf{p} is the control vector and $\mathcal{F}(t, \mathbf{y}(t), \mathbf{p})$ is derived from Newton's laws. Integrating Equation 3.1 and solving for \mathbf{y} yields the *simulation function* \mathcal{S} :

$$\mathcal{S}(t_i, \mathbf{p}) = \mathbf{y}_{t_i} = \mathbf{y}_{t_0}(\mathbf{p}) + \int_{t=t_0}^{t_i} \mathcal{F}(t, \mathbf{y}(t), \mathbf{p}) dt \quad (3.2)$$

For clarity, the subscript t on \mathbf{y}_t denotes an evaluation of the object's state vector at time t , whereas the notation $\mathbf{y}(t)$ indicates the state vector's dependence on time variable t .

The artist may wish the object to perform a combination of free-flight and clamped motion. In this case the object collides instantaneously with the external environment, changing the object's dynamics. Upon switching configurations at the time of collision t_c , the object's post-collision behavior may be governed by a different set of physics equations. A *collision transfer function* \mathcal{C} converts the state of the object immediately prior to collision, $\mathbf{y}_{t_c}^-$, to a corresponding state after collision, $\mathbf{y}_{t_c}^+$:

$$\mathcal{C}(\mathbf{y}_{t_c}^-) = \mathbf{y}_{t_c}^+ \quad (3.3)$$

To calculate \mathcal{S} after the collision time t_c , the simulator module uses the post-collision state as the initial state vector and integrates new dynamics equations \mathcal{G} :

$$\mathbf{y}_t = \mathcal{C}(\mathbf{y}_{t_c}^-) + \int_{t=t_c}^{t'} \mathcal{G}(t, \mathbf{y}(t), \mathcal{C}(\mathbf{y}_{t_c}^-)) dt, \text{ where } t' > t_c \quad (3.4)$$

The object can change configurations as many times as the artist wishes. Running from time t_0 to time t_f , the object changes configurations at known times $t_i^C \in \{t_1^C, \dots, t_m^C\}$, where $t_0 = t_0^C < t_1^C < t_2^C < \dots < t_m^C < t_f$.

A *clip* represents a continuous length of time in which the object runs in a particular configuration. Each clip can be considered to be a complete simulation by itself. As such, each clip has its own initial state vector and its own dynamics equations. So, a clip i simulates the object from time t_i^C to time t_{i+1}^C , and the object's behavior during this period is governed by the set of Newton's equation \mathcal{F}_i , where $i = [0, m]$. Equation 3.2, then, is rewritten as:

$$\mathbf{y}_t = \mathcal{S}(t, \mathbf{p}) = \begin{cases} \mathbf{y}_{t_0}(\mathbf{p}) + \int_{t_0}^t \mathcal{F}_0(t, \mathbf{y}(t), \mathbf{p}) dt & \text{if } t \in [t_0, t_1^C) \\ \mathcal{C}(\mathbf{y}_{t_1^C}^-) + \int_{t_1^C}^t \mathcal{F}_1(t, \mathbf{y}(t), \mathcal{C}(\mathbf{y}_{t_1^C}^-)) dt & \text{if } t \in [t_1^C, t_2^C) \\ \mathcal{C}(\mathbf{y}_{t_2^C}^-) + \int_{t_2^C}^t \mathcal{F}_2(t, \mathbf{y}(t), \mathcal{C}(\mathbf{y}_{t_2^C}^-)) dt & \text{if } t \in [t_2^C, t_3^C) \\ \vdots & \\ \mathcal{C}(\mathbf{y}_{t_m^C}^-) + \int_{t_m^C}^t \mathcal{F}_m(t, \mathbf{y}(t), \mathcal{C}(\mathbf{y}_{t_m^C}^-)) dt & \text{if } t \in [t_m^C, t_f] \end{cases} \quad (3.5)$$

Obviously, the object does not need a unique configuration for each clip. The configurations between adjacent clips, however, are assumed to be different. For example, an artist wishes to create a motion where a linked rigid body character jumps off a ledge, grabs onto a bar, swings around the bar once, releases, then lands. This motion can be decomposed into three clips: the character begins in free flight, transfers to a clamped configuration when it grabs the bar, and then returns to free-falling during dismount. Since the first and third clips use the same set of dynamics equations, only two sets of equations are needed for simulating the entire motion.

For LRBME, the simulator module generates frames of the motion by calculating Equation 3.2 at discrete time intervals using an adaptive fifth-order Runge-Kutta method. The computation of the appropriate accelerations is complex, but for acyclic objects they can be computed in linear time with generalized coordinates [18, 26]. Given a description of an object composed of linked rigid bodies, the SD/FAST software package [17] produces FORTRAN subroutines that compute the function $\mathcal{F}(t, \mathbf{y}(t), \mathbf{p})$.

3.3 Changing Object Dynamics

The collision transfer function \mathcal{C} , mentioned in Section 3.2, converts the object's state immediately before a change in dynamics. The function changes the joint hierarchy and replaces the current system dynamics in three steps (Figure 3-3). Firstly, the new joint hierarchy is rooted at the contact point. This step expresses the current joint positions and velocities with respect to the root joint of the new hierarchy. Secondly, the impulse force instantaneously changes the linear and angular momenta of the body. The impulse for both elastic and inelastic collisions can be computed from empirical laws which relate relative velocities of a contact point before and after contact. In this case, a clamped point would have zero velocity immediately after impact. Finally, the impulse adjusts the generalized state of the colliding body. This step converts the object's old configuration to the new one. With a clamped point, the impulse cancels the velocity of the new root in preparation for parameterization with the new state vector, which cannot modify the position or velocity of the clamped point. Use of basic mechanics properly re-parameterizes the state vector from the old to the new configuration [18]. The transition from sustained contact to free-flight follows a similar process with the exception of the impulse computation, which does not apply.

Suppose a clamping collision occurs at time t_c , where clamp point \mathbf{x} on the root body stays fixed after collision. An impulse force is applied such that the velocity of that point $\dot{\mathbf{x}}$ after collision is zero:

$$\dot{\mathbf{x}}(t_c^+) = \dot{\mathbf{x}}(t_c^-) + \Delta\dot{\mathbf{x}} = 0 \quad (3.6)$$

where $\dot{\mathbf{x}}(t_c^-)$ represents the velocity of the point before collision.

This impulse force \mathbf{J} has two effects on the root body. Firstly, it instantaneously

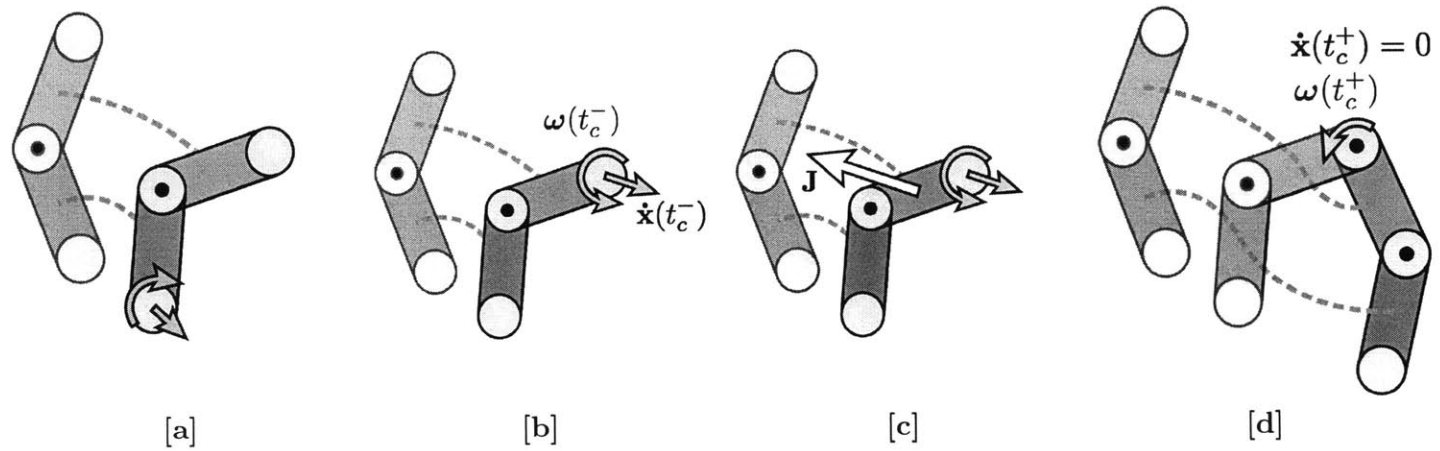


Figure 3-3: **Changing dynamics from free-flight to clamped.** [a] Initially the object is rooted on the bottom arm. The object will clamp on the end of the top arm. [b] Re-rooting expresses the current joint positions and velocities with respect to the root body of the post-impact joint hierarchy. [c] An impulse \mathbf{J} is applied to the center of mass to cancel the linear velocity of the clamp point. [d] The impact may also change the angular velocity of the point to $\omega(t_c^+)$.

changes the velocity of the body at the center of mass \mathbf{c} :

$$\begin{aligned}\dot{\mathbf{c}}(t_c^+) &= \dot{\mathbf{c}}(t_c^-) + \Delta\dot{\mathbf{c}} \\ &= \dot{\mathbf{c}}(t_c^-) + \frac{\mathbf{J}}{m}\end{aligned}$$

where m is the mass of the root body. Secondly, \mathbf{J} changes the body's angular momentum:

$$\begin{aligned}\boldsymbol{\omega}(t_c^+) &= \boldsymbol{\omega}(t_c^-) + \Delta\boldsymbol{\omega} \\ &= \boldsymbol{\omega}(t_c^-) + I^{-1} \cdot \{ (\mathbf{x} - \mathbf{c}) \times \mathbf{J} \}\end{aligned}$$

where I is the body's inertia tensor in the world frame, and $(\mathbf{x} - \mathbf{c})$ is the moment arm from the center of mass to the clamp point.

To solve for \mathbf{J} , the linear velocity of clamp point \mathbf{x} is expressed in terms of the velocity at \mathbf{c} :

$$\dot{\mathbf{x}}(t) = \dot{\mathbf{c}}(t) + \boldsymbol{\omega}(t) \times (\mathbf{x} - \mathbf{c})$$

Expressing this equation at the moment after impact, the equation becomes:

$$\begin{aligned}\dot{\mathbf{x}}(t_c^+) &= \dot{\mathbf{c}}(t_c^+) + \boldsymbol{\omega}(t_c^+) \times (\mathbf{x} - \mathbf{c}) \\ &= \dot{\mathbf{c}}(t_c^-) + \frac{\mathbf{J}}{m} + (\boldsymbol{\omega}(t_c^-) + \Delta\boldsymbol{\omega}) \times (\mathbf{x} - \mathbf{c}) \\ &= \dot{\mathbf{c}}(t_c^-) + \boldsymbol{\omega}(t_c^-) \times (\mathbf{x} - \mathbf{c}) + \frac{\mathbf{J}}{m} + \Delta\boldsymbol{\omega} \times (\mathbf{x} - \mathbf{c}) \\ &= \dot{\mathbf{x}}(t_c^-) + \frac{\mathbf{J}}{m} + \Delta\boldsymbol{\omega} \times (\mathbf{x} - \mathbf{c}) \\ &= \dot{\mathbf{x}}(t_c^-) + \frac{\mathbf{J}}{m} + \left\{ I^{-1} \cdot \{ (\mathbf{x} - \mathbf{c}) \times \mathbf{J} \} \right\} \times (\mathbf{x} - \mathbf{c})\end{aligned}\tag{3.7}$$

The cross product between two vectors \mathbf{a} and \mathbf{b} can be written as $\mathbf{a} \times \mathbf{b}$, or \mathbf{a} can

be turned into skew-symmetric matrix $[\mathbf{a}]_{\times}$ such that $\mathbf{a} \times \mathbf{b} = [\mathbf{a}]_{\times} \mathbf{b}$:

$$\mathbf{a} \times \mathbf{b} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = [\mathbf{a}]_{\times} \mathbf{b}$$

To isolate \mathbf{J} , Equation 3.7 is re-written and manipulated using this operator:

$$\begin{aligned} \dot{\mathbf{x}}(t_c^+) &= \dot{\mathbf{x}}(t_c^-) + \frac{\mathbf{J}}{m} + \left\{ I^{-1} \cdot \{ (\mathbf{x} - \mathbf{c}) \times \mathbf{J} \} \right\} \times (\mathbf{x} - \mathbf{c}) \\ &= \dot{\mathbf{x}}(t_c^-) + \frac{\mathbf{J}}{m} + \left\{ I^{-1} \cdot [\mathbf{x} - \mathbf{c}]_{\times} \cdot \mathbf{J} \right\} \times (\mathbf{x} - \mathbf{c}) \\ &= \dot{\mathbf{x}}(t_c^-) + \frac{\mathbf{J}}{m} - (\mathbf{x} - \mathbf{c}) \times \left\{ I^{-1} \cdot [\mathbf{x} - \mathbf{c}]_{\times} \cdot \mathbf{J} \right\} \\ &= \dot{\mathbf{x}}(t_c^-) + \frac{\mathbf{J}}{m} - \left\{ [\mathbf{x} - \mathbf{c}]_{\times} \cdot I^{-1} \cdot [\mathbf{x} - \mathbf{c}]_{\times} \cdot \mathbf{J} \right\} \\ &= \dot{\mathbf{x}}(t_c^-) + \left\{ \frac{1}{m} [\mathbf{1}] - [\mathbf{x} - \mathbf{c}]_{\times} \cdot I^{-1} \cdot [\mathbf{x} - \mathbf{c}]_{\times} \right\} \mathbf{J} \end{aligned}$$

where $[\mathbf{1}]$ represents a 3×3 identity matrix.

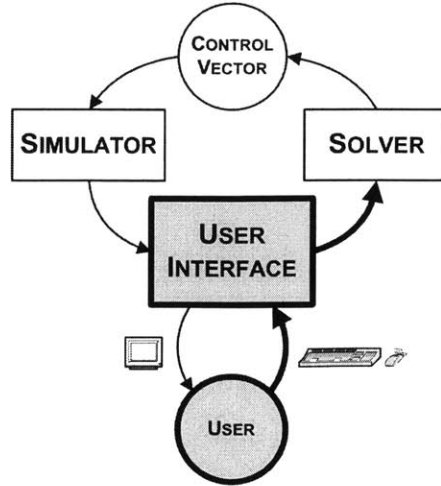
Since $\dot{\mathbf{x}}(t_c^+) = 0$ from Equation 3.6, the following linear system is used to solve for

\mathbf{J} :

$$\left\{ \frac{1}{m} [\mathbf{1}] - [\mathbf{x} - \mathbf{c}]_{\times} \cdot I^{-1} \cdot [\mathbf{x} - \mathbf{c}]_{\times} \right\} \mathbf{J} = -\dot{\mathbf{x}}(t_c^-)$$

Chapter 4

Interface



The simulation computes the motion for a given set of simulation parameters. Initially, the parameters result in a motion that does not satisfy artistic goals. For example, the initial pose may be wrong or the object may not be in a correct location at a later time. The initial pose can be corrected by resetting the initial values for the simulation. But, as the number of constraints increases, the problem becomes more complicated, making parameter tweaking less intuitive.

When the artist edits the motion, she has the option to preserve or change certain aspects of the motion at any time instant. For example, a certain point on the object must be at a specific location in the surrounding environment at a specific time. An *aspect vector* \mathbf{k} represents a controllable feature of the object that the artist wishes to preserve or change; in this example, the aspect vector would represent the location of the point on the object in the world reference frame. An *aspect function* $\mathcal{K}(\mathbf{y})$ defines the aspect vector as a function of the object's state \mathbf{y} :

$$\mathbf{k} = \mathcal{K}(\mathbf{y})$$

The artist modifies the object's behavior by providing a set of desired aspect vectors $\bar{\mathbf{k}}_i$ at times $t_i = \{t_0, \dots, t_n\}$. These aspect vectors serve as key frame constraints on the object's motion. Note that \mathcal{K} can change for each \mathbf{k}_i , and not all constraints need

to preserve the same aspects of the GSV:

$$\forall i, \mathbf{k}_i = \mathcal{K}_i(\mathbf{y}_{t_i})$$

The click-and-drag interface implements a differential technique that iteratively adjusts control vector. In contrast to parameter tweaking, which forces animators to adjustment parameters directly, differential control computes the adjustment from click-and-drag interactions with a simulated object. The interaction yields an aspect vector change $\delta\mathbf{k}$, which describes a desired adjustment in the simulated motion:

$$\mathcal{K}(\mathbf{y}_{\text{new}}) = \mathcal{K}(\mathbf{y}_{\text{old}}) + \delta\mathbf{k}$$

where $\delta\mathbf{k} = \bar{\mathbf{k}} - \mathbf{k}$. Differentiation with the chain rule reveals a linear relationship between the change in control vector $\delta\mathbf{p}$ and $\delta\mathbf{k}$:

$$\forall i, \delta\mathbf{k}_i = \frac{\partial\mathcal{K}_i(\mathcal{S}(t_i, \mathbf{p}))}{\partial\mathcal{S}(t_i, \mathbf{p})} \frac{\partial\mathcal{S}(t_i, \mathbf{p})}{\partial\mathbf{p}} \delta\mathbf{p} \quad (4.1)$$

The system updates the control vector from \mathbf{p}_{old} to \mathbf{p}_{new} by taking a small step ϵ in the direction of $\delta\mathbf{p}$:

$$\mathbf{p}_{\text{new}} = \mathbf{p}_{\text{old}} + \epsilon \cdot \delta\mathbf{p} \quad (4.2)$$

The simulator module uses the new control vector \mathbf{p}_{new} to provide an updated motion that the user interface can display. Algorithm 4.1 shows pseudocode of the process.

Implemented using a cross-platform front-end software package¹, the user interface provides the artist a connection to the simulator and solver modules. This chapter discourses the interaction with LRBME, and the procedural and mathematical implications of the interactions.

¹Qt 3.1.1 Educational Edition. <http://www.trolltech.com>.

Algorithm 4.1 General Editing Process

Inputs:

\mathbf{p}_0 — initial guess for the control vector

- 1: $\mathbf{p}_{\text{new}} \leftarrow$ initial guess \mathbf{p}_0
- 2: Simulate $\mathcal{S}(t, \mathbf{p}_0)$, $t \in [t_0, t_f]$
- 3: Artist places n constraints $\bar{\mathbf{k}}_i, i \in [1, n]$
// BEGIN EDITING LOOP
- 4: **repeat**
- 5: $\mathbf{p}_{\text{old}} \leftarrow \mathbf{p}_{\text{new}}$
- 6: Calculate $\delta \mathbf{k}$
- 7: Calculate $\delta \mathbf{p}$
- 8: $\mathbf{p}_{\text{new}} \leftarrow \mathbf{p}_{\text{old}} + \epsilon \cdot \delta \mathbf{p}$
- 9: Re-simulate $\mathcal{S}(t, \mathbf{p}_{\text{new}})$
- 10: **until** $\forall i, \mathcal{K}_i(\mathbf{y}_{t_i}) = \bar{\mathbf{k}}_i, i \in [1, n]$
// END EDITING LOOP

4.1 Layout and Camera Controls

LRBME consists of two major widgets—a *render window* which displays the motion, and a control panel (Figure 4-1). The artist sets up preferences and specifies options through the control panel, but interacts with the object directly in the render window.

While not in the editing mode, the simulator module samples and saves the object’s motion at uniform discrete time intervals. The artist can use the time slider to see a snapshot of the motion at a particular time instant. The artist has the option of seeing the object’s beginning pose, its ending pose, or the motion trajectory. Standard stop and play buttons allow the artist to view the entire motion.

The artist may adjust the camera view by clicking in a space not occupied by the object in the render window. As the mouse is dragged, the camera changes view accordingly. The artist has the option of rotating the camera freely, rotating about the $\hat{\mathbf{y}}$ -axis², panning, or zooming in or out (Appendix B).

²A hat over a vector represents a unit vector.

