A Differential Approach to Graphical Interaction

Michael L. Gleicher

November 18, 1994
CMU-CS-94-217

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891

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

Thesis Committee:
Andrew Witkin, Chair
Paul Heckbert
Brad Myers
Robert Sproull, Sun Microsystems

©1994 by Michael L. Gleicher

This research was supported in part by Apple Computer, an equipment grant from Silicon Graphics Inc., and a fellowship from the Schlumberger Foundation. The views and conclusions contained in this document are those of the author and should not be interpretted as representing the official policies, either expressed or implied, of these companies.

Keywords: Constraints, Direct Manipulation, Interaction Techniques, User Interface Toolkits

Abstract

Direct manipulation has become the preferred interface for controlling graphical objects. Despite its success, the ad hoc manner with which such interfaces have been designed and implemented restricts the types of interactive controls. This dissertation presents a new approach that provides a systematic method for implementing flexible, combinable interactive controls. This *differential approach* to graphical interaction uses constrained optimization to couple user controls to graphical objects in a manner that permits a variety of controls to be freely combined. The differential approach provides a new set of abstractions that enable new types of interaction techniques and new ways of modularizing applications.

The differential approach views graphical object manipulation as an equation solving problem: Given the desired values for the user specified controls, find a configuration of the graphical objects that meet these constraints. To solve these equations in a sufficiently general manner, the differential approach controls the motion of the objects over time. At any instant in time, controls specify desired rates of change that form linear constraints on the time derivatives of the parameters. An optimization objective selects a particular value when these constraints do not determine a unique solution. The differential approach solves these constrained optimization problems to compute the derivatives of the parameters. An ordinary differential equation solver uses these rates to compute object motions.

This thesis addresses the issues in using numerical techniques to provide interactive control of graphical objects. Techniques are presented to solve the constrained optimization problems efficiently and to dynamically define equations in response to system events. The thesis introduces an architecture, called Snap-Together Mathematics, that encapsulates these numerical needs. A graphics toolkit, constructed with Snap-Together Mathematics, provides the features of the differential approach yet hides the underlying machinery from the applications programmer.

The thesis demonstrates the differential approach by applying it to a variety of interaction problems, including manipulation of 2D and 3D objects, lighting, and camera control. Demonstrated interaction techniques include novel methods for some specific interaction tasks. A number of prototype applications, including 3D object construction and mechanisms sketching, demonstrate the tools and the approach.

If I lost my mind, would you help me find it?
If I lost my mind, would I have to be reminded?

— Soul Assylum
Spinning

Acknowledgements

I acknowledge everyone who needs acknowledged.

With so many other pieces of thesis to work on, I'm tempted to leave it at that. But, thanks to a large number of people, my six years in Pittsburgh have left me with a lot more than just the regional dialect.

It would be a lie for me to say I don't know where to begin. First I would like to thank my parents for their love and support throughout the years. The ski trips to Colorado the past few years were particularly useful in helping me keep my sanity as the throes of graduate student life stressed me out. My sister, grandmothers and Uncle Robert were all particularly understanding that my schedule made visits infrequent.

My six years at CMU have been a wonderful opportunity to learn and grow, not just as a computer graphics researcher, but as a person in general. Surviving the experience would not have been possible without a great set of friends who were always there to help me through the hard times, and to celebrate the good. Scott Nettles was there from our first attempts to figure out how to buy beer under Pennsylvania's laws to the celebrations as I finished. He always provided a willing ear for my complaining. Bryan Loyall and Peter Weyhrauch, my housemates for the past 5 years, helped make the house on S. Atlantic Ave. a great place to call home. Bruce Horn and Spiro Michaylov suffered through innumerable early drafts of my papers and still hung around for the fun things afterwards. It's impossible to list everyone, but David Steere, Lin Chase, James Landay, Jim Blythe, Phyllis Ruether and Greg Morrisett are the first people I think of.

Ian Davis encouraged me to get back to playing music, a much needed diversion. He, Shaun McDermott, and the rest of Painted Mice provided an outlet for me to do something besides computer science. The Thursday dinner club helped keep me well nourished, nutritionally and intellectually. And a special thanks to Lori Fabrizio for being special and for her care and patience over the past 2 years.

My advisor, Andy Witkin, gave me countless good ideas, talked me out of a lot of bad ones (and tried to talk me out of some good ones as well), and was patient with me as I learned to do math and write. He and the rest of my committee, Paul Heckbert,

Brad Myers, and Bob Sproull, really helped me turn a jumble of ideas into something resembling a thesis. Will Welch, my officemate and co-conspiritor for the past 5 years, shared countless amounts of caffeine and conversation, and in the process gave me an amazing amount of mathematical intuitions. David Baraff, Sebastian Grassia, Paul Heckbert, and Zoran Popovich all helped make the 4th floor of Doherty Hall an exciting place to do computer graphics. Phyllis Pommerantz was our "den mother." And no CMU CS thesis would be complete without thanking Sharon Burks and Catherine Copetas who really make the place run.

One of the most fun aspects of doing this thesis was to become part of the worldwide computer graphics research community. I'd like to thank everyone who shared ideas, encouragement, and skepticism. I would especially like to thank everyone at the graphics group at Apple ATG, which was my home away from home for two summers. A special thank you for the loaner computer to help with the thesis writing.

Writing this is a lot harder than I had expected. It's difficult to summarize six years of great experiences on one page. I guess I took two, and still only scratched the surface.

Contents

1 Introduction **1**
- 1.1 Implementing Graphical Manipulation . . . 2
- 1.2 The Differential Approach . . . 12
- 1.3 An Approach to Graphical Interaction . . . 14
- 1.4 Thesis Roadmap . . . 20
- 1.5 The Thesis . . . 22

2 Related Work **25**
- 2.1 Uses of Constraints in Graphical Applications . . . 25
- 2.2 Constraint Solving Technologies . . . 28
- 2.3 Graphics Toolkits . . . 35
- 2.4 Interaction Techniques and Applications . . . 36

3 Differential Techniques **39**
- 3.1 The Differential Optimization Problem . . . 39
- 3.2 Solving the Differential Optimization . . . 41
- 3.3 Solving the Differential Equation . . . 44
- 3.4 Generalized Objective Functions . . . 49
- 3.5 Soft Controls . . . 55
- 3.6 An Alternate Technique . . . 58
- 3.7 A Concrete Example . . . 60
- 3.8 Summary . . . 61

4 Efficient Solution Techniques **65**
- 4.1 The Demands of Interactive Systems . . . 65
- 4.2 Scalability of the Differential Approach . . . 67
- 4.3 Solving the Linear System . . . 70
- 4.4 Reducing Problem Size . . . 74
- 4.5 Trading Accuracy for Performance . . . 76

5 Snap-Together Mathematics **77**
5.1 Evaluating Functions . . . 79
5.2 Evaluating Derivatives . . . 80
5.3 Sparse Representations . . . 82
5.4 The Snap-Together Math Library . . . 84

6 Controllers **93**
6.1 Example Interactions . . . 94
6.2 Continuous Time . . . 97
6.3 Basic Controllers . . . 100
6.4 Switching Controllers . . . 102

7 A Graphics Toolkit **111**
7.1 The Bramble Application Model . . . 113
7.2 A Simple Example . . . 116
7.3 Bramble's World . . . 120
7.4 Connectors in Bramble . . . 123
7.5 Graphical Objects . . . 124
7.6 Hooks . . . 128
7.7 Other Application Components . . . 131
7.8 The Bramble Standard 3D Interface . . . 134

8 Interaction Techniques **137**
8.1 Attributes to Control . . . 137
8.2 Strategies for Interaction . . . 151
8.3 Sources of Constraints . . . 158
8.4 Employing Switching . . . 166

9 Example Applications **171**
9.1 A Drawing Program . . . 171
9.2 A Planar Mechanisms Sketcher . . . 182
9.3 A Box-and-Arrow Diagram Editor . . . 184
9.4 A Curve Modeller . . . 185
9.5 A Collision Simulator . . . 187
9.6 3D Construction Toys . . . 187
9.7 Scene Composition . . . 192