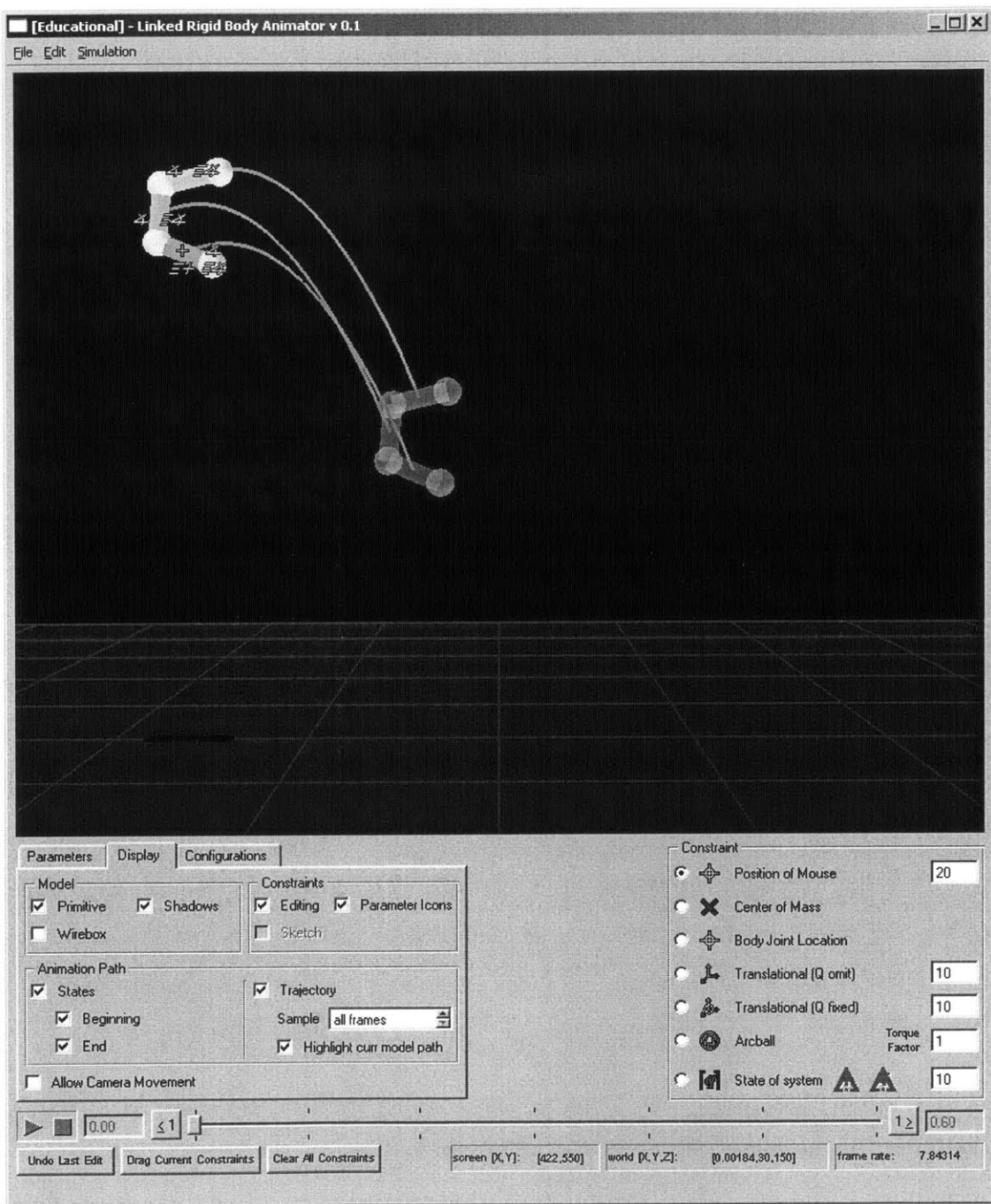


Figure 4-1: Screenshot of LRBME.

4.2 Object Selection

Ray tracing algorithms are used to determine whether the artist clicked on a body. Although more sophisticated algorithms could have been implemented, for simplicity the render window transforms the ray to the object’s frame before determining whether the ray pierces the objects’s bounding box. This is particularly convenient since the planes defining the bounding box volume are aligned with the axes of the objects’s reference frame.

The camera center (or *eye*) \mathbf{e} , and the ray direction \mathbf{d} compose the ray. The camera’s center is its location in the world frame: $\mathbf{e} = T_{\text{cam}}^{-1} \cdot [0, 0, 0, 1]^T$, where T_{cam} is the viewing matrix of the camera (Appendix B). The difference of the three-dimensional projection of the window coordinate at the near plane and the camera eye determines the ray direction.

Any point \mathbf{x} on the ray can be described by a scalar α such that $\mathbf{x} = \mathbf{e} + \alpha\mathbf{d}$. Since a point can be effectively represented by α , pairs of parallel bounding planes are used to iteratively update two scalar values representing the intersection of the nearest and farthest bounding planes with the ray. If the nearest plane scalar is larger than the farthest plane scalar, then the ray does not hit the volume. The algorithm, whose details are shown in Algorithm 4.2, is performed on all bodies in the scene to determine which one is closest to the camera.

4.3 Key Frame Constraints

The artist describes the desired motion with an interface that constrains the positions and orientations of the simulated object. She first scrubs through the motion to select the appropriate simulation time and then describes the constraint by either selecting a point on the body and dragging it to the desired location or by using an arcball widget to change the orientation of the body. In these cases the artist imposes a *mouse constraint* on the motion, since she provides continuous feedback through the mouse to properly describe the constraint. The editing loop is triggered when a mouse

Algorithm 4.2 Ray Intersection Algorithm

This algorithm is performed for each rigid body in the scene.

Inputs:

- \mathbf{e} — camera eye transformed into reference frame of primitive
- \mathbf{d} — ray direction transformed into reference frame of primitive
- l, r — left and right planes (parallel to $\hat{\mathbf{y}}\hat{\mathbf{z}}$ plane) of primitive bounding box
- t, b — top and bottom planes (parallel to $\hat{\mathbf{x}}\hat{\mathbf{z}}$ plane) of primitive bounding box
- f, k — front and back planes (parallel to $\hat{\mathbf{x}}\hat{\mathbf{y}}$ plane) of primitive bounding box

- 1: $\alpha_{\text{near}} \leftarrow -\infty$ {scalar for which $\mathbf{e} + \alpha_{\text{near}}\mathbf{d}$ reaches the furthest bounding plane}
 - 2: $\alpha_{\text{far}} \leftarrow \infty$ {scalar for which $\mathbf{e} + \alpha_{\text{far}}\mathbf{d}$ reaches the nearest bounding plane}
 - 3: **if** \mathbf{d} is not parallel to $\hat{\mathbf{y}}\hat{\mathbf{z}}$ plane **then**
 - 4: $\beta_{\text{near}} \leftarrow \min\left(\frac{l - e_x}{d_x}, \frac{r - e_x}{d_x}\right)$
 - 5: $\beta_{\text{far}} \leftarrow \max\left(\frac{l - e_x}{d_x}, \frac{r - e_x}{d_x}\right)$
 - 6: $\alpha_{\text{near}} \leftarrow \max(\alpha_{\text{near}}, \beta_{\text{near}})$
 - 7: $\alpha_{\text{far}} \leftarrow \min(\alpha_{\text{far}}, \beta_{\text{far}})$
 - 8: **end if**
 - 9: **if** \mathbf{d} is not parallel to $\hat{\mathbf{x}}\hat{\mathbf{z}}$ plane **then**
 - 10: $\gamma_{\text{near}} \leftarrow \min\left(\frac{t - e_y}{d_y}, \frac{b - e_y}{d_y}\right)$
 - 11: $\gamma_{\text{far}} \leftarrow \max\left(\frac{t - e_y}{d_y}, \frac{b - e_y}{d_y}\right)$
 - 12: $\alpha_{\text{near}} \leftarrow \max(\alpha_{\text{near}}, \gamma_{\text{near}})$
 - 13: $\alpha_{\text{far}} \leftarrow \min(\alpha_{\text{far}}, \gamma_{\text{far}})$
 - 14: **end if**
 - 15: **if** \mathbf{d} is not parallel to $\hat{\mathbf{x}}\hat{\mathbf{y}}$ plane **then**
 - 16: $\zeta_{\text{near}} \leftarrow \min\left(\frac{f - e_z}{d_z}, \frac{k - e_z}{d_z}\right)$
 - 17: $\zeta_{\text{far}} \leftarrow \max\left(\frac{f - e_z}{d_z}, \frac{k - e_z}{d_z}\right)$
 - 18: $\alpha_{\text{near}} \leftarrow \max(\alpha_{\text{near}}, \zeta_{\text{near}})$
 - 19: $\alpha_{\text{far}} \leftarrow \min(\alpha_{\text{far}}, \zeta_{\text{far}})$
 - 20: **end if**
 - 21: **if** $\alpha_{\text{near}} < \alpha_{\text{far}}$ **then**
 - 22: The ray hits the bounding box volume
 - 23: **else**
 - 24: The ray does not hit the bounding box volume
 - 25: **end if**
-

constraint is placed on the motion.

The artist can also add *nail constraints*, which enforce a particular state, and allow her to manipulate the motion at several different simulation times. Because nail constraints are off-line, they usually are not enforced when the artist does not engage the editing loop with a mouse constraint. It is possible, however, that after modification of the motion through a mouse constraint, the objects will drift from the positions described by the nail constraints. The user interface provides a simple button that engages the editing loop for all nail constraints, thereby not requiring the artist to add a mouse constraint.

The artist uses tools that emulate key framing techniques to specify the constraints. These techniques can be enforced by use of either a mouse (on-line) or a nail (off-line) constraint. *Point constraints* allow the artist to select a point on a body and drag it to its desired location in the world frame. Even though LRBME does not actually use inverse kinematic (IK) techniques, this type of constraint emulates IK since the artist, only moving the end-effector, expects the system to solve for the joint angles [34]. *State constraints* allow the artist to specifically fix or alter parameters in the state vector. These constraints emulate forward kinematics tools, where the artist dictates the desired location and angles of the object.

In the case of the point and translational state constraints, the artist selects a point on or in a body and drags it to its desired location. Since she is trying to specify a three-dimensional point on a two-dimensional screen, the point is constrained to move in a plane. In LRBME, she may select to move the point on the plane which contains the point and is parallel to either the camera view or the $\hat{\mathbf{x}}\hat{\mathbf{z}}$ plane.

The rest of this section describes the basic algorithm and mathematics for each tool.

4.3.1 Basic Algorithm

Each constraint i is placed at some unique time instant t_i during the animation. During the editing loop, the user interface iterates through all the key frame constraints and obtains from each of them the aspect vector change $\delta\mathbf{k}_i$ and the Jacobian

Constraint	Type	$\ \mathcal{K}(\mathbf{y})\ $
Body surface	point	3
Joint location	point	3
Body center-of-mass	point	3
Translational	state	3
Translational, fixed quaternion	state	$3 + n_\theta$
Arcball	state	1, 2, or 4

Figure 4-2: **Key Frame Constraint Types.** The number n_θ represents the number of angle parameters in the state vector. The number of elements in an arcball key frame constraint depends on the type of joint being manipulated (see Figure 3-2).

$$\partial\mathcal{K}_i/\partial\mathcal{S}(t_i, \mathbf{p}).$$

The size of each $\delta\mathbf{k}_i$ depends on the aspect being changed. If the artist wants a point on the object to be located at a specific point in the world frame, \mathcal{K} returns a three-element vector. If the angle of a pin joint should be at a specific value, \mathcal{K} is a vector of one element. Figure 4-2 shows a chart of the key frame constraints available for use.

Scaling factors are associated with each type to give a uniform feeling of change across all key frame constraint types during interactive editing (Figure 4-3). The artist using her mouse to enforce a point constraint, for example, should have as much of an effect on the motion as enforcing an arcball constraint. While mouse constraints of the same type have the same scaling factor, each nail constraint has its own individual factor. Consequently, the artist may change these scaling factors to make some nail constraints more important than others. For example, if the artist places nail constraints on the beginning and end frames, but decides the latter constraint was more important to enforce, she could increase its scaling factor.

4.3.2 Point Constraints

A point constraint allows the artist to adjust the location of a point on or in the object. She may select a body’s center of mass, a point on its surface, or the location of its inboard joint. The aspect function $\text{Pos}(\cdot)$ computes the world coordinate \mathbf{c}^W of

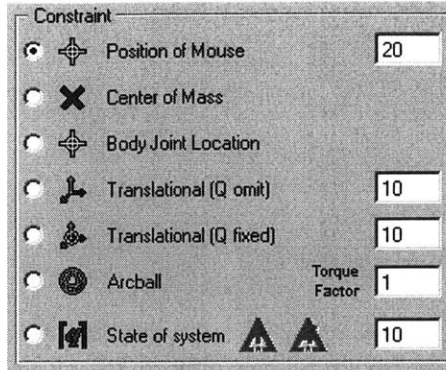


Figure 4-3: **Constraint Strength Group Box.** The relative strength of each constraint type is listed on the right-hand side. Center-of-mass and body joint location constraints have the same weight as the body surface constraint, since all three are point constraints.

a point with body coordinates \mathbf{c}^B :

$$\text{Pos}(\mathbf{y}, \mathbf{c}^B) = \mathbf{c}^W$$

The artist can use this constraint to drag the point through a mouse constraint or to nail it in place with a nail constraint. The aspect vector change $\delta\mathbf{k}$ is the difference of the desired world point $\bar{\mathbf{c}}^W$ and the current world point location of \mathbf{c}^B :

$$\delta\mathbf{k} = s_{\text{pc}} (\bar{\mathbf{c}}^W - \text{Pos}(\mathbf{y}, \mathbf{c}^B))$$

where s_{pc} is the scaling factor of the point constraint. The aspect function's dependence on only the generalized coordinates \mathbf{q} leads to a Jacobian matrix $\partial\mathcal{K}/\partial\mathcal{S}$ where sub-matrix $\partial\text{Pos}/\partial\mathbf{u}$ is $\begin{bmatrix} \mathbf{0} \end{bmatrix}$:

$$\frac{\partial\mathcal{K}}{\partial\mathcal{S}(t, \mathbf{p})} = \left[\begin{array}{c} \frac{\partial\text{Pos}(\mathbf{y})}{\partial\mathbf{q}} \\ \mathbf{0} \end{array} \right] \begin{array}{c} n_{\mathbf{u}} \\ \mathbf{0} \end{array}$$

where $n_{\mathbf{u}}$ represents the number of generalized velocities.

Since this constraint is concerned with only a point, LRBME is left to find the appropriate changes to the control vector and the generalized state that gets the point to the desired location. Point constraints are most useful in a situation when

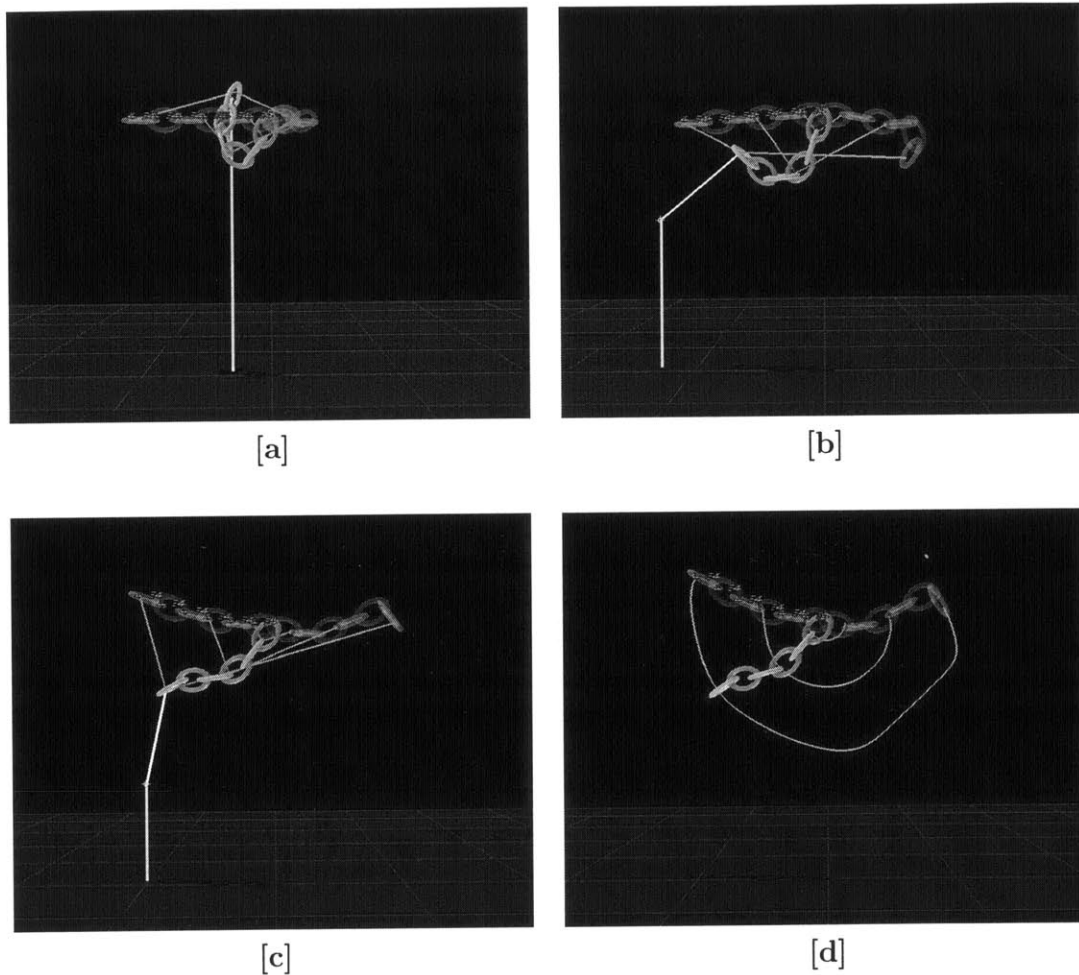


Figure 4-4: **Point constraint.** The artist simplifies the motion of a chain by adding a point constraint. [a]–[c] The center of mass of the last link is selected, and dragged toward the floor. Sampling the motion at only certain designated frames during the editing loop makes the trajectory appear incorrect (Section 5.8). [d] Once the constraint is released, however, the simulation samples the motion at uniform intervals.

an end-effector is selected and the artist does not wish to explicitly constrain joint angles (Figure 4-4).

4.3.3 State Constraints

The state constraint manipulates specific elements of the generalized coordinates, which are extracted from the state vector by the aspect function. State constraints can isolate the root position of the joint hierarchy or the orientation of each joint. The benefit of this constraint is that it separates the kinematic task, which computes joint configurations from desired positions of the end effector, and the dynamic task, which computes the simulation parameters.

Translational Constraint

The translational constraint allows the artist to alter the root position, shifting the entire object to the desired location. When using this constraint, she has the option of omitting or locking the joint angles of the object (Figure 4-5). Omitting angles implies that during editing the solver is free to alter the joint angles as needed to satisfy the translational constraint. If the artist chooses to lock the joint angles during editing, LRBME attempts to preserve the object's spatial configuration.