10 Evaluation and Future Work **195**
10.1 Contributions . . . 195
10.2 Evaluation . . . 201
10.3 Directions for Future Work . . . 211
10.4 Final Remarks . . . 215

A The Whisper Programming Language **217**
A.1 Whisper Basics 218
A.2 Some Examples 220

B Performance of the Implementations **225**
B.1 A Synthetic Benchmark 226
B.2 Application Benchmarks 229

List of Figures

1.1 3D scene with a Luxo lamp 4
1.2 Schematic representation of a simple graphical object 15
1.3 Schematic representation of objects wired together 17
1.4 Schematic representation of objects and controllers 18

3.1 Point on the plane with a radial control 42
3.2 Point moving with an Euler ODE solver 46
3.3 Euler ODE solver with various step sizes 46
3.4 Euler and Runge-Kutta ODE solvers 48
3.5 Line segment dragged by one point 50
3.6 Hard and soft controls 56
3.7 Example of an error with independent soft controls 57

4.1 Block-rectangular and block-diagonal matrices 69

5.1 Example expression graph for geometric figures 78
5.2 Simple example of derivative composition 81
5.3 Half-sparse matrix 83
5.4 Scatter/gather variable representation 88

6.1 Schematic of two line segments with an attachment constraint 94
6.2 Feedback for dragging 96
6.3 Timeline of a dragging operation 97
6.4 Discretized timeline of a dragging operation 98
6.5 Point bound to remain inside a rectangle 103
6.6 Clicking to a discrete set 105
6.7 Inequality constraint keeps a block above floor 106
6.8 Multiple blocks kept stacked by inequalities 108

7.1 Pieces of the Bramble toolkit 114
7.2 “Hello Cone” program output 117
7.3 Example of Bramble’s standard 3D interface 135

8.1 Variety of parametric curves connected with constraints 139

8.2 A crowbar . . . 140
8.3 Manipulating an inter-object shadow . . . 147
8.4 Virtual eyepoint for reflections . . . 148
8.5 Manipulating a reflection . . . 149
8.6 Differential slider . . . 152
8.7 Overlaying real and synthetic image for registration . . . 156
8.8 Registering real and synthetic images . . . 157
8.9 Fuel gauge widget . . . 162
8.10 Airplane gauges . . . 163
8.11 3D Widgets . . . 164
8.12 Generalized snapping away from the dragging action . . . 167
8.13 Preventing two rectangles from overlapping . . . 169
8.14 Simulating a mechanism with collisions . . . 170

9.1 Briar drawing program . . . 172
9.2 Briar's feedback mechanisms . . . 176
9.3 Constructing an equilateral triangle . . . 177
9.4 Briar's representation of constraints . . . 180
9.5 Mechtoy planar mechanisms sketcher . . . 183
9.6 Boxer diagram editor . . . 185
9.7 NewFF curve modeler . . . 186
9.8 Poly collision simulator . . . 187
9.9 PTinker 3D construction application . . . 188
9.10 Tinkertoys 3D construction application . . . 189
9.11 Merry-go-round constructed in the Tinkertoys simulator . . . 192

B.1 Sample run of the synthetic benchmark . . . 226
B.2 Performance of varying numbers of constraints . . . 227
B.3 Performance of varying numbers of variables . . . 228
B.4 5-bar linkage benchmark example . . . 231
B.5 Performance of simulating varying numbers of linkages . . . 231
B.6 4-bar parallel truss benchmark example . . . 232
B.7 Performance of simulating truss linkages of varying size . . . 233

and as she stepped from out her shell
and looked around for luck;
"Quack," said Jerusha,
"I seem to be a duck."

— Mildred P. Merryman
"Quack!" said Jerusha[Mer50]

Chapter 1

Introduction

Ever since computers have had graphical displays and pointing devices, graphical manipulation has been an important means of communicating between people and computers. Such interfaces couple the behavior of some graphical object to the input device, continuously tracking its changes with motion. Sketchpad [Sut63], the earliest interactive graphical application, introduced this style of interface, which has come to be known as direct manipulation.[1] Input and output devices continue to evolve from Sketchpad's vector display and light pen. Yet after 30 years of advancements in the hardware for interfaces, the basic notion of direct graphical manipulation remains the same.

As computers capable of supporting direct graphical manipulation have become more common, it has become the dominant interaction method for configuring graphical objects. However, present approaches to realizing graphical manipulation severely limit the types of interfaces which can be constructed. They restrict the types of interactive controls that can be provided to users and provide no facilities for combining these controls.

This thesis considers how the numerical and graphical performance of modern computers can be exploited to create an approach to realizing graphical manipulation that avoids the limitations of previous approaches. I will introduce a *differential approach* to graphical interaction, in which constrained optimization is used to couple the motion of graphical objects to a user's controls. To create such an approach to graphical interaction, we must consider what types of mathematical techniques to employ, what interaction techniques to build with them, and how to incorporate them into interactive applications.

[1] Although the term "direct manipulation" is generally attributed to Ben Schneiderman [Sch83], the ideas predates his work.

1.1 Implementing Graphical Manipulation

Direct manipulation has become the dominant style of graphical interaction with good reason: it provides a uniform mode of interaction that resembles interaction with real objects in the real world. The controls on a graphical object are handles that the user can grab and drag. As the user drags a handle, the object follows the motion of the pointing device with continuous motion, providing kinesthetic correspondence.

The success of graphical manipulation leads to a desire to extend its range to a wider variety of graphical objects, control types, and applications. However, present approaches to implementing graphical manipulation limit this range. The task of implementing direct manipulation requires mapping from the user's actions on the handle to changes in the program's internal representation of the object and providing feedback to the user of these changes. To date, the former has been implemented in an ad-hoc manner. Each new type of handle must be specifically hand-crafted.

Hand-crafting each handle places two significant restrictions on the types of interfaces that can be created. First, it restricts the types of handles to those for which the mapping to object parameters can be determined by the programmer. Second, it restricts how handles can be combined, as any combination must also be hand-crafted. Because there is no standardized mechanism for defining the mappings between handles and parameters, defining new types of handles can be a difficult task.

To better illustrate these problems, consider a simple example: positioning a line segment in a drawing program. Even with this simple graphical object, there are many attributes that the user might want to specify, such as the positions of the endpoints, the position of the center, the length or the orientation. Ideally, the program should permit the user to control directly whichever attribute they desired and and mix-and-match these controls as needed. That is, each attribute should have an associated handle so that the user can select controls that are most convenient to their task, and a user should be able to employ multiple, simultaneous controls to more fully specify their intents.

A simple way for a program to represent the line segment is to store its two endpoints. This representation makes it is easy to position an endpoint: simply set a pair of parameters equal to the position of the mouse. Providing other handles is more difficult. For example, to permit the user to manipulate the length of the line segment directly requires the interface implementor to work out a bit of mathematics to compute the positions of the endpoints from the length. Had a different representation been chosen, implementing this control would have been easier. For example, if the programmer had chosen to store the center, orientation and length of the line segment, the set of attributes that could easily serve as handles would be different.

With the ad-hoc implementation methods, simultaneous controls, either to support multiple input devices or to express constraints on the object changes, require explicit hand-crafting of each combination of controls. For example, maintaining the position of one endpoint of the line segment while the other is dragged can be implemented

easily if the line is represented by the positions of its endpoints. However, an interaction that maintained an endpoint's position while the center of the line segment is dragged would require some mathematical work by the interface designer if either of the representations from the previous paragraph were used.

For an object as simple as the line segment, it might be possible to predict all possible combinations of controls, or at least a sufficient set of possible combinations. However, combinatorics makes this impractical with more complicated objects. Similarly, if we consider simultaneous control of multiple objects, the increased combinatorial possibilities make explicit coding of all combinations impossible. Controls on multiple objects, such as relative positions or differences in size, further compound the problem with more potential handles, more possible combinations, and less possibility of predicting what the user will need.

Without a general mechanism for defining the mapping from a handle to the object's parameters, it is difficult to define new handles and combinations. As a result, all combinations of controls must be pre-designed by the program implementor, making experimentation with combinations of controls difficult, and dynamic combination of controls by the user impossible.

Even if combinatorics do not make it impossible to switch representations to provide alternate combinations of controls, other issues limit possible interfaces with the approach. Often, concerns such as numerical stability, freedom from singularities, and implementation convenience restrict the representations that can be used for objects. The tension between these implementation concerns and user needs leads to interfaces where the users must manipulate non-intuitive, but mathematically convenient, controls, such as B-Spline knot points, or suffer with inferior representations, such as the singularity-ridden Euler angles used by many systems for storing 3D orientations [Sho85].

In summary, the ad-hoc methods previously used to implement direct manipulation have many problems. As shown in the examples of the preceding paragraphs, they

- limit the types of interactive controls that can be provided to users;
- prevent interactive controls from being freely combined as desired;
- restrict the types of representations that programmers can use inside systems to those that user controls can be conveniently mapped to;
- fail to provide a consistent set of abstractions for defining interaction techniques;
- fail to provide a methodology for defining new controls, making it difficult to experiment with new ideas;
- prevent the realization of some potentially desirable interface styles.

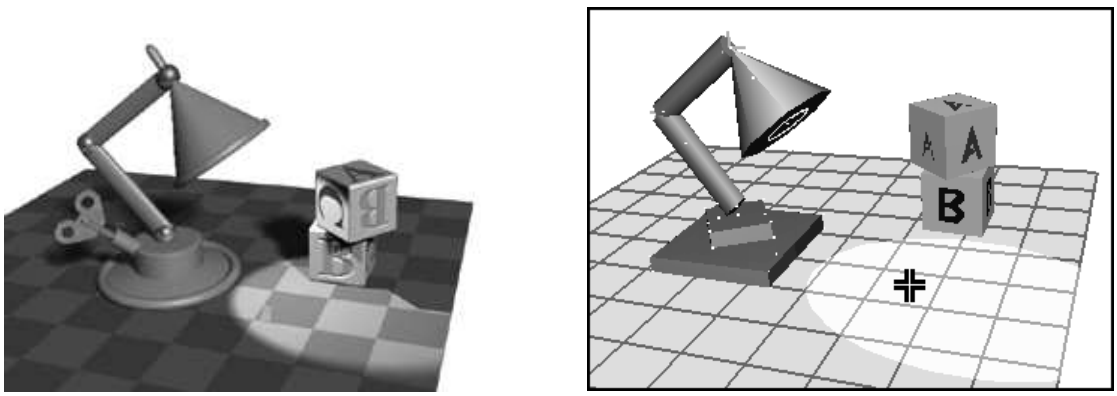

Figure 1.1: A 3D scene in which a Luxo lamp is used not only as an object in the scene, but also for illumination. To create this image, the user must configure the lamp so that the light falls in the desired location. The techniques of this thesis allow the user to control the lamp by manipulating the light's target directly, and have the lamp be adjusted accordingly. The right image shows an interactive scene composition system, described in Section 9.7, being used in this manner. (Thanks to Drew Olbrich for the ray tracing.)

This thesis provides a systematic approach to implementing direct graphical manipulation in a way that avoids these problems while keeping the essential benefits of direct manipulation.

1.1.1 A Systematic Approach to Realizing Graphical Manipulation

Our goal is to have flexible interactive controls that can be freely combined. For some interfaces, this increased power might be provided directly to users who could mix and match controls as needed for their problem. However, the extra power also helps users indirectly by giving interface designers more choices in what they can provide to users.

Some of the benefits of this flexibility are illustrated in the example of Figure 1.1. Consider an interactive application that allows a user to manipulate desktop objects, for example to create pictures of office scenes. There are many things a user may want to do with the lamp, for instance, they might want the light to shine onto a particular place, place the lamp in a particular position, or orient the lamp a certain way.

Inside the application, the configuration of the lamp might be represented as the position of the base and the angles of each of the joints, or it might be represented as the position and orientation of each part of the lamp. The former is preferable because it maintains the connections between parts of the lamp. Unfortunately, to implement a handle that permits the user to grab and drag the lightbulb, a programmer must somehow devise a mechanism to update the joint angles accordingly. The ad-hoc approaches to realizing graphical manipulation give little help in deriving such mathematics. Because the effort of deriving the handle's implementation would most likely be very

specific to the Luxo lamp, traditional[2] direct manipulation systems would most likely be forced to provide the user with only direct control over the joint angles. While this is sufficient to configure the lamp, it is not necessarily convenient for tasks like positioning the light bulb or aiming the light.

This thesis presents a systematic approach for implementing direct graphical manipulation. A general-purpose mechanism maps between the handles provided to the user and the parameters of the graphical objects. With such an approach, a user of the Luxo lamp example could not only manipulate the joint angles, but could also grab any part of the lamp directly, The programmer did not have to explicitly code the mathematics to map the manipulations into parameter changes. In fact, the flexibility in controls permits definition of other less obvious handles that permit the user to have direct control over attributes of interest. For example, a user interested in shining the light onto a particular location could simple grab the target of the lamp (the center of the spot) and drag it to the desired location. The controls can be freely combined. For example, a user could position the light's target and simultaneously specify the lamp's position on the table.

A systematic approach to realizing direct manipulation can be based on a general purpose mechanism for mapping user controls to object parameters. Creation of such a mechanism requires us to view graphical manipulation as a constrained optimization problem. To solve this problem in a practical manner, we must treat it *differentially,* that is to control how objects change rather than their final targets. This thesis introduces a *differential approach* to graphical interaction that begins by taking the view of manipulation as a mathematical problem. To realize the approach, the thesis will provide mathematical techniques to solve the problem, implementation techniques to address pragmatic issues, and a system architecture to use the approach to build applications. Example interaction techniques will be provided to show the promise of the approach, and applications will be demonstrated to show its viability.

1.1.2 Classes of Users and Tools

There are different classes of people involved with an interactive graphical application. As in Myers' survey [Mye93], we will need to distinguish these into distinct categories. Myers' categories are users, interface designers, application programmers, and tool creators. For the purposes of this thesis, we will lump interface designers and application programmers together as their tasks are similar: to build the application that the user will employ in their graphical task with the tools provided by the tool creators. The application builders will be the users of application development tools, but unless we

[2]The Luxo lamp is an example of an important special case: an articulated figure. Recently, several commercial animation systems, such as SoftImage [Sof93] and Wavefront [Wav94], have included inverse kinematic techniques to manipulate such objects by positioning end-effector points. These methods, and their limitations, will be reviewed in Section 2.2.4.

explicitly refer to the "user of the toolkit," the term "user" will refer to the "end user" of the graphical application.

The work of this thesis affects all three groups. While our approach can be employed to provide conventional interfaces, it may also be used to provide new types of interfaces for users. It gives the application programmer a new set of abstractions with which to build interactive systems. Finally, for the toolkit builder, there is a new class of services that must be provided, but these services can help enhance the modularity of the tools by: providing a standard interconnection mechanism between objects; allowing the internal representation of the objects to be hidden from applications programmers; allowing tools to be provided to the application programmer that allow pieces to be assembled by combination and composition to form interaction techniques; and allowing the encapsulation of numerical constraint computations.

One might consider applications where the user is exposed to the mathematics behind their graphical application. For example, the CONDOR system [Kas92] allows the user to construct mathematical expressions that define the graphical objects. Although such an application can be constructed using the approach of this thesis, this thesis focuses on applications where the user is insulated from the mathematics, instead directly manipulating graphical objects. In fact, a goal of this thesis is to hide as much of the mathematics as possible inside the applications development tools so that only the tool creators need see it.