The aspect function simply returns the appropriate elements of the state vector to be preserved or altered:

$$\begin{aligned} \text{With omitted joint angles: } \mathcal{K}(\mathbf{y}) &= \mathcal{K} \left(\left[\mathbf{x}^T, \boldsymbol{\theta}^T, \mathbf{v}^T, \boldsymbol{\omega}^T \right]^T \right) = \mathbf{x} \\ \text{With locked joint angles: } \mathcal{K}(\mathbf{y}) &= \left[\mathbf{x}^T, \boldsymbol{\theta}^T \right]^T \end{aligned}$$

The artist selects a point \mathbf{c} on the surface of the root body, and drags it to the desired location $\bar{\mathbf{c}}$. The difference becomes $\delta\mathbf{k}$ for translational constraints with omitted joint angles:

$$\delta\mathbf{k} = \bar{\mathbf{c}} - \mathbf{c}$$

For locked joint angles, $\delta\mathbf{k}$ is extended with zeros, to indicate that the desired

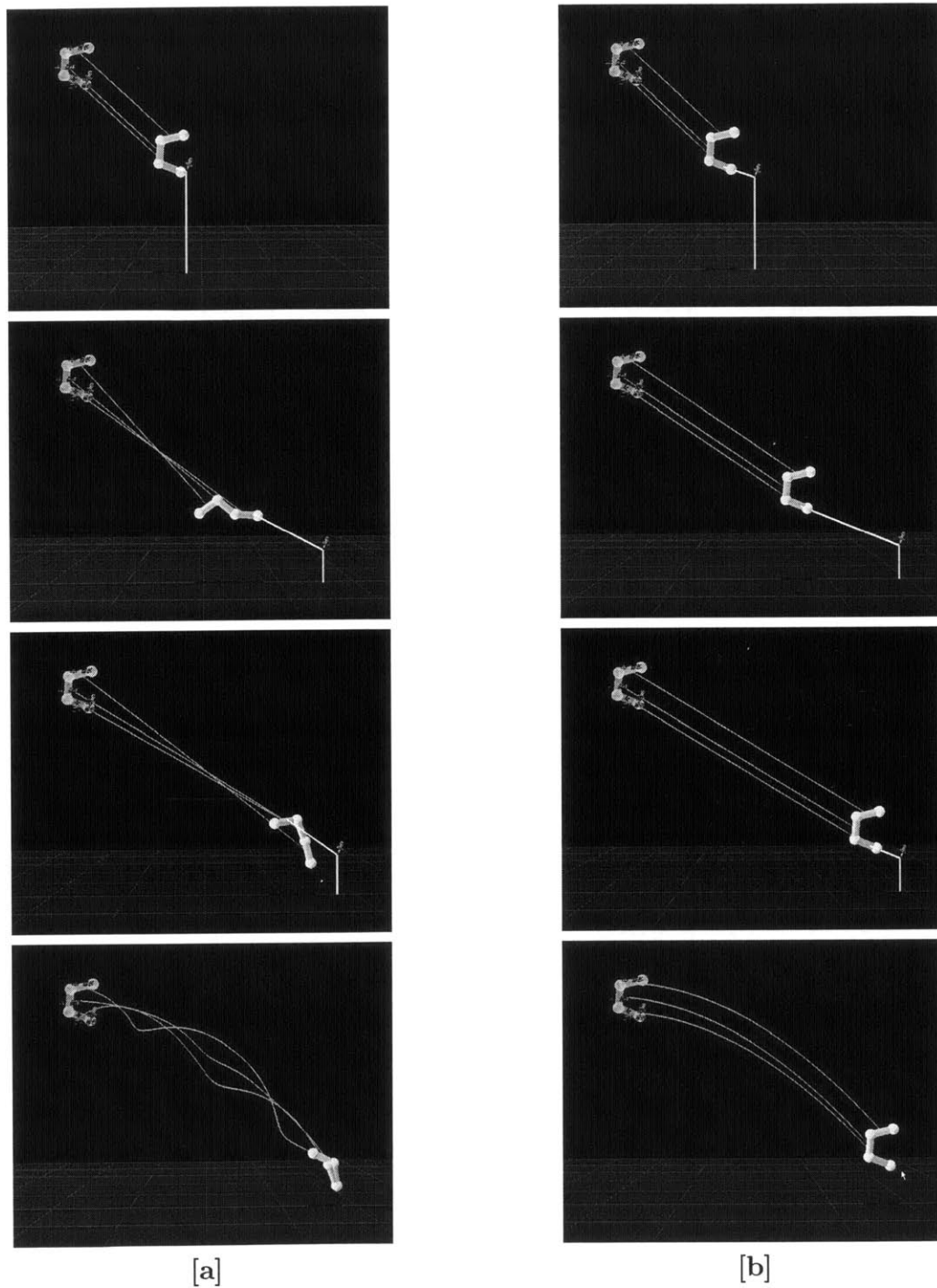


Figure 4-5: **Translational constraint.** The artist fixes the object's location and joint angles at the initial frame, and attempts to move the object further to the right in the last frame. [a] A translational constraint with omitted quaternions is placed on the motion. Although the object does approach the desired position, the joint angles change, as expected. [b] A translational constraint with fixed quaternions is used. Here the joint angles are preserved, and only the location of the root point changes.

changes in all the joint angles are 0:

$$\delta \mathbf{k} = \left[(\bar{\mathbf{c}} - \mathbf{c})^T, \overbrace{0, 0, \dots, 0}^{n_\theta \text{ elements}} \right]^T$$

Since the aspect function merely extracts the proper state vector elements to constrain, $\partial \mathcal{K} / \partial \mathcal{S}$ maps the elements of \mathcal{K} to their counterparts in the state vector:

$$\text{For omitted angles: } \frac{\partial \mathcal{K}}{\partial \mathcal{S}} = \left[\frac{\partial \mathcal{K}}{\partial \mathbf{x}} \left| \frac{\partial \mathcal{K}}{\partial \theta} \quad \frac{\partial \mathcal{K}}{\partial \mathbf{v}} \quad \frac{\partial \mathcal{K}}{\partial \boldsymbol{\omega}} \right. \right] = \left[\begin{array}{c|c} 3 & n_y - 3 \\ \hline 3 \begin{bmatrix} \mathbf{1} \end{bmatrix} & 3 \begin{bmatrix} \mathbf{0} \end{bmatrix} \end{array} \right]$$

$$\text{For locked angles: } \frac{\partial \mathcal{K}}{\partial \mathcal{S}} = \left[\begin{array}{c|c} n_a & n_u \\ \hline n_a \begin{bmatrix} \mathbf{1} \end{bmatrix} & n_a \begin{bmatrix} \mathbf{0} \end{bmatrix} \end{array} \right]$$

Arcball Constraint

The arcball constraint can only be applied online. After the artist selects a body to rotate, a form of Shoemake's arcball appears over the joint [28]. If the joint is a ball joint, three arcs appear on the arcball, one for each principal axis in the body's reference frame. If the joint is a pin or universal joint, arcs for each rotation axis are displayed. The artist can select an arc, and rotate the object along the arc. Mouse movements are modeled as torque forces on the arcball, and the joint's rate of rotational change depends on the torque as well as the arcball's scaling factor. Algorithm 4.3 provides pseudocode for finding $\delta \mathbf{k}$, and Figure 4-6 shows a typical session using the arcball constraint. The projection matrix $\partial \mathcal{K} / \partial \mathcal{S}$ is similar to that of the translational constraint.

4.4 Adjusting the Control Vector

Before the control vector is adjusted by $\delta \mathbf{p}$, the artist can specify how the system should change the parameters. To do this, she specifies the relative scaling between *attributes* before commencing the editing loop (Figure 4-7). The four main attributes are position, linear velocity, rotation, and angular velocity. Changing the relative

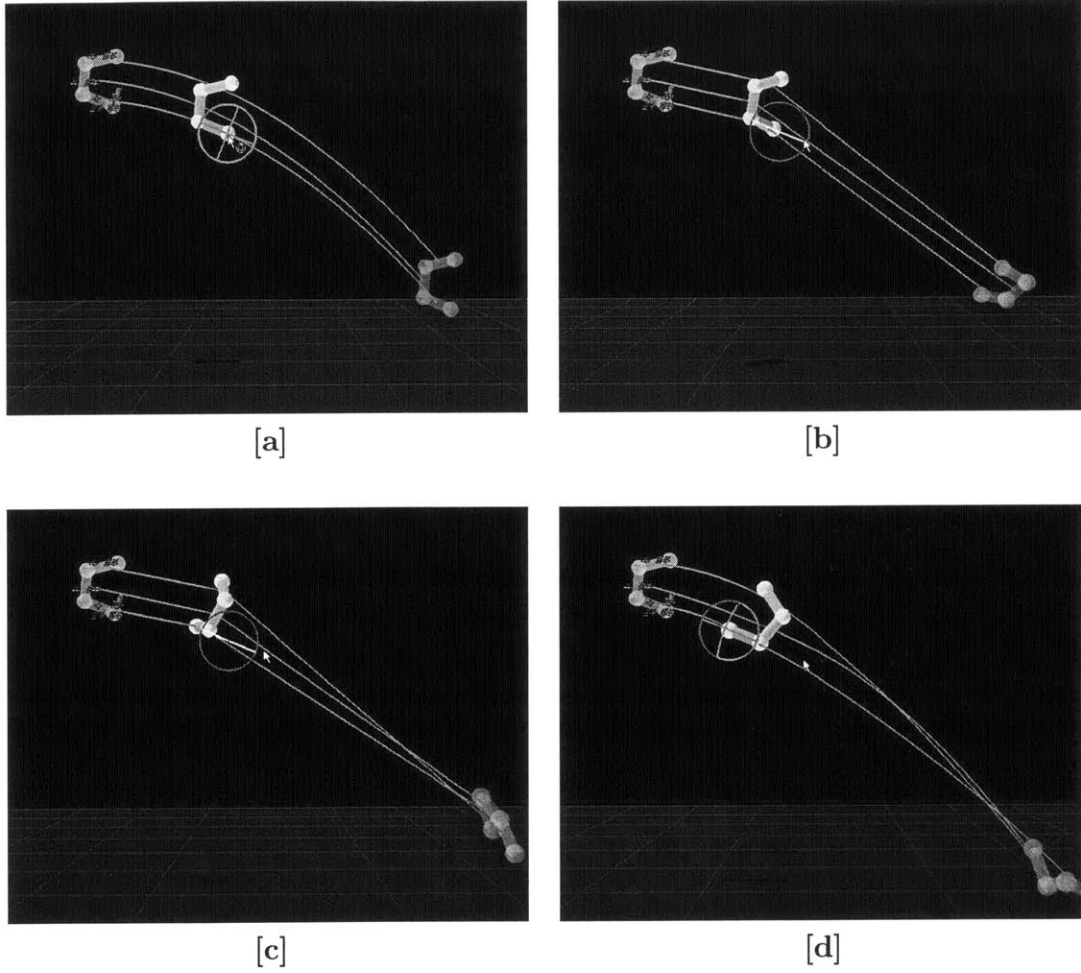


Figure 4-6: **Arcball constraint.** The artist fixes the location and joint angles of the object in the initial frame. [a] She changes the orientation of the object in mid-flight by manipulating the 3 DOF joint on the root body. [b]–[c] Because the location of the root point is omitted from $\delta\mathbf{k}$, the object drifts during editing. The arcball does not move with the joint during editing because the rotational change desired is dependent on the mouse’s relative location with the arcball. [d] When the artist releases the mouse button, the arcball is re-located to the proper position.

Algorithm 4.3 Arcball Constraint Algorithm

Returns aspect change $\delta\mathbf{k}$ **Inputs:** θ_t — the current body rotation in world frame (quaternion) s_τ — arcball constraint scaling (torque) factor

```
1:  $\omega_l \leftarrow [0, 0, 0]$  { $\omega_l$  represents direction of rotation change}
2: while mouse button is down do
3:    $\bar{\theta} \leftarrow$  desired quaternion rotation, obtained through arcball
4:    $\Delta\theta \leftarrow \bar{\theta} \circ \theta_t^{-1}$ 
5:   if joint is pin or universal joint then
6:     Express  $\bar{\theta}$  and  $\Delta\theta$  in body rather than world reference frame
7:   end if
8:   if  $\Delta\theta$  is not close to the identity quaternion then
9:     if  $\|\text{Vector}(\Delta\theta)\|$  is close to 0 then
10:       $\omega \leftarrow \|\text{Vector}(\Delta\theta)\| \cdot \text{Vector}(\Delta\theta)$ 
11:       $\omega_o \leftarrow -\|\text{Vector}(\Delta\theta)\| \cdot \text{Vector}(\Delta\theta)$ 
12:    else
13:       $\omega \leftarrow \frac{2 \cos^{-1}(\Delta\theta_r)}{\|\text{Vector}(\Delta\theta)\|} \cdot \text{Vector}(\Delta\theta)$ 
14:       $\omega_o \leftarrow \frac{-2 \cos^{-1}(-\Delta\theta_r)}{\|\text{Vector}(\Delta\theta)\|} \cdot \text{Vector}(\Delta\theta)$  { $\theta$  and  $-\theta$  are the same rotation}
15:    end if
16:    if  $\|\omega_l - \omega\| < \|\omega_l - \omega_o\|$  then { $\omega_l$  is iteratively updated}
17:       $\omega_l \leftarrow \omega$ 
18:    else
19:       $\omega_l \leftarrow \omega_o$ 
20:    end if
21:    if joint is 3 DOF ball joint then
22:       $\delta\mathbf{k} \leftarrow (s_\tau \omega_l) \circ \theta_t$ 
23:    else {joint is described by angle about axis rather than quaternion}
24:       $\theta \leftarrow$  angle about body axis of rotation of current state
25:       $\bar{\theta} \leftarrow$  angle about body axis of rotation of arcball
26:       $\Delta\theta \leftarrow \bar{\theta} - \theta$ 
27:       $\Delta\theta^* \leftarrow -\text{sign}(\Delta\theta) |2\pi - |\Delta\theta||$  {rotation in opposite direction}
28:      if  $\omega_l =$  axis of rotation then
29:         $\delta\mathbf{k} = s_\tau \max(\Delta\theta, \Delta\theta^*)$ 
30:      else { $\omega_l$  is opposite to the axis of rotation}
31:         $\delta\mathbf{k} = s_\tau \min(\Delta\theta, \Delta\theta^*)$ 
32:      end if
33:    end if
34:  end if
35: end while
```

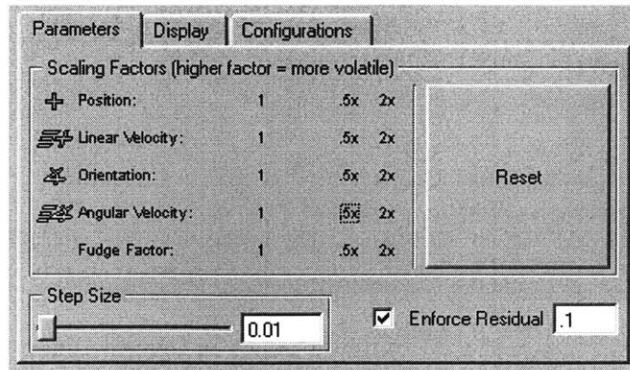


Figure 4-7: **Control Vector Attribute Mass Group Box.** Here the artist can specify relative weights of attributes by pressing the ‘.5x’ and ‘2x’ buttons, which halves or doubles the attribute’s weight, respectively. The weight assigned to an attribute affects all elements in the control vector related to that attribute. Step size ϵ is determined by the slider on the left-hand side, used for differential update. The “Enforce Residual” box on the lower right-hand side provides a threshold of error above which the editing loop halts. More about the residual is discussed in Section 5.5.

scaling of an attribute affects all elements in the control vector associated with that attribute. Section 5.2 explains how the relative scales are incorporated into solving for $\delta\mathbf{p}$.

To leave a parameter in the control vector unchanged, the artist has two options. She can scale the parameter’s attribute down to a very low value such that it hardly gets altered. Alternately, she can turn the parameter off completely, effectively making its relative scale 0. Turning a parameter off reduces the control vector size, which can help the editing loop maintain interactive speeds. At the same time, fewer DOFs are available for the solver module to change, and finding a suitable adjustment to the control vector to satisfy the key frame constraints may become more difficult. Section 5.4 discourses how these changes affect the editing process.

On each body, small icons are overlaid on a silhouette of the object’s initial state to denote which attributes of that body may be altered during editing (Figure 4-8). The artist toggles alteration of attributes by holding down modifier keys and directly clicking on the body. This interface allows her to dictate which attributes of which bodies to change visually.

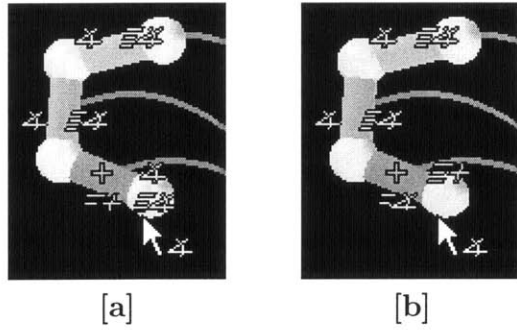
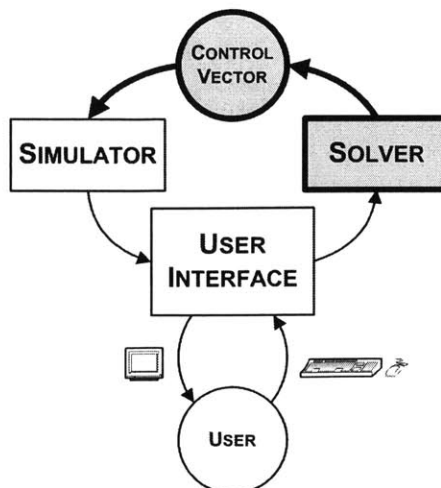


Figure 4-8: **Toggling Control Vector Parameters.** Icons denoting attributes associated with each joint are overlaid on top of the inboard body. In this example, the bottom body is clearly the root body since it contains all four attributes (left). The other connected bodies can only have orientation and angular velocity attributes. The artist decides that she wants to keep the root body's orientation fixed during editing, so she clicks on the body to turn that attribute off (right).

Chapter 5

Editing Motion



For a set of aspect vectors, differential control computes the corresponding parameter adjustment as a solution to a quadratic program:

$$\begin{aligned} \min_{\delta \mathbf{p}} \quad & \frac{1}{2} \delta \mathbf{p}^T M \delta \mathbf{p} \quad \text{such that} \\ \delta \mathbf{k}_1 &= \frac{\partial \mathcal{K}_1}{\partial \mathbf{p}} \delta \mathbf{p} \\ \delta \mathbf{k}_2 &= \frac{\partial \mathcal{K}_2}{\partial \mathbf{p}} \delta \mathbf{p} \\ & \vdots \\ \delta \mathbf{k}_n &= \frac{\partial \mathcal{K}_n}{\partial \mathbf{p}} \delta \mathbf{p} \end{aligned} \tag{5.1}$$

where n is the number of key frame constraints, and the diagonal matrix M scales the units of simulation parameters [25, 16] (Section 5.2). While still satisfying the key frame constraints, the objective function seeks the smallest change to the control vector and the current state of the object. The linear constraints require the computation of sensitivity Jacobian matrices $\Phi = \partial \mathcal{S} / \partial \mathbf{p}$, which define locally linear approximations of the relationship between the state vector \mathbf{y} and the simulation parameters \mathbf{p} . The solution of the quadratic program computes the parameters for a new simulation, after which the system displays the simulated paths and repeats the entire process.

5.1 Jacobian of Motion

Formulating the quadratic program first involves calculating the sensitivity matrix Φ of the simulation function:

$$\begin{aligned}\Phi = \frac{\partial S(t_i, \mathbf{p})}{\partial \mathbf{p}} &= \frac{\partial}{\partial \mathbf{p}} \left(\mathbf{y}_{t_0}(\mathbf{p}) + \int_{t=t_0}^{t_i} \mathcal{F}(t, \mathbf{y}(t), \mathbf{p}) dt \right) \\ &= \frac{d\mathbf{y}_{t_0}(\mathbf{p})}{d\mathbf{p}} + \int_{t=t_0}^{t_i} \frac{\partial \mathcal{F}(t, \mathbf{y}(t), \mathbf{p})}{\partial \mathbf{p}} dt\end{aligned}\quad (5.2)$$

When simulating a passive object without non-conservative forces, $\mathbf{y}_{t_0} = \mathbf{p}$, and so $d\mathbf{y}_{t_0}/d\mathbf{p} = [\mathbf{1}]$ (the identity matrix). To derive the linear constraints for the quadratic program shown in Equation 5.1, the matrix $\partial\mathcal{K}/\partial\mathbf{y}$ is obtained from each constraint, as discussed in Section 4.3.

The differential approach hinges on the efficient computation of Φ . A straightforward approximation with forward differences is insufficient, as errors compound throughout numerical integration to yield unstable convergence with the differential technique. Instead, automatic differentiation is used to avoid making any compromises.

Differentiating the equations of motion shown in Equation 3.1 with respect to \mathbf{p} yields a system of differential equations that describe the evolution of the sensitivity matrix:

$$\frac{d}{dt}\Phi(t) = \frac{\partial}{\partial \mathbf{p}}\mathcal{F}(t, \mathbf{y}(t), \Phi(t), \mathbf{p})\quad (5.3)$$

Since no closed-form expression exists for \mathcal{F} , the most difficult term to compute in Equation 5.3 is the derivative $\partial\mathcal{F}/\partial\mathbf{p}$. Fortunately, automatic differentiation simplifies this task even for complex equations of linked rigid-body dynamics. In implementation, the ADIFOR software package [4] computes the derivatives of complex FORTRAN subroutines generated by SD/FAST. As such, the simulation module integrates Equations 3.1 and 5.3 simultaneously during simulation.

5.2 Scaling Attributes

In Equation 5.1, the incorporation of a *mass matrix* allows the artist to instruct the system to favor changing one attribute of the control vector over another. Without it, all elements of the control vector, regardless of attribute, will equally reduce the value of the objective function. Generally, with no mass matrix (or one that is equal to the identity matrix), the solver module ends up changing angle-related parameters more drastically than position-related parameters. This happens particularly when the artist uses point constraints (Figure 5-1).

Two factors contribute to this behavior. Firstly, the state and control vectors of a linked rigid body generally have more angle-related parameters than position-related ones. Since the solver module treats all parameters equally when minimizing the objective function, the larger percentage of angle-related parameters already implies that most of the adjustments made to the control vector will be angle-related. Secondly, linear and angular units are incomparable. One radian or one radian per second change in the initial parameters has much larger effect on the motion than one linear unit or one unit per second change. Since smaller adjustments to the angle-related parameters have greater influence over the entire motion and lead to smaller objective function evaluations, the solver module will tend to change those parameters.

To compensate for the different units, the artist specifies weights which denote the importance of attributes through the interface discussed in Section 4.4. The higher the weight number, the more volatile the attribute becomes and the more likely the solver module will alter the attribute. The values of the weights are mapped onto a $n_{\mathbf{p}} \times n_{\mathbf{p}}$ *mass matrix* M^1 which has along its diagonal the weight attributed to the

¹Although M^{-1} is used to find the optimal Lagrange multipliers (Section 5.3), M itself is never used. In the implementation, the relative weights specified by the artist represent the values in M^{-1} to avoid the cost of inverting M .

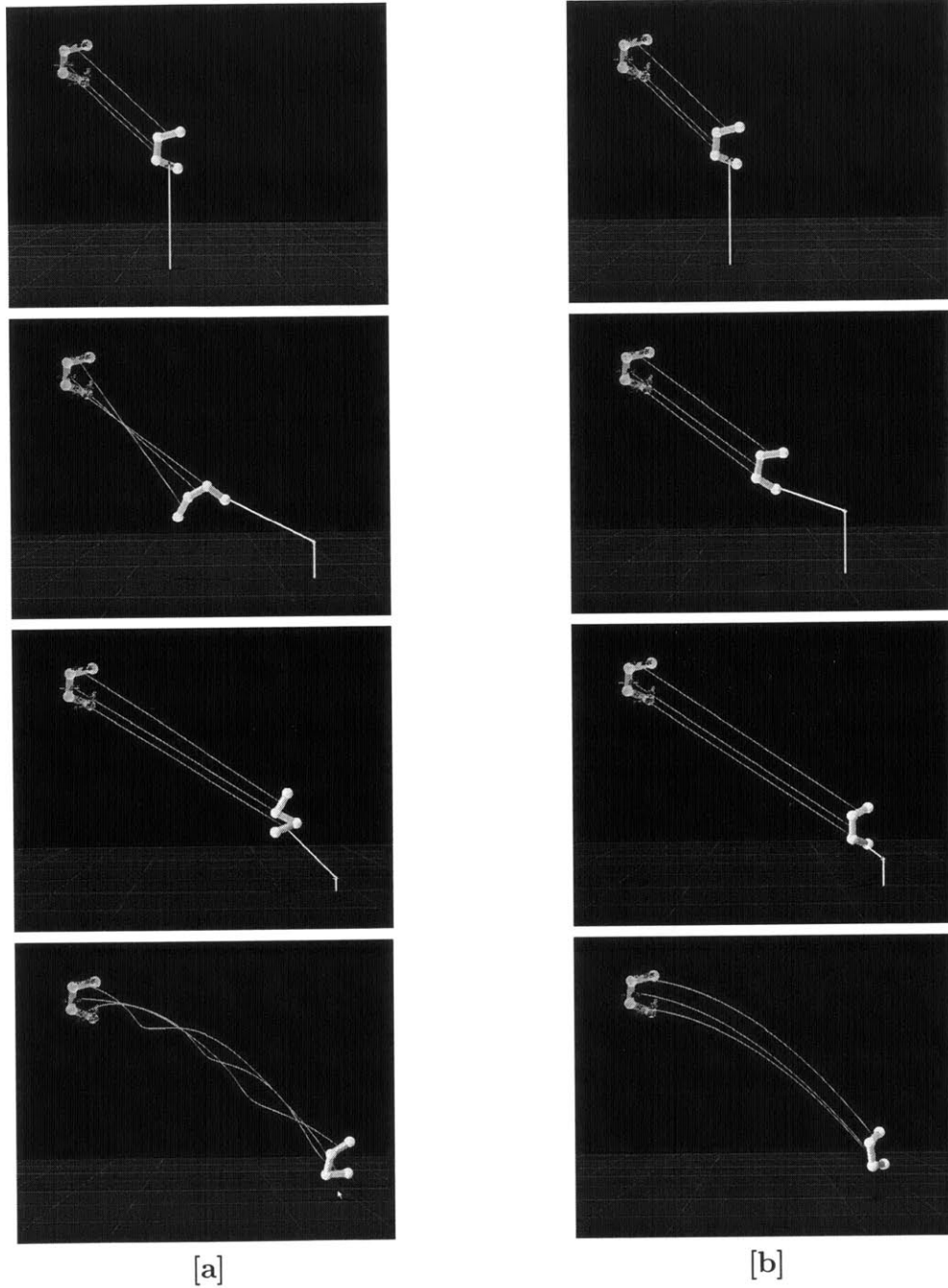


Figure 5-1: **Comparison of system behavior with residual calculation.** Keeping the initial frame fixed, the artist wants the figure to arrive at a location further to the right. [a] The artist edits without the mass matrix. Consequently, many different attributes are changed all at once, adding motion that the artist may not necessarily want. [b] Before editing, the artist sets the linear velocity scaling factor much higher (256:1), indicating that the system should try to change the linear velocity more than the other attributes.

corresponding elements of the state vector:

$$M = \begin{bmatrix} m_{11} & 0 & 0 & 0 \\ 0 & m_{22} & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & m_{n_p n_p} \end{bmatrix}$$

Since only attributes can have weights and not individual elements, all position-related elements have the same weight, and all angular elements, regardless of whether they are part of a quaternion, have the same relative weight.

5.3 Finding Optimal Change

To find the optimal change to the control vector, $\delta \mathbf{p}^*$, the objective function in Equation 5.1 is converted into the Lagrangian $L(\delta \mathbf{p}, \boldsymbol{\lambda})$:

$$\begin{aligned} L(\delta \mathbf{p}, \boldsymbol{\lambda}) &= \frac{1}{2} \delta \mathbf{p}^T M \delta \mathbf{p} - \boldsymbol{\lambda}_1 \cdot \left(\delta \mathbf{k}_1 - \frac{\partial \mathcal{K}_1}{\partial \mathbf{p}} \delta \mathbf{p} \right) - \\ &\quad \boldsymbol{\lambda}_2 \cdot \left(\delta \mathbf{k}_2 - \frac{\partial \mathcal{K}_2}{\partial \mathbf{p}} \delta \mathbf{p} \right) - \dots - \boldsymbol{\lambda}_n \cdot \left(\delta \mathbf{k}_n - \frac{\partial \mathcal{K}_n}{\partial \mathbf{p}} \delta \mathbf{p} \right) \\ &= \frac{1}{2} \delta \mathbf{p}^T M \delta \mathbf{p} - \boldsymbol{\lambda} \cdot (\delta \mathbf{k} - A \delta \mathbf{p}) \end{aligned} \quad (5.4)$$

where $\boldsymbol{\lambda} = [\boldsymbol{\lambda}_1^T \ \boldsymbol{\lambda}_2^T \ \dots \ \boldsymbol{\lambda}_n^T]^T$, $\delta \mathbf{k} = [\delta \mathbf{k}_1^T \ \delta \mathbf{k}_2^T \ \dots \ \delta \mathbf{k}_n^T]^T$, and A is the juxtaposition of all the $\partial \mathcal{K}_i / \partial \mathbf{p}$ Jacobian matrices:

$$A = \begin{bmatrix} \left[\frac{\partial \mathcal{K}_1}{\partial \mathbf{p}} \right] \\ \left[\frac{\partial \mathcal{K}_2}{\partial \mathbf{p}} \right] \\ \vdots \\ \left[\frac{\partial \mathcal{K}_n}{\partial \mathbf{p}} \right] \end{bmatrix}$$

To find the local optimum, $\nabla L(\delta \mathbf{p}, \boldsymbol{\lambda})$ is set to 0. To clarify derivation of the gradient,

Equation 5.4 is rewritten in summation notation:

$$\begin{aligned}
L(\delta\mathbf{p}, \boldsymbol{\lambda}) &= \frac{1}{2}\delta\mathbf{p}^T M\delta\mathbf{p} - \boldsymbol{\lambda} \cdot (\delta\mathbf{k} - A\delta\mathbf{p}) \\
&= \frac{1}{2} \sum_{i=1}^{n_{\mathbf{p}}} m_{ii}\delta p_i^2 - \sum_{i=1}^{n_{\mathbf{k}}} \lambda_i \delta k_i - \sum_{i=1}^{n_{\mathbf{k}}} \lambda_i \sum_{j=1}^{n_{\mathbf{p}}} a_{ij}\delta p_j
\end{aligned} \tag{5.5}$$

where a_{ij} refers to the $(i, j)^{\text{th}}$ element of the Jacobian matrix A , $n_{\mathbf{k}}$ is the sum of the sizes of all aspect vectors, and $n_{\mathbf{p}}$ is the size of the control vector.

The gradient of the Lagrangian in Equation 5.5 becomes:

$$\begin{aligned}
\nabla L(\delta\mathbf{p}, \boldsymbol{\lambda}) &= \begin{bmatrix} m_{11}\delta p_1 - \sum_{i=1}^{n_{\mathbf{k}}} \lambda_i a_{i1} \\ m_{22}\delta p_2 - \sum_{i=1}^{n_{\mathbf{k}}} \lambda_i a_{i2} \\ \vdots \\ m_{n_{\mathbf{p}}n_{\mathbf{p}}}\delta p_{n_{\mathbf{p}}} - \sum_{i=1}^{n_{\mathbf{k}}} \lambda_i a_{in_{\mathbf{k}}} \end{bmatrix} \\
&= M\delta\mathbf{p} - A^T \boldsymbol{\lambda}
\end{aligned}$$

Setting $\nabla L(\delta\mathbf{p}, \boldsymbol{\lambda})$ to $\mathbf{0}$ leads to the minimal change $\delta\mathbf{p}^*$ and optimal Lagrange multipliers $\boldsymbol{\lambda}^*$:

$$\delta\mathbf{p}^* = M^{-1}A^T \boldsymbol{\lambda}^* \tag{5.6}$$

To find $\delta\mathbf{p}^*$, Equation 5.6 is incorporated into Equation 4.1:

$$\frac{\partial \mathcal{K}}{\partial \mathbf{p}} \delta\mathbf{p}^* = A\delta\mathbf{p}^* = AM^{-1}A^T \boldsymbol{\lambda}^* = \delta\mathbf{k} \tag{5.7}$$

5.4 Changing Control Vector Format

Some artist-driven settings may necessitate altering the format of the control vector before calculating the optimization problem discussed in Section 5.3. Changing the format of the control vector changes little of the actual editing process as long as the Jacobian of the control vector with respect to the re-formatted control vector can be calculated.

5.4.1 Masking

As mentioned in Section 4.4, the artist may decide to not alter certain elements of the control vector. To reflect this mathematically, those elements are eliminated from the control vector before the control problem is derived. In particular, the control vector \mathbf{p} is converted to a *masked* version of itself, \mathbf{p}_M :

$$\left(\mathbf{p}^T \frac{d\mathbf{p}}{d\mathbf{p}_M} \right)^T = \mathbf{p}_M$$

Masking matrix $d\mathbf{p}/d\mathbf{p}_M$ is created dynamically when the artist toggles the attribute of a body.

Containing only the elements that are allowed to be changed by Equation 4.2, the masked control vector replaces the complete control vector in the editing process, and thereby modifies the optimization problem stated in Equation 5.1 to:

$$\min_{\delta\mathbf{p}_M} \frac{1}{2} \delta\mathbf{p}_M^T M_M \delta\mathbf{p}_M, \quad \text{such that } \frac{\partial\mathcal{K}}{\partial\mathbf{p}_M} \delta\mathbf{p}_M = \partial\mathbf{k} \quad (5.1a)$$

where $M_M = (d\mathbf{p}/d\mathbf{p}_M)^T M (d\mathbf{p}/d\mathbf{p}_M)$.

Three adjustments are made to the editing process as a result of using the masked control vector. Firstly, the rows and columns of the Jacobian associated with the unaltered elements are eliminated, quickening computation. To conform to the Lagrangian, the matrix Φ which originally contained the $\partial\mathcal{K}_i/\partial\mathbf{p}$ matrices is instead populated with $\partial\mathcal{K}_i/\partial\mathbf{p}_M$. An expansion of the term via the chain rule shows that calculating $\partial\mathcal{K}/\partial\mathbf{p}_M$ requires only post-multiplying the masking matrix:

$$\frac{\partial\mathcal{K}}{\partial\mathbf{p}_M} = \frac{\partial\mathcal{K}}{\partial\mathbf{p}} \frac{d\mathbf{p}}{d\mathbf{p}_M}$$

Secondly, the mass matrix M is replaced with M_M in Equations 5.6 and 5.7 when solving for $\delta\mathbf{p}_M^*$. Finally, once $\delta\mathbf{p}_M^*$ is found, the masking matrix is used again to

convert $\delta\mathbf{p}_M$ to $\delta\mathbf{p}$, which Equation 4.2 can use to alter the complete control vector:

$$\mathbf{p}_{\text{new}} = \mathbf{p}_{\text{old}} + \epsilon \cdot \frac{d\mathbf{p}}{d\mathbf{p}_M} \delta\mathbf{p}_M$$

5.4.2 Re-parameterizing Quaternions

Modifying the control vector through linear gradient descent becomes problematic when dealing with ball joints, since they are represented by quaternions. To address this, the quaternions $\boldsymbol{\theta}$ of $\delta\mathbf{p}_M$ are *logarithmically-mapped* to three-element vectors \mathbf{v}_θ , resulting in a *re-parameterized* control vector $\delta\mathbf{p}_R$. The modifications made to the quadratic program to accommodate solving for $\delta\mathbf{p}_R$ are similar to that done by the masked control vector $\delta\mathbf{p}_M$. The quadratic program stated in Equation 5.1a is modified to:

$$\min_{\delta\mathbf{p}_R} \frac{1}{2} \delta\mathbf{p}_R^T M_R \delta\mathbf{p}_R \quad \text{such that} \quad \frac{\partial \mathcal{K}}{\partial \mathbf{p}_R} \delta\mathbf{p}_R = \partial \mathbf{k}$$

where $M_R = (d\mathbf{p}_M/d\mathbf{p}_R)^T M_M (d\mathbf{p}_M/d\mathbf{p}_R)$. Expanding $\partial \mathcal{K}/\partial \mathbf{p}_R$ leads to:

$$\frac{\partial \mathcal{K}}{\partial \mathbf{p}_R} = \frac{\partial \mathcal{K}}{\partial \mathbf{p}_M} \frac{d\mathbf{p}_M}{d\mathbf{p}_R}$$

Since the only difference between $\delta\mathbf{p}_M$ and $\delta\mathbf{p}_R$ is the quaternion parameters, the Jacobian matrix $d\mathbf{p}_M/d\mathbf{p}_R$ is mostly an identity matrix, with the matrix $\partial\boldsymbol{\theta}/\partial\mathbf{v}_\theta$ inserted in the appropriate rows and columns.

The logarithmically-mapped quaternions are stored and updated with each successive iteration of the editing loop. To prevent singularities, the logarithmically-mapped quaternions are checked to see if they are within an acceptable range of values. If not, they are *dynamically re-parameterized* to represent the same rotations that exhibit better derivative values [12]. More details on the logarithmic map and dynamic re-parameterization are discussed in Appendix A.

5.5 Residual Check

With an under-constrained system a solution will always be available. The artist, however, may over-constrain the optimization problem by placing too many key frame constraints. This can happen especially if the desired motion requires external joint torques, a force not addressed in this thesis. Since no solution will completely satisfy the linear constraints in Equation 5.1, the system tries to find a $\delta\mathbf{p}^*$ that minimizes the residual error $(A\delta\mathbf{p}^* - \delta\mathbf{k})^T (A\delta\mathbf{p}^* - \delta\mathbf{k})$.

Yet even the minimized error may be too great, leading to erratic system behavior. To monitor the system's response, an extra step is taken to ensure that the residual is below a certain threshold. In the implementation, the default value is 0.1. If the residual is greater than this threshold, then the control vector is left unedited. To provide feedback, the widget displayed, whether a pole or an arcball, turns to a red color (Figure 5-2). The artist has the option to turn off the residual calculation or change the threshold value.