1.1.3 Graphical Manipulation as Equation Solving

To introduce the differential approach of this thesis, graphical manipulation must be viewed as a constrained optimization problem. Graphical manipulation deals with how a user configures a set of graphical objects to achieve some desired goals. For the lamp example, the set of graphical objects consists of the Luxo lamp, the table top, and the other objects on the table such as the blocks. I will often use the term *model* to refer to the set of objects.

In the class of graphical manipulation tasks considered in this thesis, users manipulate objects whose configurations can be stored as a concise set of real-valued parameters, called the object's *state vector.* For a given object, there are potentially many sets of parameters which might equivalently serve as a representation, as demonstrated by the line segment example. A *parameterization* is a particular representation of the state of an object.

Objects usually have many attributes that may be of interest to an observer. Since, by definition, the state vector fully describes the configuration of the object, the attributes must be determined as functions of these parameters. For this thesis, we restrict ourselves to the broad class of object attributes which can be computed by closed-form, differentiable expressions over the state variables. This class includes many of the types of models used in interactive computer graphics such as most parametric and

implicit curve and surface representations, transformation hierarchies, virtual cameras, and many simple shading models. We will not consider things such as combinatorial or discrete attributes, such as the number of sides of a polygon, or attributes computed by recursive or iterated functions such as fractals.

A *control* is an attribute of an object that can be specified or directly manipulated. For example, if a system allowed the user to drag the position of the lamp's lightbulb or the target location of the light, these attributes of the lamp would be serving as controls. A *constraint* is a control for which a fixed value is given, preventing the value of the attribute from changing. Such controls constrain the behavior of objects by restricting their motion so that the constraint is not violated. For the purposes of this thesis, the terms constraint and control are nearly interchangeable: a constraint is a control with its value fixed, a control is a constraint whose value is being specified dynamically by the user, e.g. a value constrained to follow the mouse.

A single control generally does not uniquely determine a configuration of the object. For example, if one endpoint of a line segment is specified, there is still a continuum of possible configurations for the segment. To combat such *under-constrained* situations, it is often desirable to use multiple controls simultaneously. In the cases where there is only a single input device, dragging manipulation might be combined with constraints (e.g. controls that are restricted from changing). In a sense, even a single dragging operation can be thought of as multiple controls if we consider each axis of the pointing device independently.

It is unreasonable to require the user to employ enough controls to uniquely determine the configuration of the graphical objects. The user simply may not know or care about some attributes of some objects, or it might be too much work to specify everything. In such under-constrained cases, the system must somehow choose one of the possible configurations. Without mind reading, it is impossible to reliably select the solution that the user most desires. Systems must settle for simply trying to select a solution that is reasonable. One version of this is the "Principle of Least Astonishment" [BDFB+87] which suggests the system should try to select the option that will surprise the user the least.

For an analogy, think of a model as a large machine which has a few knobs for the user to turn and many gauges whose values the user may be interested in. Suppose there are a few gauges for which the user desires a particular value. The graphical manipulation task is to find settings of the knobs such that the gauges reach these desired values. If each gauge to be specified corresponds directly to a knob, the task is easy, because each knob can be turned and set independently. However, most gauges will depend on complicated combinations of the knobs, making it harder to find settings of the knobs that achieve desired values. In this metaphor, the knobs are the parameters of the graphical objects, the gauges are attributes of the objects that the user may be interested in, and the internals of the machine correspond to the functions that compute the attributes from the parameters. Traditional implementations of direct manipulation

require the user to control the knobs directly. The methods of this thesis permit the user to use any of the gauges as controls by automatically adjusting the knobs as needed.

1.1.4 Goals for Graphical Manipulation

Treating interactive control as the specification of values for controls as in the last section leads to a concise mathematical problem. The user would like to specify some set of controls, $\mathbf{p}$. The system needs to find some configuration of the state variables, $\mathbf{q}$, which meets this. Since the controls can be computed as a function of the state variables, we have

$$\mathbf{p} = \mathbf{f}(\mathbf{q}). \tag{1.1}$$

Solving the manipulation problem is, at one level, as straightforward as solving this equation for $\mathbf{q}$. However, there are many difficult goals which we might want our solution technique to meet:

1. flexibility in the types of controls, and therefore the functions which compute them;
2. freedom to combine controls arbitrarily, "mixing-and-matching" them dynamically;
3. keeping the good properties of direct manipulation, e.g. continuous motion, rapid feedback, tight coupling of the input device to objects on the screen, . . . ; [Sch83]
4. choosing the "best" solution in under-constrained cases, and finding a "reasonable" answer even if there is no exact solution.

To aid the application implementor, there are several other goals:

5. freedom in picking representations independently of user concerns;
6. a standard procedure for defining new controls that minimizes the amount of difficult mathematical work in defining a new type of control;
7. a solving mechanism that is general purpose and encapsulatable so that a single common implementation can serve a number of applications and so that the application developers need not worry about the details of the solving mechanisms.

We would like the techniques developed to realize the approach to also:

8. work over a variety of domains;
9. be fast and scale well;
10. require only readily available, easy to code numerical algorithms. Reliance on sophisticated numerical codes that must be purchased from commercial vendors or developed by expert numerical analysts would be unacceptable.

1.1.5 The Problems of Other Approaches

Our goals make solving Equation 1.1 for **q** impractical for three general reasons:

- in order to have flexibility in the types of controls, non-linear equations may need to be solved. Such equations are hard to solve;
- in order to have flexibility in the number of controls that are specified, we must permit under-constrained and over-constrained cases;
- in order to provide the desired direct manipulation interface, object must move with continuous motion. Therefore, the solver must be fast enough and provide continuity in the solutions.

In order to provide direct manipulation with general controls by solving Equation 1.1, we must solve arbitrary systems of non-linear equations fast enough to allow for frequent enough updates to give the user the illusion of continuous motion.

In order to meet goal 1, the equation solver must be able to handle a wide range of functions, including non-linear equations. Without knowledge about the functions to be solved, sets of equations are difficult to solve. Not only is good information hard to find in general, but each combination of equations might also require specific knowledge. Because of this, [PFTV86, Chapter 9] argues that not only does no reliable, general, non-linear solver exist, but that one cannot exist.

Without global information about functions and combinations, solving techniques must rely on local information, effectively searching for solutions. Almost all non-linear solvers are iterative methods that take an initial guess as to the solution and repeatedly update the guess until they find a solution. Such a solver can never determine that there is not a solution: if it fails to find a solution it might simply mean that it has not searched hard enough. These solvers will be discussed further in Section 2.2.2.

As computers grow faster, it might become practical to consider using a sophisticated non-linear equation solver to provide direct manipulation. However, such an approach is unlikely to succeed for a number of reasons:

- despite their sophistication, the methods are heuristic and not completely reliable;
- because they are doing searches, it is difficult to predict how long it will take them to find a solution;
- the solvers may fail to find a solution, but only after spending a long time looking for it;
- the solvers do not degrade gracefully: it is difficult to limit the amount of time that they spend because their intermediate states may not be close to the answer;

When we examine the previous approaches to implementing graphical manipulation, we see that they all fail to meet some of these goals. Previous work will be explored in more detail in Chapter 2.

Traditional Direct Manipulation – The traditional method for implementing direct graphical manipulation has been to couple parameters directly to the pointing device. For example, with the luxo lamp, a conventional direct manipulation system would allow the user to connect a joint angle to a knob. Some mappings between the input and the values are possible, for example to convert the linear motion of a slider to the rotary motion of the joint, but there must be some direct way of computing the parameter values from the inputs.

Traditional implementations have been the mainstay of direct manipulation interfaces. Such interfaces have been very successful, largely due the fact that it meets goals 3 and 9. However, its limitations have restricted the types of interfaces that have been constructed. Traditional direct manipulation severely restricts the types of functions which can be used as controls (goal 1) and it provides no automatic way to combine controls (goal 2). Parameters must be chosen so that the controls will map onto them easily (failing goal 5). Because good representations must be developed for any new controls, and because these closed form mapping for controls must be found, developing new controls can be difficult work (violating goal 6).