5.6 Editing Across Configuration Changes

Initially free-falling, an object is animated from time t_0 to time t_f and clamps at time t_c , where $t_0 < t_c < t_f$. This animation can be decomposed into two clips, or segments where the object exhibits the same dynamics. The first clip simulates the object in free-flight, while the second simulates the object in a clamped configuration. Let \mathcal{F} represent the equations of motion of the first clip, and \mathcal{G} that of the second clip. Extrapolating from Equation 3.5, the simulation function is:

$$\mathcal{S}(t, \mathbf{p}) = \begin{cases} \mathbf{y}_{t_0}(\mathbf{p}) + \int_{t_0}^t \mathcal{F}(t, \mathbf{y}(t), \mathbf{p}) dt & \text{if } t \in [t_0, t_c) \\ \mathcal{C}(\mathbf{y}_{t_c}^-) + \int_{t_c}^t \mathcal{G}(t, \mathbf{y}(t), \mathcal{C}(\mathbf{y}_{t_c}^-)) dt & \text{if } t \in [t_c, t_f] \end{cases} \quad (5.8)$$

The Jacobians of analytically differentiable functions \mathcal{F} and \mathcal{G} compose sensitivity matrix $\Phi = \partial\mathcal{S}/\partial\mathbf{p}$ via the chain rule. Editing the motion of the object at a time t where $t_0 \leq t < t_c$ uses Equation 5.2 to find Φ . To edit the motion at $t \geq t_c$, the

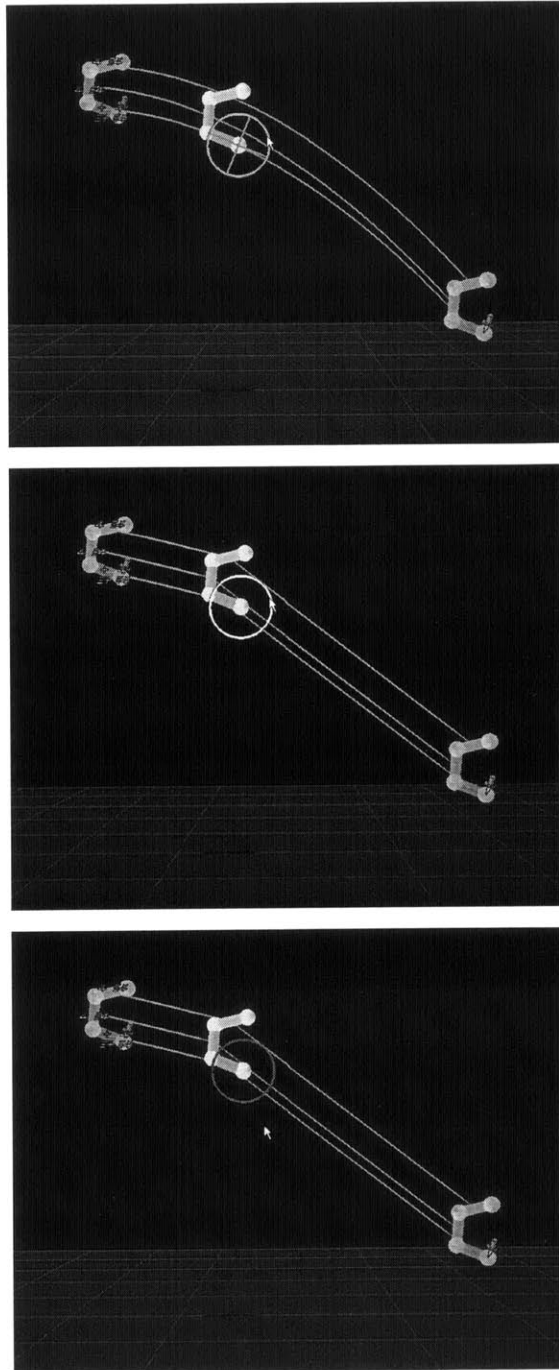


Figure 5-2: **Residual check prevents editing.** The artist constrains the position and the rotations of all the joints in both the beginning and end frames. The artist then tries to enforce an arcball constraint to make the object twirl during free-flight (top, center). The arc turns red to tell the artist the edit is not possible (bottom).

second case of Equation 5.8 is differentiated with respect to \mathbf{p} :

$$\Phi = \frac{\partial \mathbf{y}_t}{\partial \mathbf{p}} = \frac{\partial \mathcal{C}(\mathbf{y}_{t_c}^-)}{\partial \mathbf{p}} + \int_{t_c}^t \frac{\partial \mathcal{G}(t, \mathbf{y}(t), \mathcal{C}(\mathbf{y}_{t_c}^-))}{\partial \mathbf{p}} dt$$

To decompose the right hand side, consider how the simulation is calculated. The first clip is simulated until collision time t_c . The object is then re-rooted at the clamp point, leading to state vector $\mathbf{y}_{t_c}^-$. An impact force is applied to the free-falling object and the post-impact state vector $\mathcal{C}(\mathbf{y}_{t_c}^-)$, re-parameterized to the clamped configuration, becomes the initial state vector of the second clip. Finding the object's generalized state during the length of the second clip depends only on \mathcal{G} and $\mathcal{C}(\mathbf{y}_{t_c}^-)$. In general, the composition of the simulation function includes functions that are contained independently within each clip, with the collision function \mathcal{C} connecting the motion between clips. The previous equation expands using the chain rule:

$$\begin{aligned} \frac{\partial \mathbf{y}_t}{\partial \mathbf{p}} &= \frac{\partial \mathcal{C}(\mathbf{y}_{t_c}^-)}{\partial \mathbf{p}} + \int_{t_c}^t \frac{\partial \mathcal{G}(t, \mathbf{y}(t), \mathcal{C}(\mathbf{y}_{t_c}^-))}{\partial \mathbf{p}} dt \\ &= \frac{\partial \mathcal{C}(\mathbf{y}_{t_c}^-)}{\partial \mathbf{y}_{t_c}^-} \frac{\partial \mathbf{y}_{t_c}^-}{\partial \mathbf{p}} + \int_{t_c}^t \frac{\partial \mathcal{G}(t, \mathbf{y}(t), \mathcal{C}(\mathbf{y}_{t_c}^-))}{\partial \mathcal{C}(\mathbf{y}_{t_c}^-)} \frac{\partial \mathcal{C}(\mathbf{y}_{t_c}^-)}{\partial \mathbf{y}_{t_c}^-} \frac{\partial \mathbf{y}_{t_c}^-}{\partial \mathbf{p}} dt \end{aligned}$$

An arbitrary number of configuration transitions can be composed using similar methods.

Since numerically integrating Equation 5.3 derives Φ , the computation required to calculate $\partial \mathcal{S} / \partial \mathbf{p}$ remains independent of how many configuration transitions exist between the beginning of the animation and the editing time instant. Numerical integration and cumulative application of the collision function cause the initial state vector and Jacobian of each clip to include past chain rule multiplications. Thus, finding $\partial \mathcal{S} / \partial \mathbf{p}$ will always require multiplying $\partial \mathcal{S}(t, \mathbf{p}) / \partial \mathcal{C}(\mathbf{y}_{t_c}^-)$ with $\partial \mathcal{C}(\mathbf{y}_{t_c}^-) / \partial \mathbf{p}$.

When a clamping collision occurs, translational degrees of freedom are lost since a point on the object's root body is held fixed in space. Any edits made within a clip where the object is clamped will deny adjustment of the clamp point's location, since

the location of the object has no influence on the object's movement. This situation is undesirable, especially if the initial control vector guess leads to a motion where the clamp point is not at the desired location.

A glance at the format of the Jacobians substantiates this argument. Analytically, the Jacobian $\partial \mathbf{y}_t / \partial \mathbf{p}$ can be broken into four parts:

$$\frac{\partial \mathbf{y}_t}{\partial \mathbf{p}} = \begin{bmatrix} \partial \mathbf{x}_t / \partial \mathbf{p} \\ \partial \boldsymbol{\theta}_t / \partial \mathbf{p} \\ \partial \mathbf{v}_t / \partial \mathbf{p} \\ \partial \boldsymbol{\omega}_t / \partial \mathbf{p} \end{bmatrix}$$

where \mathbf{x} represents the location of the root body, \mathbf{v} the root body's linear velocity, and $\boldsymbol{\theta}$ and $\boldsymbol{\omega}$ refer to the angles and angular velocities of all rigid bodies, respectively. With a clamped model, $\partial \mathbf{y}_t / \partial \mathbf{p}$ reduces to:

$$\frac{\partial \mathbf{y}_t}{\partial \mathbf{p}} = \begin{bmatrix} \partial \boldsymbol{\theta}_t / \partial \mathbf{p} \\ \partial \boldsymbol{\omega}_t / \partial \mathbf{p} \end{bmatrix}$$

since the location of the root body does not influence its motion, and the clamp point's linear velocity is zero.

To allow the artist to specify a displacement of the clamp point, the Jacobian is extended to include $\partial \mathbf{x} / \partial \mathbf{p}$ at the time of collision from the previous clip, assuming the object was previously free-falling:

$$\frac{\partial \mathbf{y}_{\text{ext}}}{\partial \mathbf{p}} = \begin{bmatrix} \partial \mathbf{x}_{t_c} / \partial \mathbf{p} \\ \partial \boldsymbol{\theta}_t / \partial \mathbf{p} \\ \partial \boldsymbol{\omega}_t / \partial \mathbf{p} \end{bmatrix}$$

If the object is clamped in the previous clip, then $\partial \mathbf{x}_{t_c} / \partial \mathbf{p}$ is found while calculating the transfer function. However, if the object begins its animation in a clamped state, $\partial \mathbf{x}_{t_c} / \partial \mathbf{p}$ is taken directly from $\partial \mathbf{p} / \partial \mathbf{p}$, which is the identity matrix.

Since the location of the clamped object does not affect the dynamics equations,

changing the position of the object is independent of changing its motion. The state vector contains only angle-related attributes, necessitating external storage of \mathbf{x}_{t_c} and $\partial\mathbf{x}_{t_c}/\partial\mathbf{p}$.

5.7 Creating Plausible Motion

Some motions are simply impossible to accomplish with an object whose joints are passive. Consequently, the range of possible motions are limited to the artist, sometimes preventing her from being able to guide the motion to its desired trajectory (Section 5.5). A possible solution extends the description of the control vector such that it contains additional DOFs not explicitly contained by the simulated object. These DOFs may specify properties of the environment, such as surface normals or body masses. Their addition makes the motion *physically plausible*, where numerical accuracy can be sacrificed without affecting the perceived realism of the animation [3]. Even though its presence may go unnoticed by the artist, these factors can improve the system's response to the artist's interactions.

As an example, a three-link chain which clamps onto a bar at a time t_c could have additional parameters added to the control vector to denote an extra change in angular velocity at the clamp point. The state vector of the object in free-flight is:

$$\mathbf{y} = \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\theta} \\ \mathbf{v} \\ \boldsymbol{\omega} \end{bmatrix} = \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\theta} \\ \mathbf{v} \\ \boldsymbol{\omega}^R \\ \boldsymbol{\omega}^{\bar{R}} \end{bmatrix}$$

where $\boldsymbol{\omega}^R$ represents the angular velocity of the root body and $\boldsymbol{\omega}^{\bar{R}}$ represents the angular velocity of all other bodies. The format of the extended control vector consists of the free-flight state vector appended with three additional velocity parameters:

$$\mathbf{p} = [\mathbf{y}^T, (\boldsymbol{\omega}^{R+})^T]^T$$

where the three-element vector ω^{R+} represents the angular velocity added to the clamp point at the time of collision t_c . At time t_c , the root body's angular velocity is changed to equal the sum of the post-impact angular velocity and the added angular velocity:

$$\mathcal{C}(\mathbf{y}_{t_c}^-) = \begin{bmatrix} \theta_{t_c} \\ \omega_{t_c} \end{bmatrix} = \begin{bmatrix} \theta_{t_c} \\ \omega_{t_c}^R + \omega^{R+} \\ \omega_{t_c}^{\bar{R}} \end{bmatrix}$$

where θ_{t_c} and ω_{t_c} represent the post-impact angles and angular velocities of the object.

When the artist edits the motion at a time $t > t_c$, the Jacobian $\partial\mathcal{S}/\partial\mathbf{p}$ extends to:

$$\frac{\partial\mathcal{S}}{\partial\mathcal{C}(\mathbf{y}_{t_c}^-)} \frac{\partial\mathcal{C}(\mathbf{y}_{t_c}^-)}{\partial\mathbf{p}}$$

Since the extra angular velocity is added at the time of collision, only $\partial\mathcal{C}/\partial\mathbf{p}$ is altered:²

$$\frac{\partial\mathcal{C}(\mathbf{y}_{t_c}^-)}{\partial\mathbf{p}} = \left[\frac{\partial\mathcal{C}(\mathbf{y}_{t_c}^-)}{\partial\mathbf{y}} \left| \frac{\partial\mathcal{C}(\mathbf{y}_{t_c}^-)}{\partial\omega^{R+}} \right. \right] = \left[\begin{array}{c} \left[\frac{\partial\theta_{t_c}}{\partial\mathbf{y}} \right] \\ \left[\frac{\partial\omega_{t_c}^R}{\partial\mathbf{y}} \right] \\ \left[\frac{\partial\omega_{t_c}^{\bar{R}}}{\partial\mathbf{y}} \right] \end{array} \left| \begin{array}{c} \left[\mathbf{0} \right] \\ \left[\mathbf{1} \right] \\ \left[\mathbf{0} \right] \end{array} \right]$$

5.8 Re-Simulation

Once $\delta\mathbf{p}$ is found, the change is incorporated into the existing control vector as expressed in Equation 4.2:

$$\mathbf{p}_{\text{new}} = \mathbf{p}_{\text{old}} + \epsilon \cdot \delta\mathbf{p} \tag{4.2}$$

where ϵ is the step size. For a large gradient step size, the gradient descent method may diverge. Although line minimization would be the preferred method to finding an appropriate step size, its high computation requirement prevents the editing process from completing at interactive speeds. In practice, a small fixed step size has decent convergence properties while also enabling interactive update rates.

²Of course, the Jacobian is also extended with $\partial\mathbf{x}/\partial\mathbf{p}$ as discussed in Section 5.5. The extension is not reflected here since it is not pertinent to discussion.

The new control vector is used to re-simulate the entire motion. To keep the editing loop interactive, further optimizations are made by simulating the motion at times needed by the solver module, rather than sampling the motion at discrete time intervals. These times include:

- **The beginning and ending frames.** The beginning frame initializes the entire motion, and the ending frame, while unnecessary, helps complete the visualization of the motion.
- **Key frame constraints.** The Jacobian at the time of the key frame constraints help compose the constraints to the objective function.
- **Transfer frames.** The time instants at which the transfer function \mathcal{C} is calculated to transition from one clip to the next are needed for efficient calculation of the Jacobian.

Since only a select subset of time instants is simulated, the trajectory displayed on the screen will look ragged and simplified. Even so, the simplified trajectory can provide sufficient visual cues to the artist about the evolving motion.

Chapter 6

Results

This chapter shows some example sessions using LRBME and benchmark statistics measuring the rate of feedback and the overhead incurred by editing.

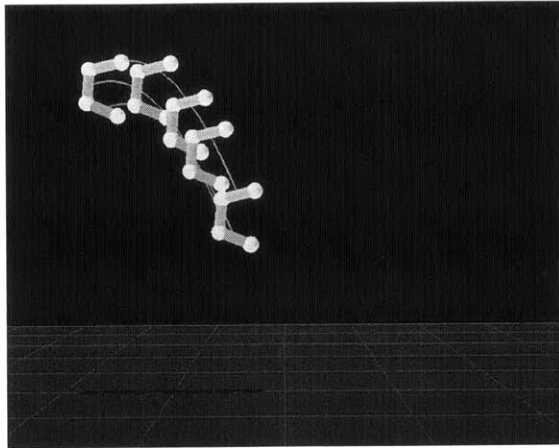
6.1 Example Sessions

Each of the sessions shown in this section took less than 5 minutes to create.

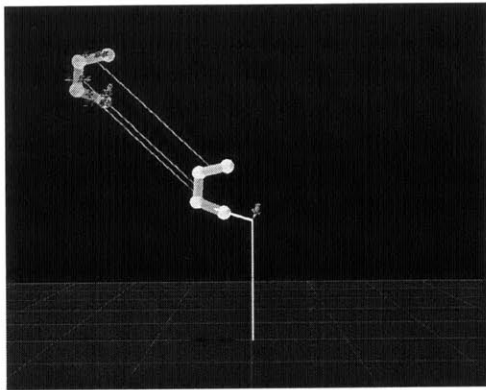
In Figure 6-1, the artist controls the motion of a three-link object with two pin joints. At first, the object is dragged across the screen to its desired end state. Once this location is enforced with a nail constraint, the artist adjusts the orientation of the body in mid-flight to create a more interesting spinning motion.

In Figure 6-2, a clamped chain with three universal joints and arbitrary initial state bounces around the clamped point. The artist uses the point constraint to create a swinging motion by pulling the chain apart. In this example, the artist uses a mass matrix in Equation 5.1 to state a preference for changing the angular velocities instead of orientations between the links.

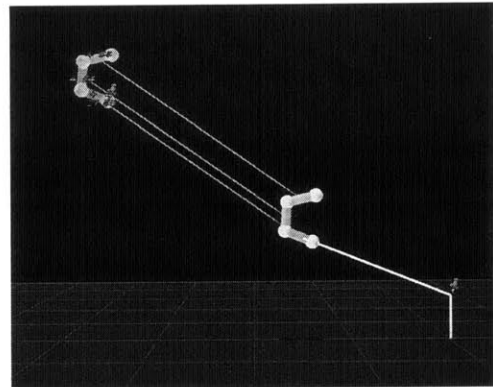
Figure 6-3 demonstrates the switch between free-flight dynamics and clamped dynamics at a prescribed simulation time. Here, a three-link object with two pin joints becomes clamped. The artist uses techniques similar to the first example to place the clamp point of the object correctly on the bar. Although editing the orientation of the clamped model, the system is able to properly alter the control vector to accommodate



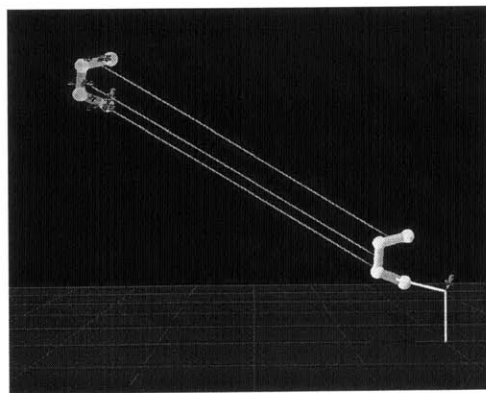
[a]



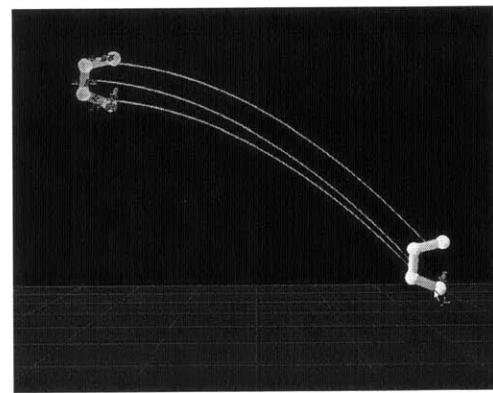
[b]



[c]

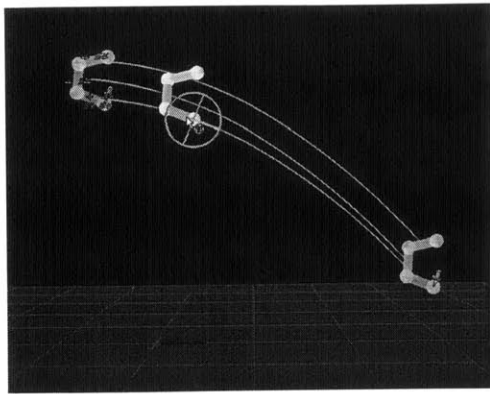


[d]

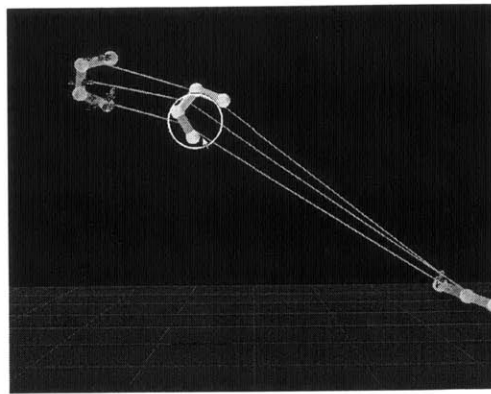


[e]

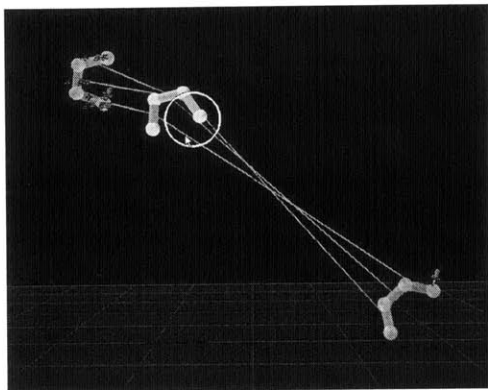
Figure 6-1: **Example 1 — Free-falling three-link object.** Each link of the object is connected by a pin joint. [a] Initial motion. [b]–[d] With the first frame held fixed by a nailed translational constraint with locked joint angles, the artist drags the object at the last frame to the desired location. [e] A nail constraint is placed on the last frame to fix translational components but omit angles.