Parametric Modeling Approaches – Parametric modeling is a variant of the traditional direct manipulation approach. Such schemes permit end users to create models with parameter dependencies. These parameters are directly specified. Parametric approaches permit a clever user to overcome some of the deficiencies of the traditional direct approach. For example, if the designer of the Luxo lamp knew the user would want to control the height of the lamp, but not the joint angles, they might have devised a way of representing the configuration of the lamp so that height is a parameter, and the joint angles are computed from that. Parametric approaches suffer from the same failures of direct manipulation, although it does permit a clever user to sometimes have some additional flexibility in the types of controls.

Traditional Constraint-Based Approaches – A constraint-based interface[3] treats Equation 1.1 by employing an equation solver. Typically, the user specifies values for various aspects of the model and then the system solves for some value of the state vector which meets these constraints. We call such a constraint-based approach a "specify-then-solve" style.

[3] I use the term *constraint-based* interface to mean that constraints are an abstraction provided to the end user of a system, rather than simply as an abstraction used by programmers.

Although the problem of solving the equations required to meet goals 1 and 2 is difficult, a bigger problem with a specify-then-solve approach is that it fails to meet goal 3. After the user specifies the constraints, the system solves the equations and then displays the result to the user. Objects jump to the new configuration, leaving the user to puzzle out what happened. This makes goal 4 even more difficult. It becomes critical to pick a good solution to avoid confusing the user. Picking the correct solution is also important because without the rapid feedback of direct manipulation it can be difficult to explore possible solutions.

A system designer might consider using interpolation to provide the desired continuous motion in a constraint solving system. After solving for a new configuration a system might make a smooth transition by interpolating between the old state and the new. However, jumping between configurations cannot be avoiding by simply interpolating. Unless something enforces the constraints in the intermediate states, the constraints may be broken, leading to potentially undesirable behavior.

Specialized Constraint-Based Approaches – The primary drawback of the traditional constraint-based approach is that it violates goal 3, the desire for direct manipulation. One approach to handling this is to restrict the class of constraints so that they can be solved faster. The best examples of this are the propagation constraint solvers, such as DeltaBlue [FBMB90]. In essence, these algorithms trade-off goals 1 and 2, in order to better meet goal 3. As a side effect, some of these algorithms provide techniques, such as constraint- hierarchies [BFBW92], to handle under-constrained cases (goal 4). Unfortunately, propagation solvers restrict the set of possible controls and the ways controls can be combined in ways that are unacceptable for graphical manipulation (failing goals 1 and 2). Also, for each new control, a variety of bi-directional methods must be generated, which may not be easy for many types of functions (failing goal 6).

The problem of determining configurations that achieve the desired attribute values is an important problem in robotics and computer animation. Such solving is referred to as *inverse kinematics.* The inverse kinematics literature, examined in Section 2.2.4, includes numerical methods that solve the systems of non-linear equations. A problems of particular interest to robotics, namely configuring articulated figures by positioning end-effectors, is particularly well-studied. Highly developed techniques have been developed and are commonplace enough to be surveyed in robotics textbooks, such as those by Craig [Cra86] or Paul [Pau81]. The techniques are now appearing in commercial computer animation systems, such as Softimage [Sof93] and Wavefront [Wav94]. The methods in such systems are not general: they only permit manipulation of a very specific control on a very specific class of model (failing goals 1, 5 and 8), and typically provide only a single control at a time (failing goal 2). The differential approach can be

viewed as a use of generalized inverse kinematics to create a general approach to implementing graphical manipulation.

1.2 The Differential Approach

Existing approaches fail to meet the goals for graphical manipulation, demanding the development of a new approach. An advantage that we have over the developers of previous approaches is that computer hardware has advanced to the point that the machines on which graphical applications are run have considerable computational and graphics performance. Such machines make it possible to do non-trivial numerical calculations in between each frame of continuous motion animation. This means that it is possible to perform some numerical constraint calculations and still provide a continuous-motion direct manipulation interface. This thesis presents such an approach to graphical interaction.

Our goals make solving the manipulation problem of Equation 1.1 difficult. Previous approaches have either restricted the equations, or restricted the desired direct graphical interaction. In this thesis, I will present an approach which makes a different kind of restriction: that we are interested only in direct graphical interaction and will always demand that objects move with continuous motion, not jump between very different configurations. The interfaces desired for graphical interaction have this property.

Because we are considering cases where objects move continuously, it is sufficient to control them by controlling how they change over time. By controlling how objects are changing, rather than controlling their configurations directly, a variant of Equation 1.1 may be solved. Controls specify the attributes' rates of change and the system solves for the state variables' rates of change to make this happen. I call this approach to graphical interaction based on this control by time derivatives the *differential approach.*

With the differential approach, at particular instants in time a solver must determine the time derivatives of the state vector given the time derivatives of the controls. We refer to this as *differential optimization.* Solving the differential optimization is a much more mathematically tractable problem than solving Equation 1.1 directly. This means that it is possible to provide direct graphical interaction (meeting goal 3), while handling a general class of non-linear functions (meeting goal 1), and allowing these to be combined in arbitrary ways (goal 2). Methods for solving the differential optimization problem address the issues of under-constrained and over-determined cases (goal 4).

The differential approach meets the implementation goals as well. By allowing almost arbitrary non-linear functions to map between controls and parameters, it provides flexibility in selecting representations of objects independently of how they will be manipulated (goal 5). The solving methods require little information about the control functions, in fact, all that is required can be found automatically given the control

function (goal 6). The mechanisms behind the differential approach are general purpose and can be encapsulated in a manner that not only hides the underlying mathematical techniques, but also permits a single implementation to serve as a building block for almost any type of system requiring graphical manipulation (goals 7 and 8). The techniques to realize the approach perform well enough to work on current machines (goal 9), without resorting to numerical routines beyond those in standard textbooks (goal 10).

1.2.1 Direct Manipulation in the Differential Approach

Digital computers provide the illusion of continuous motion of graphical objects by repeatedly redrawing the image. The time between these redraws must be sufficiently small in order for the illusion to be maintained. To support direct manipulation, a system must sample the position of the mouse and update the positions of the objects at a rapid rate.

The differential approach breaks the numerical constraint solving problem into two parts: computing the rates of change of the parameters at particular instants, and computing the trajectory of the parameters over time, given the rates of change at particular instants. The former problem is the differential optimization problem, and the latter is solving an ordinary differential equation (ODE). Between each redraw, the ODE must be solved to update the configurations of the graphical objects. Each of these solver steps advances the configuration by solving some number of differential optimizations, each determining the rate of change at some particular instant.

With the differential approach, the graphical objects cannot simply be moved with the mouse. Instead, each step they move towards a target. Limitations in ODE solving, discussed in Section 3.3, provide speed limits on how quickly objects can move, so they may not be able to reach their target in the time provided. If the target is the position of the mouse, this will cause the object to lag behind its target, gradually catching up as the mouse slows down. This can make manipulation feel as if the objects are connected to the input devices by springs, and will be discussed in Section 6.1.2. As computers grow faster, more computation can be done between each redraw while maintaining a rate sufficient to provide the illusion of continuous motion. This allows raising the effective speed limits of the objects, and can reduce the lag.

1.2.2 An Alternate View of Graphical Manipulation

An alternatative view of graphical manipulation is to imagine graphical objects as physical entities that are manipulated as physical objects in the real world: by pushing and pulling on them. With such a view, implementing direct manipulation becomes a problem of implementing an interactive physical simulation. The issues in creating such

simulations are explored by Witkin et al.[WGW90]. The techniques presented in that paper form the basis for this thesis.