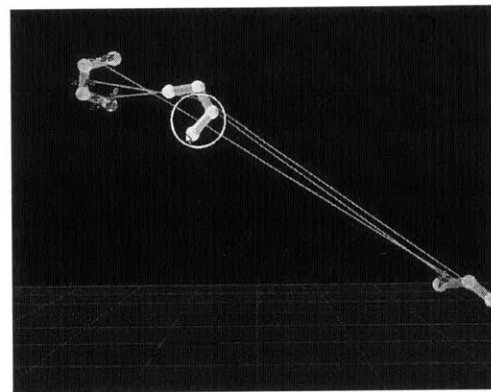
[f]



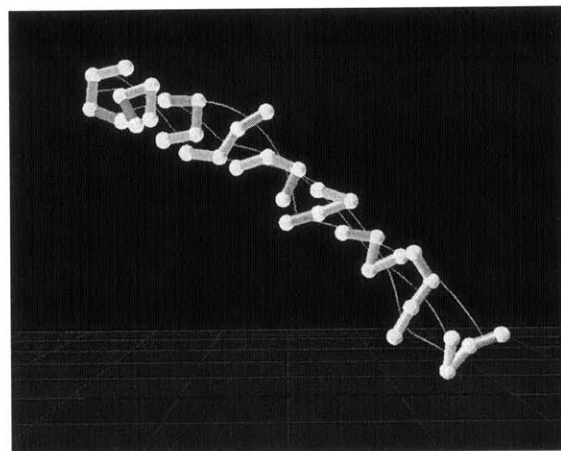
[g]



[h]

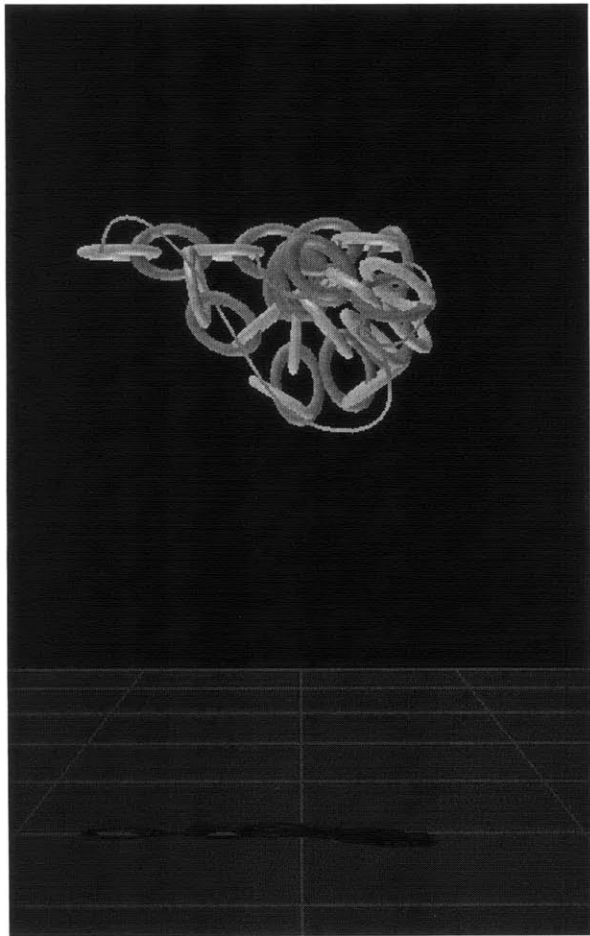


[i]

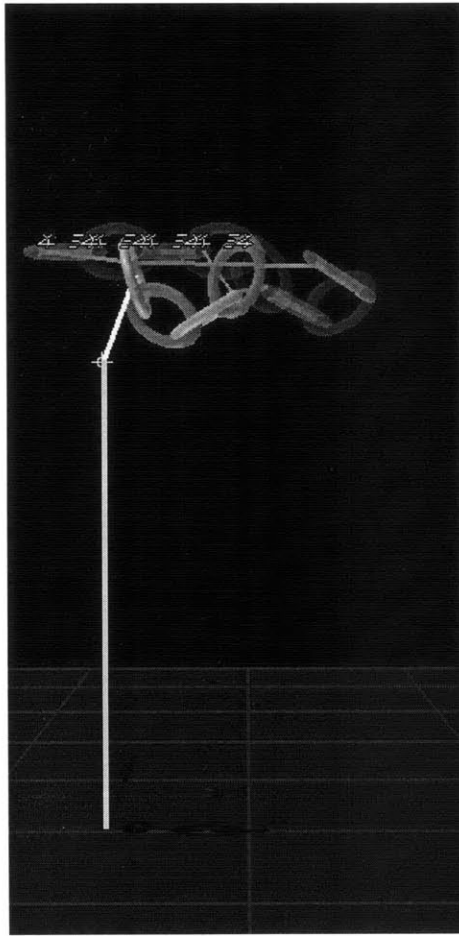


[j]

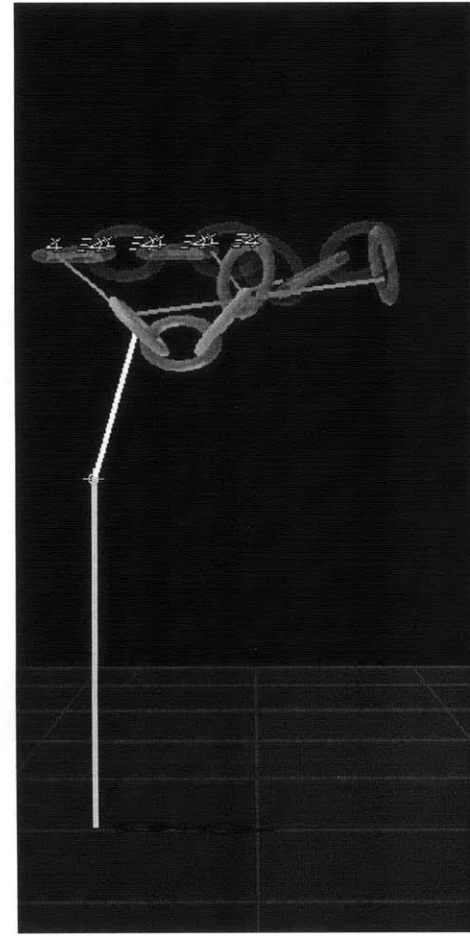
Figure 6-1 (continued): [f]–[i] An arcball constraint is used to rotate the object mid-flight. [j] Final motion.



[a]

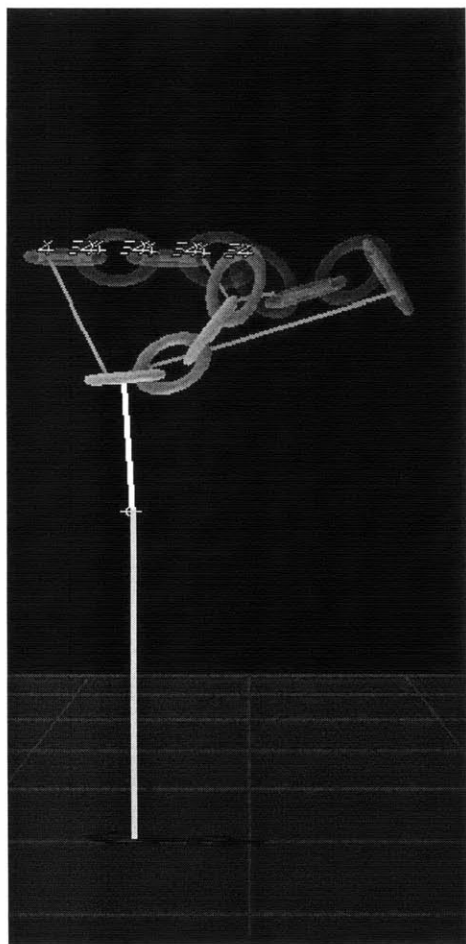


[b]

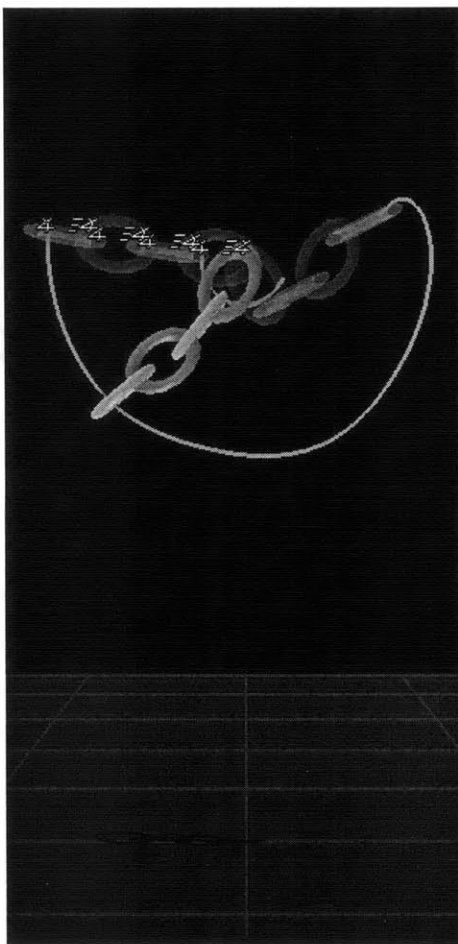


[c]

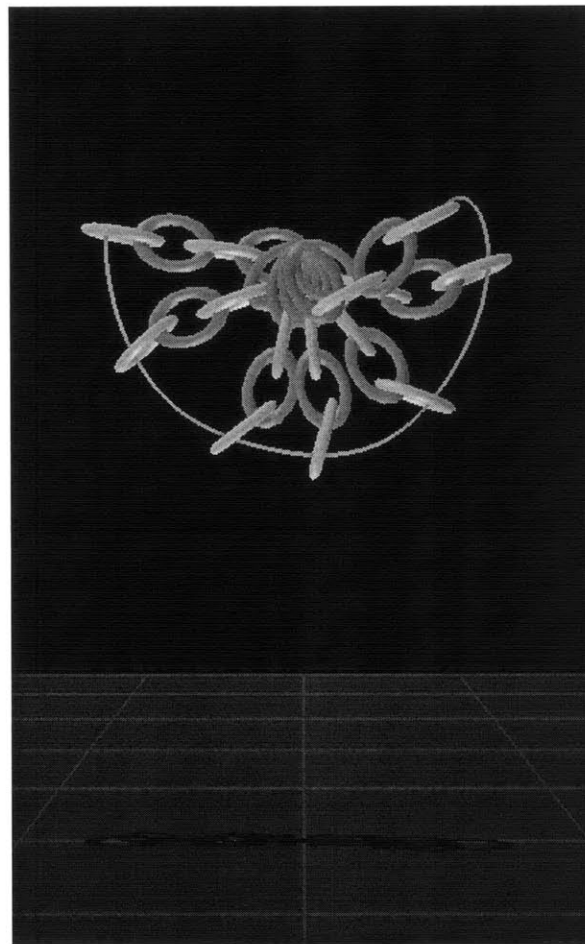
Figure 6-2: **Example 2 — Clamped 4-link chain.** The root body is clamped to a point in space by a pin joint. The chains are linked together by 2-DOF universal joints. [a] Initial motion. [b]–[e] The artist uses a point constraint and pulls the chain apart.



[d]

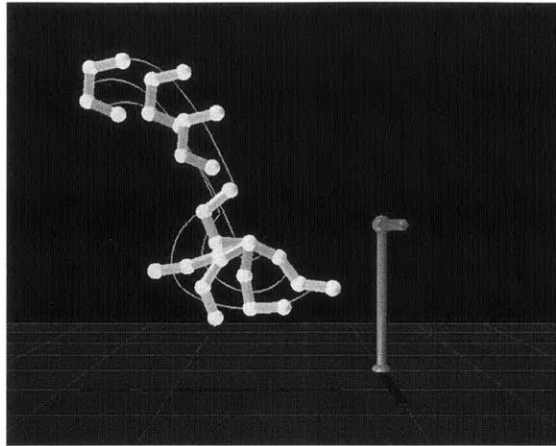


[e]

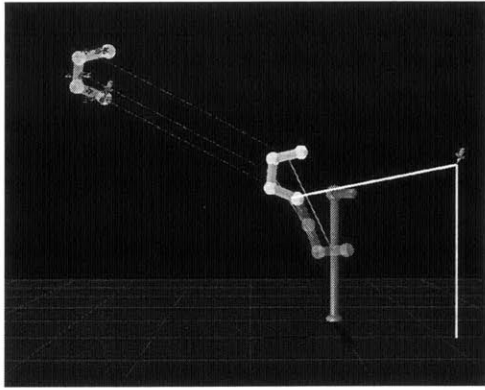


[f]

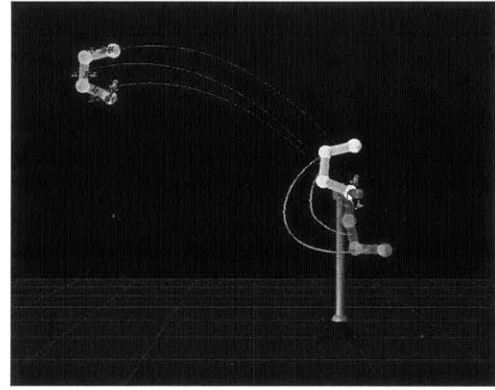
Figure 6-2 (continued): [f] Final motion.



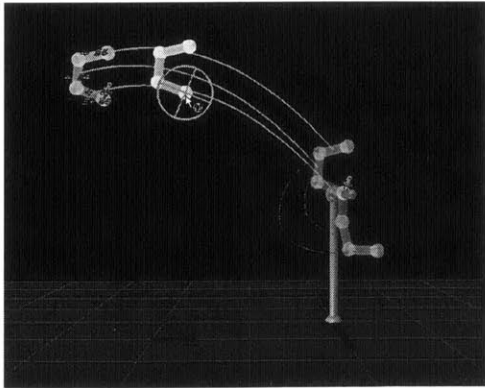
[a]



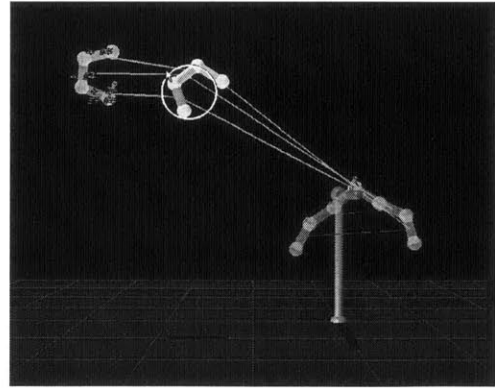
[b]



[c]

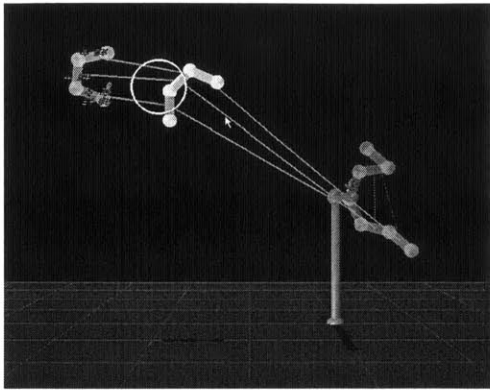


[d]

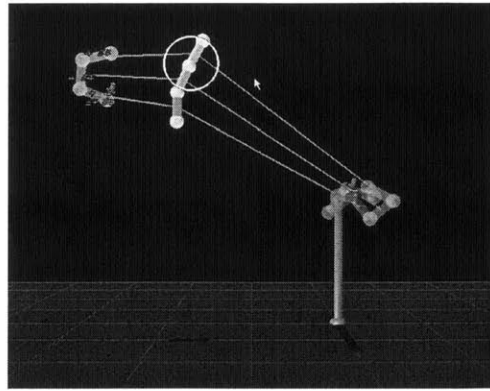


[e]

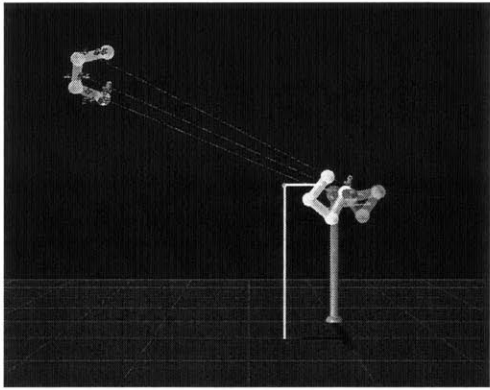
Figure 6-3: **Example 3 — Clamping Collision.** The model is similar to that in Figure 6-1. [a] Initial motion. [b] With the first frame fixed, the object is dragged to the desired location using a translational constraint. [c] The artist adds a nail constraint at the time of impact. [d]–[g] With the arcball, each body’s rotation is adjusted.



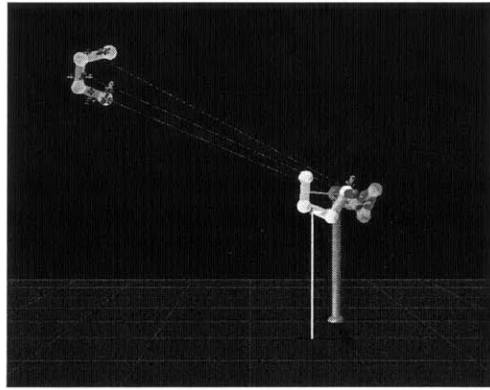
[f]



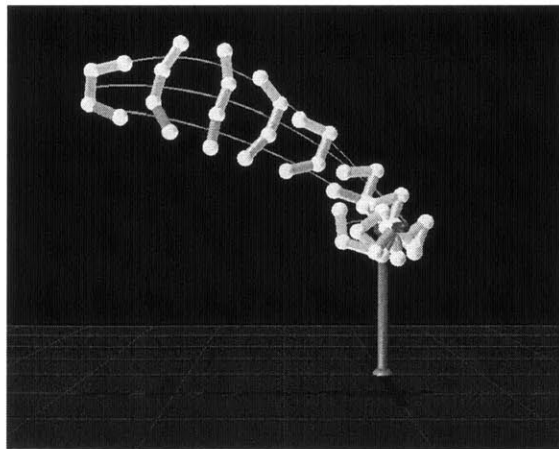
[g]



[h]



[i]



[j]

Figure 6-3 (continued): [h]–[i] The artist makes adjustments post-collision using a point constraint. [j] Final motion.

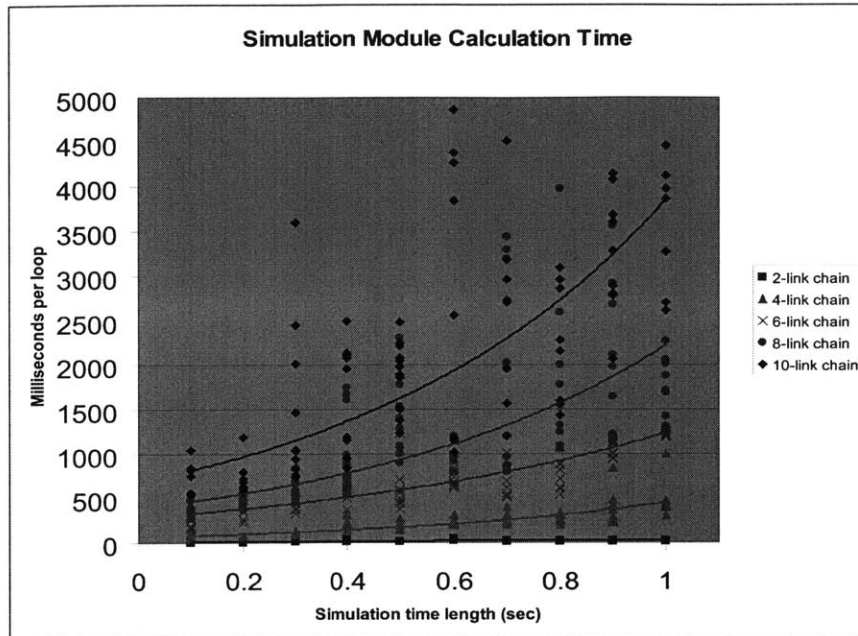


Figure 6-4: **Graph of time taken per editing loop during re-simulation.** The data was collected by recording the time it took the simulation module to re-simulate the motion at every editing loop. One interesting feature of this graph is the high variability of the data as the number of links increase. This behavior can be attributed to the varied solution space of higher-DOF motion.

the artist's edits.

6.2 Benchmarks

Benchmarking was performed on a dual Pentium Xeon 1.7GHz computer with 512 MB of RAM.

The most important concern of this technique is the overhead incurred by using the editing loop instead of simulation alone. The editing loop was divided into three individually timed parts: the differential solver which finds $\delta\mathbf{p}$, the simulator which re-calculates the motion, and the GUI update which displays the updated motion. To time each phase of the loop, linked chains of various lengths were edited with a point constraint at various time instants. Figure 6-4 shows the time taken by the simulator module and Figure 6-5 shows that by the solver module.

As the time duration increased, re-simulation time grew polynomially, as shown

in Figure 6-4. The more variable loop rate data exhibited by the longer chains exemplified the effect the complexity of the simulated motion had on response time. Consider editing the motion of a 10-link chain. A very simple dropping motion tends to make the time of the editing loop shorter than a motion where the chain jiggles around. The difference in editing time occurs simply because the numerical integrator, attempting to stay within a given tolerance of error, takes adaptive step sizes. Higher frequency motion causes the integrator to take smaller steps, and therefore more iterations to simulate the motion.