The differential approach can be viewed as a variant of the physical simulation approach. The physics of the "world" is modified from that of the real world in order to facilitate manipulation. Most significantly, inertia is removed by replacing Newton's law of motion, $f = ma$, by its first derivative equivalent, $f = mv$. An object in motion is only in motion while it is being acted upon by a force. For manipulation, this has the advantage that objects remain where they are placed, rather than skidding around.

The mathematical methods used for implementing the differential approach presented in Chapter 3 are the same as those used for implementing physical simulations. Many of the numerical methods and implementation techniques in the thesis were originally conceived for implementing interactive simulations. Presenting the differential approach as constrained optimization, as done in this thesis, rather than presenting it as a physical simulation, is largely a matter of taste.

1.3 An Approach to Graphical Interaction

The ultimate goal of this research is to improve the quality of graphical manipulation interfaces. The central focus of this thesis makes an indirect step towards this goal, providing a new set of abstractions which provide more flexibility in the type of interaction techniques that can be created. This increased flexibility does not necessarily imply better interfaces — in fact, they give interface designers new ways to baffle and confuse users. However, there are several reasons to believe that the differential approach can lead to improved interaction techniques.

The differential approach permits building interfaces which have many desirable properties. It provides for continuous motion of the graphical objects. It permits interfaces to provide controls to the user which permit directly controlling attributes of interest. These controls need not directly connect to the parameters. It permits controls to be combined, either by the user or by interface elements.

The example interaction techniques of Chapter 8 show the promise of the approach. The examples which recreate prior techniques show that the abstractions provided by the differential approach are sufficiently rich to create usable interactions. Some of the newer techniques, such as the through-the-lens camera controls of Section 8.1.4 could not have been considered with previous approaches to building interfaces. Some of the examples, like the artificial horizon of Section 8.3.5, are not good interfaces. But with the differential approach, techniques can be explored without deriving the inverse mathematics, so it is possible to learn that they are unusable before investing a large amount of time and effort in their development.

The differential approach provides a new set of abstractions for building graphical interaction techniques. In the remainder of this section, we briefly introduce the

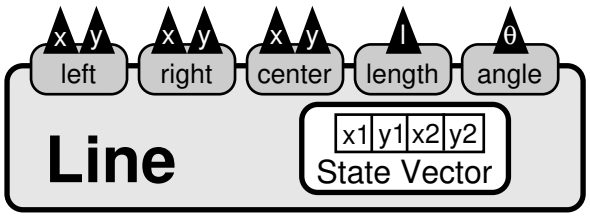

Figure 1.2: A schematic representation of a simple graphical object. The object stores a set of parameters internally in its *state vector*. However, the outside world accesses the object via its connectors, providing flexibility and parameter independence.

abstractions, along with the terminology used throughout the thesis.

1.3.1 Graphical Objects and Connectors

For the purposes of this thesis, we are concerned with what are commonly called *object-oriented* graphical editors. In such applications, the user deals with finite sets of graphical objects which must be manipulated to create the desired model or drawing.

For the most part, graphical objects are the visible entities that the user manipulates. However, we will consider structural elements, such as the groups that aggregate objects or the viewing transforms that map virtual worlds to screen coordinates as objects as well.

For a graphical object, there are two "sides" which we must consider. On one hand is what the programmer "sees," the object's internal representation. An important part of this are the parameters that determine the configuration of the object. Each object stores this set of numbers as its *state vector*.

To the user, the graphical object should appear as a graphical object. We assume that the user is interested in the graphical entity, not in the internal data structures used by the programmer. For any object, there are many attributes that may be of interest to the user, or to other objects in the program for that matter.

Ideally, we would like to think of a graphical object as a sealed box. Inside is the programmer's internal representation, including the state vector. To the outside world, all that is visible are the many attributes which other parts of the program, or the user, may want to observe. Our desire to think this way leads us to draw graphical objects schematically as Figure 1.2. The central notion is that the state is internal to the object and the object's "outputs" are its attributes. How the object computes these attributes is the concern of the object itself, not the outside world.

The state vector of an object fully specifies its configuration. Therefore, any attribute of the object must be a function of these variables. This function must be known, otherwise it would be impossible to compute the value of the attribute.

A graphical object may know how to compute many attributes. The set of attributes

of an object is not necessarily fixed — an object may have many attributes, and new attributes may be created in response to the needs of some other part of the system or the user. The schematic of Figure 1.2 may be slightly misleading in that it should not imply that the depicted outputs are a fixed, small set.

We will call the outputs of graphical objects *connectors.* As the name implies, these are the sockets into which the outside world will connect to the object. A connector is an attribute that an object provides for the outside world to access. Throughout this thesis, the notion of connector will be both a conceptual idea as well as a data structure that realizes it.

1.3.2 Compound Objects and Dependencies

Many attributes can be computed as functions of other attributes, rather than from inside the object. For example, if we wish to know the length of a line segment, this attribute could be computed as a function of the positions of endpoints. Therefore, if the line segment did not know how to produce its length as a connector, we might create a special ruler object that looks at the positions of two points and "connect" it to the endpoint outputs of the line segment.

An important notion in the ruler example is that the ruler object takes as its "inputs" the "outputs" of another object. The ruler measures the distance between two points, without concern for what these points are. This is significant for three reasons:

- It means that the objects, such as the line segment, can be extended to have new behaviors without being internally modified.
- It means that we need only one type of ruler, no matter how many different types of objects we might be measuring.
- We are not necessarily restricted to points on a single object. Instead we could measure the distance between two points on two different objects.

Objects like the ruler have inputs that plug in to the output connectors of other graphical objects. Considering such dependencies leads us to draw schematic diagrams such as Figure 1.3. The outputs of the connective objects are attributes just like the outputs of the simpler objects. The distance output of the ruler should be a first-class citizen, just as the position outputs of the line segments. Like the outputs on simpler object, the connectors on the ruler object's outputs are also functions of the state vector, except that they are potentially functions of the state vector of the entire model (which we will call the *global* state vector), rather than just the state vector of a single object. The function that determines the attribute's value can be built by composition: first computing the values of the inputs and then using these values as the inputs to a function which computes the distance.

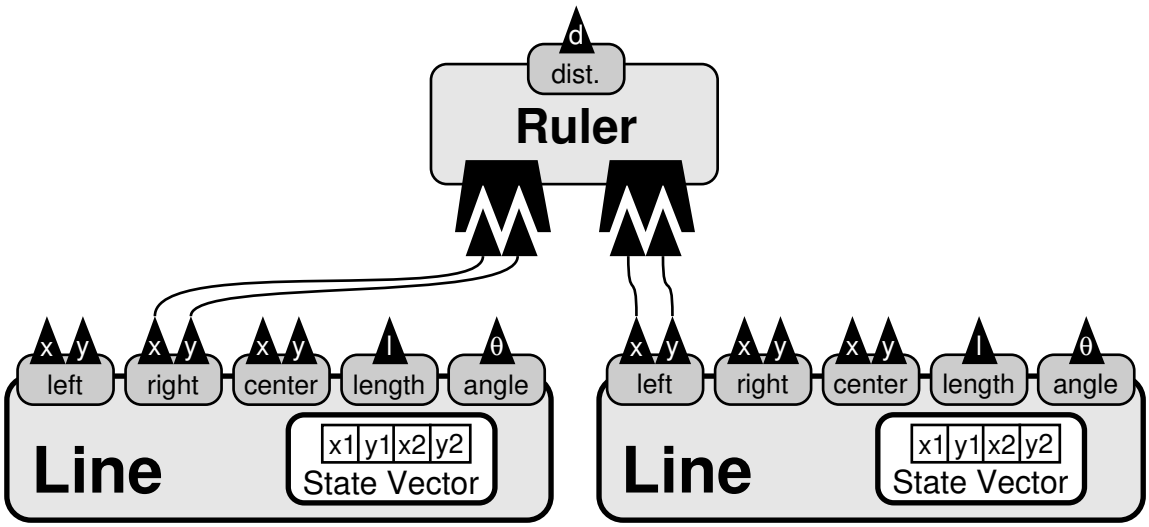

Figure 1.3: Compound objects are composed by plugging objects' connectors into sockets, like wiring together a circuit. A standardized protocol allows independence in wiring.