While the differential solver roughly took the same amount of time regardless of simulation time length, its time did grow polynomially with respect to the number of DOFs in the object (Figure 6-5). This behavior is expected since both the sizes of the Jacobian matrices and the number of computations needed to solve the linear system are polynomial functions of the number of DOFs in the object.

When considering the percentage of time taken by each phase, the re-simulation stage took anywhere from 60–80% of the editing loop with fewer DOFs, and 95–99% of the editing loop with more DOFs. The solver module and GUI update contributed only 5–7% to the editing loop’s time combined. So among the three phases, the re-simulation stage took the most time.

As discussed in Section 5.1, both the simulation function \mathcal{S} and the sensitivity matrix Φ are numerically integrated and stored during simulation. To determine the overhead of calculating the sensitivity matrix, two times were recorded—the time taken to complete the gesture via the editing loop, and the time taken to only simulate the motion, thereby ignoring time taken for the calculation of Φ . Each time record was divided by the number of iterations to get the average time per re-simulation. The difference between the two numbers was considered to be the overhead of editing the motion. The results are shown in Figure 6-6. With increased DOFs, the additional time required to compute the sensitivity matrices quickly became the performance bottleneck, taking as much as 1–2 orders of magnitude more time than was required to calculate the simulation function.

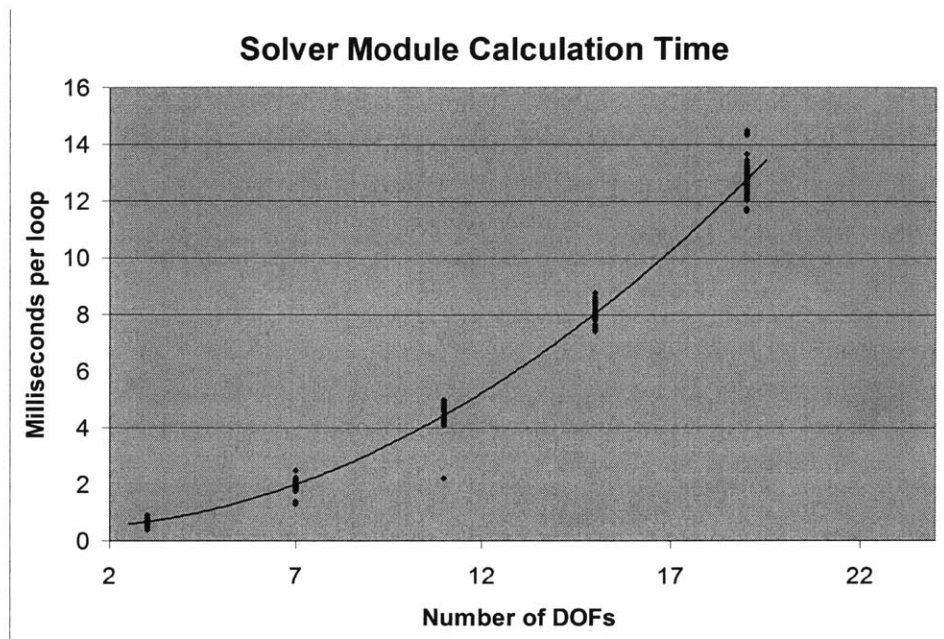


Figure 6-5: Graph of time taken per editing loop by the solver module. The solver module's time grows polynomially with the number of DOFs in the model. Despite the increase, the time the solver module takes pales in comparison to the time taken by the simulation module.

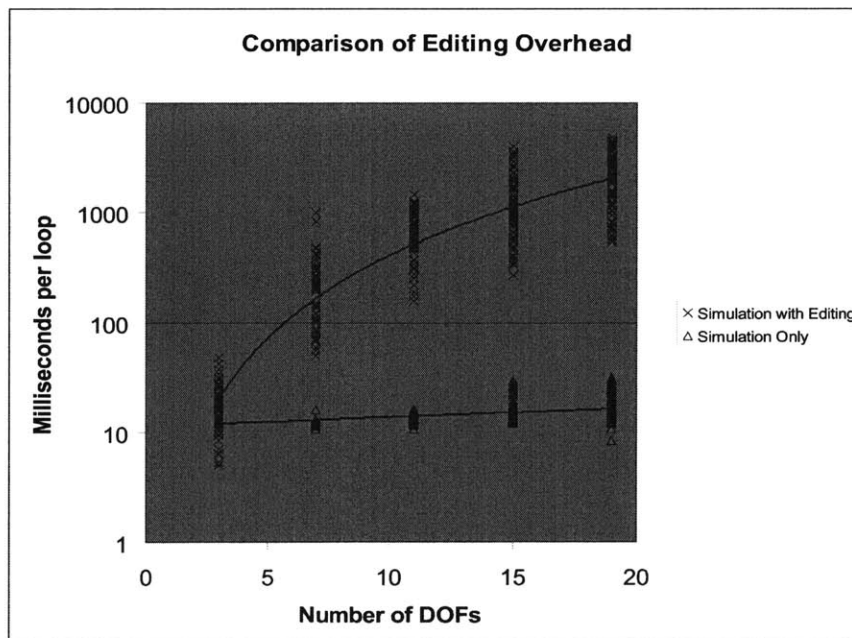


Figure 6-6: Graph depicting overhead of editing. As the number of DOFs increases, the cost of calculating the sensitivity matrix becomes the bottleneck of the editing loop's performance. Both regression curves are polynomials of degree 2.

Chapter 7

Conclusion

With the differential technique described in this thesis, an artist can rapidly prototype complex passive linked rigid-body motions. Simple, intuitive click-and-drag gestures translate into constraint equations that the system attempts to satisfy through gradient descent. Although the applicability of this technique is restricted to simulations with acyclic joint hierarchies and single-point contacts, it is a significant improvement over naïve parameter tweaking.

This click-and-drag interface can easily be extended to improve system behavior or the animation pipeline:

- With parameter estimation techniques, the artist must have an accurate sense of timing in order to make the character perform as desired. Specifically, motions requiring switches in dynamics would require the artist to know how long the character is in a particular configuration before transitioning to the next. Under the current framework, the task reduces to tweaking the timings between clips. Previous work has shown that use of error metrics can be utilized to provide additional constraints to change the time of clips as well as the object's simulation parameters [25]. Extending the technique to allow proper timing adjustments among clips could greatly improve system response.
- Automatic detection of collisions could also improve the system's functionality. A discrete search similar to that used by Harada et al. could be performed to

find minimal and local changes in current animation state [16]. For example, an animator could direct a trapeze artist to grab onto a bar automatically once he gets sufficiently near. This type of control assimilates that of an avatar, in which the artist directs a character who knows what to do within the context of its environment. For a less experienced artist, controlling a character in this manner transforms her into a director, and may enable her to express intent with broader, more abstract gestures.

- The motion sketching paradigm described in Chapter 2 could benefit from use of LRBME. A complicated motion can be divided into smaller subproblems, each of which can be manipulated using LRBME. This effectively splits the motion into the spacetime windows used by Cohen [6]. Initially, the artist sketches the desired motion by placing key frame constraints at proper times. Off-line optimization would then enforce continuity constraints as well as adjust the timing of the subproblems so that the overall motion becomes plausible.
- The control vector could also be extended to include parameters for articulated characters. Although the artist would have no mechanism for directly controlling these parameters (Section 5.7), the addition of active joints can make the solution space smoother, improving system response and reducing the need of residual enforcement (Section 5.5).

One challenge this approach faces, however, is constructing an effective parameterization of the muscle force function, especially if this is to be generalized to any type of motion. If an efficient parameterization is known for certain types of motions, such as jumping or running, an entire corpus of motion “primitives” could be compiled, whereby each primitive is designed by the artist using the differential technique. These primitives could then be concatenated to create more complex motions. This technique has been successful in active controllers [13] and motion capture [19].

- Replacing the numerical integrator with a neural network to approximate rather than simulate motion may allow an artist to interactively edit motions of com-

plex animals with many DOFs. Previous work has already shown that neural networks can produce effective and realistic locomotion, even with inputs they have never seen [14]. Since the simulation is never numerically integrated and gradient computations can be quickly derived, use of a neural network may also improve the overhead of motion editing.

In general, the technique described in this thesis could be appended to any motion creation pipeline involving physical simulation. By this point in the pipeline, the created motion will more or less satisfy the artist's intents. An application such as LRBME could then be used to refine the motion. With computation power rising in both the CPU and GPU processors, the bottleneck of numerically integrating the sensitivity matrices will become less of a hindrance, allowing the artist to create complex and realistic motion effectively for characters with many DOFs.

Appendix A

Re-parameterization of Quaternions

The non-Euclidean space that describes quaternions prevents solvers from interpolating between two quaternions linearly. Several approaches can address this issue. One solution could later enforce the unit-length constraint on the linearly interpolated quaternion. Although this technique would work, the size of the control problem increases, slowing performance. Another solution could map quaternions to vectors that belong to a space that is Euclidean. The thesis uses this technique. Although other sources discuss the mathematics of quaternions and the process of mapping quaternions more thoroughly [27, 12], a quick overview of the process and mathematics is explained here.

A.1 Basic Quaternion Notation

Quaternions lie within a four-dimensional vector space (\mathbb{R}^4). A special subset called *unit quaternions* (\mathbb{S}^3) define all possible rotations in three-dimensional space. The quaternion $\mathbf{q} = [0, 0, 0, 1]^T$ corresponds to the identity rotation, and a rotation of θ

radians about the unit axis $\hat{\mathbf{a}} = [a_{\hat{x}}, a_{\hat{y}}, a_{\hat{z}}]^T$ can be encoded into quaternion $\mathbf{q}_{\theta, \hat{\mathbf{a}}}$ as:

$$\begin{aligned} \mathbf{q}_{\theta, \hat{\mathbf{a}}} = [q_i, q_j, q_k, q_r]^T &\equiv \left[a_{\hat{x}} \sin\left(\frac{\theta}{2}\right), a_{\hat{y}} \sin\left(\frac{\theta}{2}\right), a_{\hat{z}} \sin\left(\frac{\theta}{2}\right), \cos\left(\frac{\theta}{2}\right) \right]^T \\ &\equiv \left[\sin\left(\frac{\theta}{2}\right) \hat{\mathbf{a}}, \cos\left(\frac{\theta}{2}\right) \right]^T \end{aligned} \quad (\text{A.1})$$

The length of a quaternion is defined as:

$$\|\mathbf{q}\| \equiv \sqrt{q_i^2 + q_j^2 + q_k^2 + q_r^2}$$

A unit quaternion's length is 1.

A quick example illustrates the problem with linear interpolation of quaternions. Imagine two frames of animation, where the body is rotated 0 degrees about the \hat{y} axis in the first frame, and rotated 180 degrees about the \hat{y} axis in the second frame. These original rotations map into quaternion vectors $\mathbf{q}_1 = [0, 0, 0, 1]^T$ and $\mathbf{q}_2 = [0, 1, 0, 0]^T$ respectively. An additional frame is added in between these two frames to make the animation of the body smoother. In this frame the body should be rotated 90 degrees about the \hat{y} axis. Linear interpolation between the original two quaternions leads to quaternion $\mathbf{q}' = [0, .5, 0, .5]^T$, the length of which is $(.5)^2 + (.5)^2 = .5 \neq 1$.

A few more functions and notation details are mentioned here for later use. The $\text{Vector}(\cdot)$ and $\text{Scalar}(\cdot)$ functions operate on quaternions thus:

$$\begin{aligned} \text{Vector}\left(\left[q_i, q_j, q_k, q_r\right]^T\right) &\equiv [q_i, q_j, q_k]^T \\ \text{Scalar}\left(\left[q_i, q_j, q_k, q_r\right]^T\right) &\equiv q_r \end{aligned}$$

A tilde over a vector $\mathbf{v} \in \mathbb{R}^3$ extends it to a quaternion:

$$\tilde{\mathbf{v}} \equiv [\mathbf{v}, 0]^T$$

Finally, a quaternion with an exponent of -1 denotes its *conjugate*:

$$\mathbf{q}^{-1} \equiv \left[-q_i, -q_j, -q_k, q_r \right]^T$$

A conjugate quaternion can be thought of as the inverse rotation of the original quaternion. Following a rotation expressed by \mathbf{q} by conjugate \mathbf{q}^{-1} (or vice versa) always leads to the identity rotation.

A.2 Quaternion Manipulation

Manipulating a reference frame by a series of rotations can be accomplished “traditionally” by multiplying 3×3 matrices representing those rotations. Matrix multiplication, however, is expensive and excessive since the rotation matrix includes redundant information. Since quaternions can represent rotations with fewer parameters, they significantly reduce the number of computations needed to rotate a reference frame.

Rotating a reference frame or vector by a quaternion involves use of a multiplication operator \circ , defined as:

$$\mathbf{p} \circ \mathbf{q} = \left[\hat{\mathbf{a}}_p, p_r \right] \circ \left[\hat{\mathbf{a}}_q, q_r \right] = \left[\hat{\mathbf{a}}_p \times \hat{\mathbf{a}}_q + p_r \hat{\mathbf{a}}_q + q_r \hat{\mathbf{a}}_p, p_r q_r - \hat{\mathbf{a}}_p \cdot \hat{\mathbf{a}}_q \right]$$

where $\mathbf{p}, \mathbf{q} \in \mathbb{S}^3$. Rotating a three-dimensional vector \mathbf{v} to \mathbf{v}' via the rotation described by $\mathbf{q}_{\theta, \hat{\mathbf{a}}} \in \mathbb{S}^3$ employs this operator:

$$\mathbf{v}' = \text{Vector} \left(\mathbf{q}_{\theta, \hat{\mathbf{a}}} \circ \tilde{\mathbf{v}} \circ \bar{\mathbf{q}}_{\theta, \hat{\mathbf{a}}} \right)$$

The scalar component of the resulting quaternion always equates to 0; the $\text{Vector}(\cdot)$ operator extracts the rotated vector from the quaternion.

A.3 Exponential and Logarithmic Maps

The *exponential map* correlates a vector in \mathbb{R}^3 describing a three-DOF rotation with a quaternion in \mathbb{S}^3 describing the same rotation. Conversely, the *logarithmic map* correlated unit quaternions into three-dimensional vectors. As shown in Equation A.1, all unit quaternions can be decomposed into an angle and an axis of rotation. The exponential map essentially reduces the four-element quaternion by coding the angle of rotation into the length of the axis of rotation:

$$e^{\mathbf{v}} = \begin{cases} [0, 0, 0, 1]^T & \text{for } \mathbf{v} = [0, 0, 0]^T \\ \left[\sin\left(\frac{\theta}{2}\right) \hat{\mathbf{v}}, \cos\left(\frac{\theta}{2}\right) \right]^T & \text{for } \mathbf{v} \neq [0, 0, 0]^T \end{cases}$$

where $\mathbf{v} \in \mathbb{R}^3$, $\theta = |\mathbf{v}|$, and $\hat{\mathbf{v}} = \mathbf{v}/|\mathbf{v}|$. Calculation of $\hat{\mathbf{v}}$ becomes numerically unstable as $|\mathbf{v}|$ reaches 0. Reorganization of the Vector term fixes this problem:

$$\text{Vector}(e^{\mathbf{v}}) = \sin\left(\frac{\theta}{2}\right) \frac{\mathbf{v}}{|\mathbf{v}|} = \frac{\sin(\theta/2)}{|\mathbf{v}|} \mathbf{v} = \frac{\sin(\theta/2)}{\theta} \mathbf{v} = \frac{1}{2} \text{sinc}\left(\frac{\theta}{2}\right) \mathbf{v}$$

Luckily, the sinc function is computable and continuous at and around zero. Since this function is not included in most standard math packages, the Taylor expansion of the sinc function provides a means for calculation:

$$\begin{aligned} \frac{\sin(\theta/2)}{\theta} &= \frac{1}{\theta} \left(\frac{\theta}{2} - \frac{(\theta/2)^3}{3!} + \frac{(\theta/2)^5}{5!} - \dots \right) \\ &= \frac{1}{2} - \frac{\theta^2}{48} + \frac{\theta^4}{2^5 \cdot 5!} - \dots \end{aligned}$$

Because computers have limited machine precision, the sinc function can be computed with only a few terms of the expansion, without any error. More precisely, when $|\theta| \leq \sqrt[3]{\text{machine precision}}$:

$$\frac{\sin(\theta/2)}{\theta} = \frac{1}{2} - \frac{\theta^2}{48}$$

otherwise the actual value can be acquired by computing the sine and dividing by θ .

Computation of the logarithmic map is a simple two-step process. Use of the

inverse cosine function leads to θ , which is the length of the mapped three-dimensional vector:

$$\theta = 2 \cos^{-1} q_r$$

The vector component of the quaternion is normalized, then multiplied by θ .

Due to small numerical imprecision at the asymptotes of the inverse cosine function, a “clamped” inverse cosine function $[\cos^{-1}](\cdot)$ is used instead:

$$[\cos^{-1}] \alpha = \begin{cases} \pi & \text{if } \alpha \leq -1 \\ \cos^{-1} \alpha & \text{if } -1 < \alpha < 1 \\ 0 & \text{if } \alpha \geq 1 \end{cases}$$

A.4 Calculation of Derivatives

Derivation of the Jacobian matrix $\partial \mathbf{q} / \partial \mathbf{v}$, where $\mathbf{v} \in \mathbb{R}^3$ and $\mathbf{q} = e^{\mathbf{v}}$, is as follows:

$$\frac{\partial \mathbf{q}}{\partial \mathbf{v}} = \begin{bmatrix} -\frac{v_x^2 s_\theta}{\theta^3} + \frac{v_x^2 c_\theta}{2\theta^2} + \frac{s_\theta}{\theta} & -\frac{v_x v_y s_\theta}{\theta^3} + \frac{v_x v_y c_\theta}{2\theta^2} & -\frac{v_x v_z s_\theta}{\theta^3} + \frac{v_x v_z c_\theta}{2\theta^2} \\ -\frac{v_y v_x s_\theta}{\theta^3} + \frac{v_y v_x c_\theta}{2\theta^2} & -\frac{v_y^2 s_\theta}{\theta^3} + \frac{v_y^2 c_\theta}{2\theta^2} + \frac{s_\theta}{\theta} & -\frac{v_y v_z s_\theta}{\theta^3} + \frac{v_y v_z c_\theta}{2\theta^2} \\ -\frac{v_z v_x s_\theta}{\theta^3} + \frac{v_z v_x c_\theta}{2\theta^2} & -\frac{v_z v_y s_\theta}{\theta^3} + \frac{v_z v_y c_\theta}{2\theta^2} & -\frac{v_z^2 s_\theta}{\theta^3} + \frac{v_z^2 c_\theta}{2\theta^2} + \frac{s_\theta}{\theta} \\ -\frac{v_x s_\theta}{2\theta} & -\frac{v_y s_\theta}{2\theta} & -\frac{v_z s_\theta}{2\theta} \end{bmatrix}$$

where $c_\theta = \cos(\theta/2)$ and $s_\theta = \sin(\theta/2)$. If $|\theta| \leq \sqrt[4]{\text{machine precision}}$ the sine and cosine terms are replaced with their respective Taylor expansions, and terms with

powers of 4 or greater in the numerator are discarded:

$$\frac{\partial \mathbf{q}}{\partial \mathbf{v}} = \begin{bmatrix} \frac{v_x^2}{24} \left(\frac{\theta^2}{40} - 1 \right) + \frac{1}{2} - \frac{\theta^2}{48} & \frac{v_x v_y}{24} \left(\frac{\theta^2}{40} - 1 \right) & \frac{v_x v_z}{24} \left(\frac{\theta^2}{40} - 1 \right) \\ \frac{v_y v_x}{24} \left(\frac{\theta^2}{40} - 1 \right) & \frac{v_y^2}{24} \left(\frac{\theta^2}{40} - 1 \right) + \frac{1}{2} - \frac{\theta^2}{48} & \frac{v_y v_z}{24} \left(\frac{\theta^2}{40} - 1 \right) \\ \frac{v_z v_x}{24} \left(\frac{\theta^2}{40} - 1 \right) & \frac{v_z v_y}{24} \left(\frac{\theta^2}{40} - 1 \right) & \frac{v_z^2}{24} \left(\frac{\theta^2}{40} - 1 \right) + \frac{1}{2} - \frac{\theta^2}{48} \\ -\frac{v_x}{2} \left(\frac{1}{2} - \frac{\theta^2}{48} \right) & -\frac{v_y}{2} \left(\frac{1}{2} - \frac{\theta^2}{48} \right) & -\frac{v_z}{2} \left(\frac{1}{2} - \frac{\theta^2}{48} \right) \end{bmatrix}$$