This picture emphasizes an important notion in the thesis: the idea of plugging objects into the "outputs" of other objects. The facility to dynamically plug and unplug such connections in response to user actions or other system events is an essential part of the differential approach, and will figure prominently in the design of the machinery to realize it.

The key element for creating the vision of snap-together objects in the differential approach is a standard protocol for the outputs so that anything can be plugged in. Since the connector outputs are primarily functions, the aggregate connection operation is function composition: building more complicated functions from simpler pieces. By supporting this operation in a dynamic environment, the machinery to realize the differential approach can permit the needed plugging and unplugging.

Compound objects, like the ruler, can come in many forms. Typically, they are used to compute aggregate properties of many different objects. For example, the distance between two points, or the relative orientation of two line segments. They may also be used to compute conversions, for example from degrees to radians. More complex attributes can also be built this way, for example, we might compute the position of a shadow as a compound operation that takes the position of a point, the position of a light source, and the position of the floor as its inputs. Flexibility in building new types of attribute outputs is a useful feature of the differential approach.

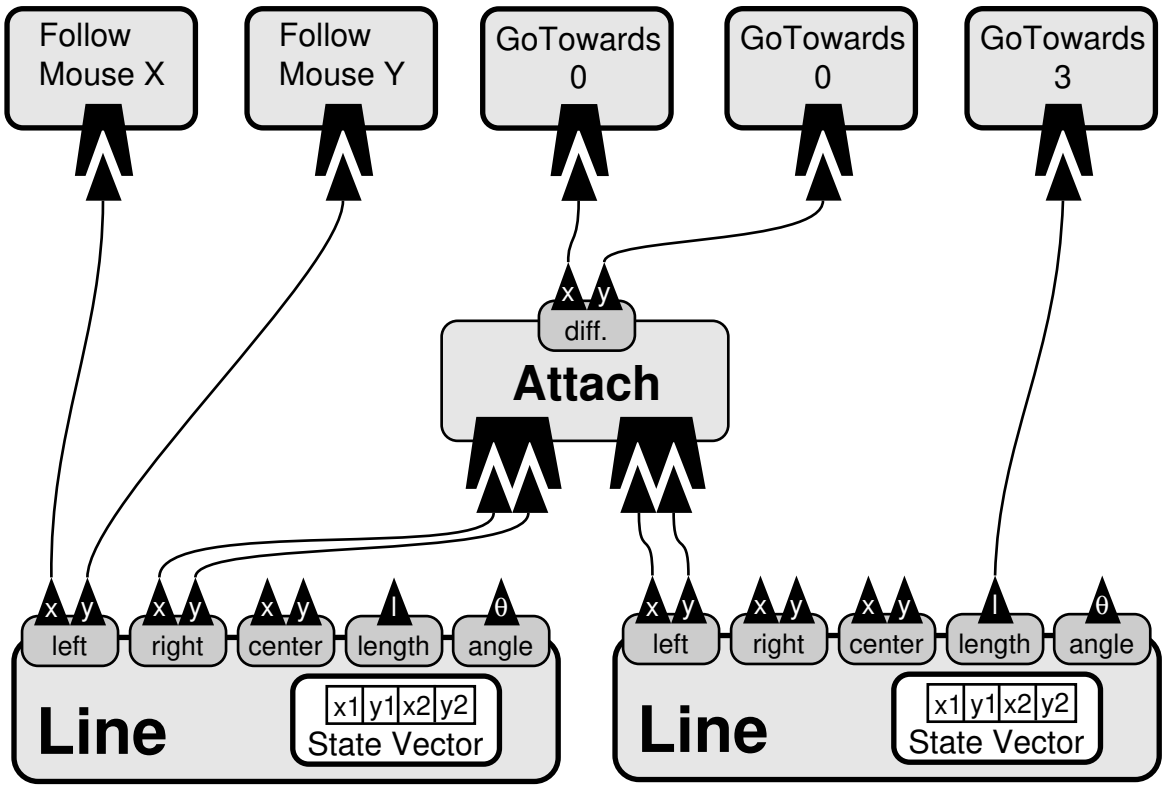

Figure 1.4: Objects are manipulated by attaching controllers to their connectors. A controller specifies how the value of a connector should be changing. Controllers can be plugged into any connector. This diagram represents a model with two line segments that are attached. One segment has its length constrained, while the other is being dragged.

1.3.3 Control of Graphical Objects

Since the attributes are the only view of an object that the programmer is given, it follows that these attributes must also serve as the handles by which the object is controlled. The vision of the differential approach is that any attribute output should be able to serve as a mechanism to control the object, and that these controls should be able to be freely applied as needed in any desired combination. Thus, any output should also be able to serve as an input.

Our notion of using an output as an input can be best discussed by introducing another kind of special object, the *controller.* A controller is a simple object that plugs into a connector and specifies what behavior the outside world desires from it. With this final abstraction, we are led to draw schematics such as Figure 1.4.

With the abstractions in place, we can now examine Figure 1.4 to see the mathematical constraint problem. We have specified the outputs of the functions that compute

the attributes being used as controls, and must determine the inputs to these functions (the value of the state vector) to achieve the desired values.

As discussed in Section 1.2, we cannot solve this constraint problem directly. Instead, we will solve it differentially. This means that rather than specifying desired values for attributes, controllers specify how they should be changing over time. A controller specifies a rate of change for the attribute it is connected to.

It is important to notice that the controllers cannot instantaneously affect the values of the connectors they control, nor the state variables of the objects. Instead, they specify how those connectors are changing, and over time, those changes will take effect. This implies that there is a continuous flow of time over which the controllers can act. At discrete instants, the set of active controllers may be altered, but values cannot be changed.

What a controller can do is quite limited: it can simply specify the desired rate of change of an attribute. The diversity of interaction techniques comes not from diversity in the types of controllers, but rather, from the way they are applied. Interesting interaction techniques result from:

- attaching controllers to interesting attributes;
- connecting controllers at interesting times;
- using controllers in interesting combinations.

Interaction techniques are defined by controlling connectors over time. For example, to drag a point, the connector that computes the point's position is connected to a controller when the mouse button is pressed to initiate the drag, and the controller is removed when the mouse button is released. Similarly, a mechanical connection between two points is created by using an object which computes the displacement between two points and creating a controller which drives the displacement to zero.

The differential approach provides a basic set of abstractions from which interfaces and interaction techniques can be built. The ability to wire together attributes and attach controllers to them provides machinery that can be applied in a wide variety of manners.

One interface style which is enabled by the differential approach is to provide the abstractions directly to the user, permitting them to mix and match controls as needed. For example, in the lamp demonstration, the user would be permitted to grab and drag many points involving the lamp, including the light's target, the bulb, and the corners of the base. Attributes which are not positional, such as joint angles or bulb brightness, might be connected to sliders. The user could configure the lamp by manipulating any of these controls, or by constraining their values. Controls are mixed-and-matched by manipulating or locking their values. This interface style is similar to a traditional constraint-based interface. Many of the issues which make constraint-based interfaces difficult to design must be addressed, such as how to present the palette of options to the user effectively.

Another way that the differential abstractions may be employed is to build interaction techniques which are more similar to the traditional direct manipulation interfaces. An example is the 3D translation widget discussed in Section 8.3.6. To the user, the translation handles appear as they do in other systems which provide them. However, this interaction technique can be concisely described by defining sets of controllers during dragging. While the differential approach is merely used to recreate an existing technique in such cases, it does have some interesting benefits. The differential approach addresses the difficult question of how to define such interesting behaviors in a way that is parameter independent, and easy to generalize to other controllers.

1.3.4 Impact on Application Architecture

Just as the differential approach frees the user from worrying about the object representations, it can also hide such parameterizations from the programmers of graphical applications, helping to foster encapsulation. Objects merely expose mathematical functional outputs for attributes that other pieces of the system may be interested in. The program manipulates the object by placing constraints and controls on these ports, and the differential solving mechanism takes care of adjusting the parameters accordingly.