A.5 Dynamic Re-parameterization

As stated in Section 5.4.2, the exponentially-mapped quaternions are stored and maintained during the editing cycle. One more step must occur, however, before new exponentially-mapped quaternion values can be stored. The space of exponentially-mapped quaternions contains singularities when the length of the vector is a multiple of 2π . This makes sense since a rotation of 2π radians about any axis is equivalent to no rotation at all. Restricting the vectors to within a sphere of radius 2π would prevent reaching a singularity. Before storage, the length of the exponentially-mapped quaternion is checked to be no larger than a buffer value. In implementation this buffer value is 1.7π . This prevents the new exponentially-mapped quaternion from approaching the singularity region during gradient descent. If over the buffer value, the exponentially-mapped quaternion is *dynamically re-parameterized* to keep it within the allowed region. To do this, the exponentially-mapped quaternion \mathbf{v} is replaced by an equivalent rotation with better derivatives:

$$\mathbf{v} \leftarrow 1 - \frac{2\pi}{|\mathbf{v}|} \mathbf{v}$$

Appendix B

Camera Controls

The interactive camera knows the world point on which it is focused (\mathbf{c}_{cam}), how far away it is from the focus point (z_{cam}), and the rotation at which it is viewing the point (θ_{cam}) (Figure B-1). From these parameters the $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$ directions are derived, which represent the ‘x’ and ‘y’ directions of the camera view plane. Within the camera’s reference frame, the focus point is at the origin, and the camera is located at $[0, 0, z_{\text{cam}}]$, facing the $-\hat{\mathbf{z}}$ direction (Figure B-1[a]). The quaternion rotation θ_{cam} rotates the entire frame. Thus, calculation of the homogeneous modelview matrix is as follows:

$$T_{\text{cam}} = \left[\begin{array}{c|c} \mathbf{0} & -\mathbf{z}_{\text{cam}} \\ \hline \mathbf{0}^T & 1 \end{array} \right] \left[\begin{array}{c|c} \theta_{\text{cam}} & \mathbf{0} \\ \hline \mathbf{0}^T & 1 \end{array} \right] \left[\begin{array}{c|c} \mathbf{0} & -\mathbf{c}_{\text{cam}} \\ \hline \mathbf{0}^T & 1 \end{array} \right]$$

where $\mathbf{0} = [0, 0, 0]^T$, $[\mathbf{0}]$ is a 3×3 matrix of zeros, $[\theta_{\text{cam}}]$ is the rotation matrix derived from θ_{cam} , and $\mathbf{z}_{\text{cam}} = [0, 0, z_{\text{cam}}]^T$.

To rotate the camera about \mathbf{c}_{cam} , the window coordinates are mapped onto a hemisphere of canonical length 1. Coordinates outside of the sphere radius are constrained to its surface. The quaternion difference between the rotations denoted by the current and original mouse points is applied to a temporary camera rotation. Once the artist is satisfied with the new view, θ_{cam} is set to the new rotation. To constrain the rotation about the $\hat{\mathbf{y}}$ -axis, the spherical coordinates of the mouse points are first

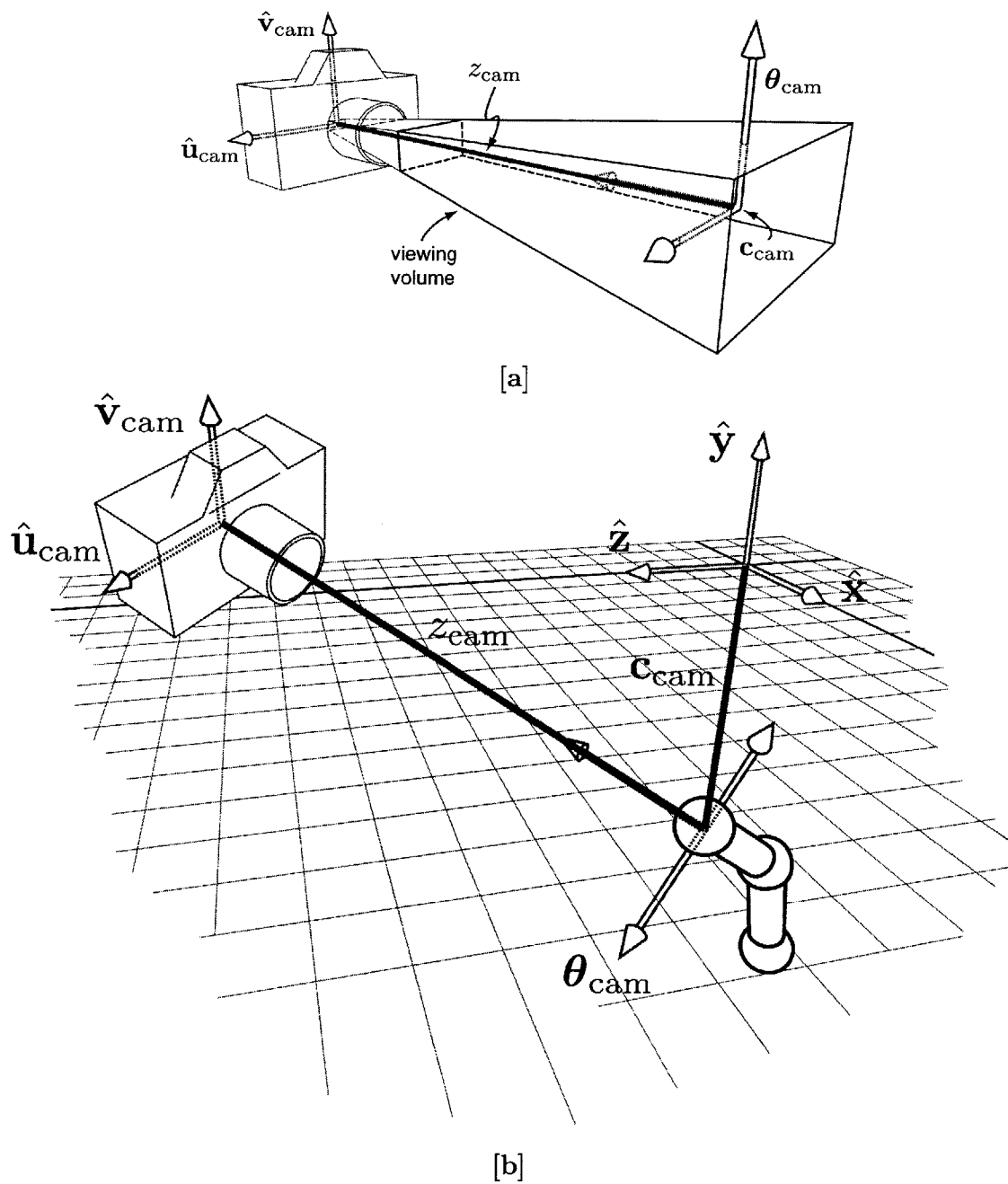


Figure B-1: **Camera Properties.** [a] Model of the camera within its own reference frame. The axes \hat{u}_{cam} and \hat{v}_{cam} represent the axes aligned with the camera view plane. The origin of the reference axes represents the point on which the camera is focused. [b] An example of the camera in the world frame. It is rotated by quaternion θ_{cam} . Vector c_{cam} represents the location of the camera focus point relative to the world origin.

projected onto the $\hat{x}\hat{z}$ plane before the quaternion difference is found. Algorithm B.1 shows pseudocode of how the camera rotation is accomplished.

Camera panning simply moves the focus point \mathbf{c}_{cam} along the $\hat{u}\hat{v}$ plane, while zooming alters the scalar z_{cam} . Algorithms B.3 and B.4 provide pseudocode that moves the camera along these directions.

Algorithm B.1 Camera Rotation

Moves camera along surface of sphere with center \mathbf{c}_{cam} and radius z_{cam}
Pseudocode for WINDOWTOSPHERE() listed in Algorithm B.2

Inputs:

\mathbf{m}_{win} — current window coordinates of mouse (in pixels)

- 1: $\mathbf{v}_{\text{old}} \leftarrow \text{WINDOWTOSPHERE}(\mathbf{m}_{\text{win}})$
 - 2: **while** mouse button is down **do**
 - 3: $\mathbf{v}_{\text{new}} \leftarrow \text{WINDOWTOSPHERE}(\mathbf{m}_{\text{win}})$
 - 4: $\Delta\boldsymbol{\theta} \leftarrow [\mathbf{v}_{\text{old}} \times \mathbf{v}_{\text{new}}, \mathbf{v}_{\text{old}} \cdot \mathbf{v}_{\text{new}}]$ {change from original rotation}
 - 5: $\boldsymbol{\theta}_{\text{temp}} \leftarrow \Delta\boldsymbol{\theta} \circ \boldsymbol{\theta}_{\text{cam}}$
 - 6: Use $\boldsymbol{\theta}_{\text{temp}}$ instead of $\boldsymbol{\theta}_{\text{cam}}$ to compute camera matrix
 - 7: Update camera view
 - 8: **end while**
 - 9: $\boldsymbol{\theta}_{\text{cam}} \leftarrow \boldsymbol{\theta}_{\text{temp}}$
-

Algorithm B.2 WINDOWTOSPHERE(\mathbf{m}_{win})

Returns unit vector \mathbf{v} , the spherical projection of window coordinate \mathbf{m}_{win}

Inputs:

$\mathbf{m}_{\text{win}} = [m_{\hat{\mathbf{u}}}, m_{\hat{\mathbf{v}}}]$ — window coordinates of mouse (in pixels)
 w, h — width and height of render window (in pixels)
 $\boldsymbol{\theta}_{\text{cam}}$ — current rotation of the camera (quaternion)

1: $\hat{\mathbf{y}}_{\theta_{\text{cam}}} = \boldsymbol{\theta}_{\text{cam}} \circ \hat{\mathbf{y}} \circ \bar{\boldsymbol{\theta}}_{\text{cam}}$ { $\hat{\mathbf{y}}$ -axis in camera space}
2: $r \leftarrow .9 \min(w, h)$ {radius of sphere in pixels}
3: $\mathbf{v} \leftarrow \frac{1}{r} [m_{\hat{\mathbf{u}}} - \frac{w}{2}, m_{\hat{\mathbf{v}}} - \frac{h}{2}, 0]$ {distance from center of sphere to mouse}
4: **if** $\|\mathbf{v}\| < 1$ **then**
5: $v_z \leftarrow \sqrt{1 - \|\mathbf{v}\|}$
6: **else**
7: Normalize \mathbf{v} {restrain to surface of sphere}
8: **end if**
9: **if** rotating about $\hat{\mathbf{y}}$ -axis **then**
10: $\mathbf{v} \leftarrow \mathbf{v} - (\mathbf{v} \cdot \hat{\mathbf{y}}_{\theta_{\text{cam}}}) \hat{\mathbf{y}}_{\theta_{\text{cam}}}$ {project onto plane denoted by $\hat{\mathbf{y}}_{\theta_{\text{cam}}}$ }
11: Normalize \mathbf{v}
12: **end if**

Algorithm B.3 Camera Panning

Moves camera focal point along plane parallel to camera view plane

Inputs:

\mathbf{m}_{win} — window coordinates of mouse (in pixels)
 w — width of render window (in pixels)

1: $\Delta \mathbf{m} \leftarrow \mathbf{m}_{\text{win}} - \mathbf{m}_{\text{old}}$ { $\Delta \mathbf{m} = [\Delta m_{\hat{\mathbf{u}}}, \Delta m_{\hat{\mathbf{v}}}]$ }
2: $\hat{\mathbf{u}} \leftarrow [\boldsymbol{\theta}_{\text{cam}}]^T \cdot [1, 0, 0]^T$ { $\hat{\mathbf{u}}$ -axis in world frame}
3: $\hat{\mathbf{v}} \leftarrow [\boldsymbol{\theta}_{\text{cam}}]^T \cdot [0, 1, 0]^T$ { $\hat{\mathbf{v}}$ -axis in world frame}
4: Normalize $\hat{\mathbf{u}}, \hat{\mathbf{v}}$
5: **while** mouse button is down **do**
6: $\mathbf{c}_{\text{temp}} \leftarrow \mathbf{c}_{\text{cam}} + (-z_{\text{cam}} \frac{\Delta m_{\hat{\mathbf{u}}}}{w}) \hat{\mathbf{u}} + (-z_{\text{cam}} \frac{\Delta m_{\hat{\mathbf{v}}}}{w}) \hat{\mathbf{v}}$
7: Use \mathbf{c}_{temp} instead of \mathbf{c}_{cam} to compute camera matrix
8: Update camera view
9: **end while**
10: $\mathbf{c}_{\text{cam}} \leftarrow \mathbf{c}_{\text{temp}}$

Algorithm B.4 Camera Zoom

Alters distance from camera to focal point

Inputs:

\mathbf{m}_{win} — window coordinates of mouse (in pixels)
 s_z — zoom speed

- 1: $\Delta\mathbf{m} \leftarrow \mathbf{m}_{\text{win}} - \mathbf{m}_{\text{old}} \{ \Delta\mathbf{m} = [\Delta m_{\hat{u}}, \Delta m_{\hat{v}}] \}$
 - 2: **while** mouse button is down **do**
 - 3: $z_{\text{temp}} \leftarrow z_{\text{cam}} + s_z \Delta m_{\hat{v}}$
 - 4: Clamp z_{temp} to near and far plane values
 - 5: Use z_{temp} instead of z_{cam} to compute camera matrix
 - 6: Update camera view
 - 7: **end while**
 - 8: $z_{\text{cam}} \leftarrow z_{\text{temp}}$
-

Bibliography

- [1] David Baraff and Andrew Witkin. Dynamic simulation of non-penetrating flexible bodies. In *Siggraph 1992, Computer Graphics Proceedings*, pages 303–308, 1992.
- [2] David Baraff and Andrew Witkin. Large steps in cloth simulation. In *Siggraph 1998, Computer Graphics Proceedings*, pages 43–54, 1998.
- [3] Ronen Barzel, John F. Hughes, and Daniel N. Wood. Plausible motion simulation for computer graphics animation. In *Computer Animation and Simulation '96, Eurographics Workshop Proceedings*, pages 184–197, Poitiers, France, September 1996.
- [4] Christian Bischof, Alan Carle, Peyvand Khademi, and Andrew Mauer. Adifor 2.0: Automatic differentiation of fortran 77 programs. *IEEE Computational Science and Engineering*, 3(Fall):18–32, 1996.
- [5] Stephen Chenney and D. A. Forsyth. Sampling plausible solutions to multi-body constraint problems. In *Siggraph 2000, Computer Graphics Proceedings*, pages 219–228, 2000.
- [6] Michael F. Cohen. Interactive spacetime control for animation. In *Siggraph 1992, Computer Graphics Proceedings*, pages 293–302, 1992.
- [7] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *Siggraph 2001, Computer Graphics Proceedings*, pages 15–22, 2001.

- [8] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *Siggraph 2001, Computer Graphics Proceedings*, pages 23–30, 2001.
- [9] Michael Gleicher. Motion editing with spacetime constraints. In *Proceedings 1997 Symposium on Interactive 3D Graphics*, pages 139–149, 1997.
- [10] Michael Gleicher and Andrew Witkin. Through-the-lens camera control. In *Siggraph 1992, Computer Graphics Proceedings*, pages 331–340, 1992.
- [11] Michael Gleicher and Andrew Witkin. Drawing with constraints. *The Visual Computer*, 11(1), 1994.
- [12] F. Sebastian Grassia. Practical parameterization of rotations using the exponential map. *Journal of Graphics Tools*, 3(3):29–48, 1998.
- [13] Radek Grzeszczuk and Demetri Terzopoulos. Automated learning of muscle-actuated locomotion through control abstraction. *Siggraph 1995, Computer Graphics Proceedings*, pages 63–70, 1995.
- [14] Radek Grzeszczuk, Demetri Terzopoulos, and Geoffrey Hinton. Neuroanimator: Fast neural network emulation and control of physics-based models. In *Siggraph 1998, Computer Graphics Proceedings*, pages 9–20, 1998.
- [15] James K. Hahn. Realistic animation of rigid bodies. In *Siggraph 1988, Computer Graphics Proceedings*, pages 299–308, 1988.
- [16] Mikako Harada, Andrew Witkin, and David Baraff. Interactive physically-based manipulation of discrete/continuous models. In *Siggraph 1995, Computer Graphics Proceedings*, pages 199–208, 1995.
- [17] Michael G. Hollars, Dan E. Rosenthal, and Michael A. Sherman. *SD/FAST User's Manual*. Version B.2. Symbolic Dynamics, Inc., Mountain View, California, September 1994.
- [18] Thomas R. Kane and David A. Levinson. *Dynamics: Theory and Applications*. McGraw-Hill, New York, New York, December 1985.

- [19] Lucas Kovar, Michael Gleicher, and Frédéric Pighin. Motion graphs. In *Siggraph 2002, Computer Graphics Proceedings*, pages 473–482, 2002.
- [20] John Lasseter. Principles of traditional animation applied to 3d computer animation. In *Siggraph 1987, Computer Graphics Proceedings*, pages 35–44, 1987.
- [21] Matthew Moore and Jane Wilhelms. Collision detection and response for computer animation. In *Siggraph 1988, Computer Graphics Proceedings*, pages 289–298, 1988.
- [22] J. Thomas Ngo and Joe Marks. Spacetime constraints revisited. In *Siggraph 1993, Computer Graphics Proceedings*, pages 343–350, 1993.
- [23] John C. Platt and Alan H. Barr. Constraint methods for flexible models. In *Siggraph 1988, Computer Graphics Proceedings*, pages 279–288, 1988.
- [24] Jovan Popović, Steven M. Seitz, and Michael Erdmann. Motion sketching for control of rigid-body simulations. *ACM Transactions on Graphics*, 22(4), October 2003. In print.
- [25] Jovan Popović, Steven M. Seitz, Michael Erdmann, Zoran Popović, and Andrew Witkin. Interactive manipulation of rigid body simulations. In *Siggraph 2000, Computer Graphics Proceedings*, pages 209–218, 2000.
- [26] Dan E. Rosenthal. An order n formulation for robotic systems. *Journal of Astronautical Sciences*, 38(4):511–529, October 1990.
- [27] Ken Shoemake. Animating rotation with quaternion curves. *Computer Graphics*, 19(3):245–254, 1985.
- [28] Ken Shoemake. Arcball: A user interface for specifying three-dimensional orientation using a mouse. In *Graphics Interface 1992, Conference Proceedings*, pages 151–156, May 1992.

- [29] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. Elastically deformable models. In *Siggraph 1987, Computer Graphics Proceedings*, pages 205–214, 1987.
- [30] Frank Thomas, Ollie Johnston, and Clie Johnston. *The Illusion of Life: Disney Animation*. Hyperion, revised edition, October 1995.
- [31] Adrien Treuille, Antoine McNamara, Zoran Popović, and Jos Stam. Keyframe control of smoke simulations. In *Siggraph 2003, Computer Graphics Proceedings*, pages 716–723, 2003.
- [32] Andrew Witkin, Michael Gleicher, and William Welch. Interactive dynamics. In *Proceedings of the 1990 symposium on Interactive 3D graphics*, pages 11–21, 1990.
- [33] Andrew Witkin and Michael Kass. Spacetime constraints. *Computer Graphics*, 22(4):159–168, 1988.
- [34] Jianmin Zhao and Norman I. Badler. Inverse kinematics positioning using nonlinear programming for highly articulated figures. *ACM Transactions on Graphics*, 13(4):313–336, October 1994.