The solving mechanism needs very little information about the functions that are being constrained and controlled. This means that objects need not expose much information about the functions they provide. It also simplifies the composition of functions from pieces, such as object outputs. This allows creation of a utility which permits functions to be defined dynamically, for example in response to user actions. The core functionality of the differential approach, the ability to define functions and place constraints and controls on them, can be built in a general purpose manner.

The general protocol for connecting the outputs of objects permits the creation of general purpose objects, constraints, and interaction techniques. Objects can provide mathematical ports without regard for what will "plug-in" to these ports. Constraints and interaction techniques can be defined in terms of types of outputs, without regard for the objects that are being connected to. For example, we define graphical objects that produce outputs that are the positions of points, and define constraints and interaction techniques in terms of point position outputs.

1.4 Thesis Roadmap

This thesis introduces the differential approach, presents techniques to realize it, and provides examples to illustrate its power and viability. Following this introduction, the thesis proceeds to review some relevant related work.

Chapter 3 introduces the basic set of mathematical techniques required to realize

the differential approach. The methods treat manipulation as equation solving. This problem is handled differentially to make it feasible to solve. The fundamental computation is solving a constrained optimization problem to compute how the parameters of objects are changing given the rates of change of the controls. Basic methods for solving these constrained optimization problems are developed and extended to handle under- and over- constrained cases. The chapter also considers how to use the computed rates of change to actually create the motion, a problem of solving ordinary differential equations from initial values. The chapter concludes with a simple example, worked through in detail.

In order to use numerical techniques in an interactive system, there are two central challenges that must be faced. The computations must be made to go fast enough, and the computations must be defined dynamically in response to the users actions. These issues are considered in Chapter 4 and Chapter 5 respectively. Chapter 4 considers methods to achieve the needed performance in such solving. After analyzing the computational bottlenecks of the approach, a variety of methods are presented to enhance performance. One key element is exploiting the inherent sparsity of systems of equations to be solved. Other techniques include solving smaller problems while still giving the user the illusion that the system is solving a larger problem, and trading unneeded accuracy for speed.

Chapter 5 considers the task of dynamically defining functions in a way that they can be rapidly evaluated with their derivatives. A tool called Snap-Together Mathematics that allows functions to be built dynamically from smaller pieces is presented. Snap-Together Mathematics is an important element of the differential approach because it provides the software structure for dynamically mixing and matching controls, and provides a mechanism for encapsulating the mathematics of the approach.

With the basic machinery in place, Chapter 6 considers how the tools are applied to create interaction techniques. It defines the set of abstractions provided to interface designers by the approach, and describes how the differential notion of time is different than what is commonly used in interactive-systems programming. The chapter provides some basic examples of how the abstractions are employed, and provides some extensions to the basic differential techniques to permit such things as inequality constraints.

Chapter 7 discusses how the differential approach can be encapsulated into a graphics toolkit. The Bramble toolkit was designed to aid in the development of graphical editing applications with the differential approach. Various elements of the toolkit are described, with an emphasis on how it supports the differential approach.

Chapter 8 describes interaction techniques built using the abstractions of the differential approach. It begins by discussing some basic strategies. It then provides concrete examples of techniques to address various interaction tasks. In addition to several novel interaction techniques, many previous techniques are recreated, in order to show how the Differential Approach can be applied to these problems.

Chapter 9 presents some example applications built with the approach. The applications serve to demonstrate the viability of the approach and to give some idea of its promise in constructing tools for users. Chapter 10 concludes the thesis by summarizing the contributions, evaluating the various contents, and suggesting some directions for future work.

1.5 The Thesis

It is the premise of this thesis that:

- The numerical and graphical performance of modern processors can be applied to address issues in graphical manipulation.
- A *differential approach* to graphical interaction provides a systematic implementation of direct manipulation. This approach allows a system to provide users with a broad class of interactive controls that can be freely combined, yet preserves direct manipulation, so it does not suffer from the drawbacks of other previous approaches.
- Mathematical techniques to realize the differential approach can be provided, and that these techniques can be realized such that the issues of interactive systems are addressed. In particular, methods permit the computations to be defined dynamically in response to user actions and to be performed sufficiently fast on current generation hardware.
- The techniques of the differential approach can be encapsulated, providing a set of abstractions with which to build interfaces as well as a general purpose implementation.
- The differential approach can have a positive impact on the way that interaction techniques are developed and that interactive systems are constructed, by helping separate manipulation from representation and by enabling general purpose constraints and interaction techniques.
- The differential approach can lead to interesting new interaction techniques and applications, but can also serve as a substrate for implementing existing popular interaction techniques.

1.5.1 Contributions

The contributions of this thesis are detailed in the final chapter. Briefly and generally, the contributions of this thesis are (in the order they will be presented in the thesis):

- To introduce a systematic approach to graphical interaction based on the use of numerical non-linear constraint techniques, which I call the differential approach.
- To present mathematical techniques for solving the particular constrained optimization problems encountered in using the differential approach.
- To provide techniques to implement these mathematical techniques that address the pragmatic needs of interactive systems.
- To provide a toolkit that encapsulates the differential approach, providing its features to application developers while shielding them from the details of its implementation.
- To provide new interaction techniques and examples to address problems faced by users of interactive graphical applications, and to show how these techniques fit in the context of graphical applications.
- To provide example applications demonstrating the viability of the approach.

I think the past is behind us. Real confusing if it was not, but anyway.

— Blues Traveler
But Anyway

Chapter 2

Related Work

The differential approach uses constraint techniques to realize graphical manipulation. Like other uses of constraints in computer graphics, the differential approach must address a number of challenges in applying, solving, and implementing constraints. This chapter looks at previous work on applications of constraints and constraint solving technologies. It then looks at previous work on the creation of toolkits for the construction of graphical applications, as the differential approach will be used to create such a toolkit in Chapter 7. Finally, previous work on particular 3D interaction problems used as examples Chapter 8 will be examined.

Both the basic idea of graphical manipulation, and the use of constraints to enhance it, date back to Ivan Sutherland's Sketchpad system [Sut63]. Englebart pioneered the more general use of a graphical pointing device in computer interfaces, as chronicled in [Eng86]. The style of interaction in which a pointing device controls a graphical object in a tight coupling is commonly referred to as direct manipulation, a term generally attributed to Ben Schneiderman [Sch83]. His classification of interfaces in terms of the user experience led to later attempts to better define it [WR87], and even to arguments as to why such categorizations are not helpful [WG87].

2.1 Uses of Constraints in Graphical Applications

Constraints have been used in graphical applications in many ways. Some systems provide constraint-based interfaces, that is, the users of the system are presented with constraints to use in completing their tasks. Constraint techniques have also been used to aid the programmer of graphical applications, by providing them with a tool to use in the construction of their systems. The two uses of constraints are orthogonal: it is common to write an application with a constraint-based interface using conventional tools, and to use constraint-based tools to write applications with conventional interfaces.

2.1.1 Constraint-Based Graphical Interfaces

The central idea of a constraint-based graphical interface is that the user is able to make persistent constraints: declarations that the system maintains after they are specified. The canonical example application of a constraint-based graphical interface is drawing. In a constraint-based drawing program, the user specifies relationships among parts of the drawing as persistent constraints that the system maintains during subsequent editing. For example, a user can attach an arrow to an object, and the position of the arrow is altered as the object is moved.

Sketchpad pioneered direct manipulation permitting users to directly manipulate graphical objects by dragging them with the light pen. It also introduced constraint methods, permitting users to specify relationships between parts, for example that two lines should be parallel. Sketchpad would "relax" the drawing until the constraints were satisfied, and continue to maintain the constraints during subsequent manipulations.

Since this ground-breaking application, graphical manipulation has been continually refined and has become standard. Constraints have not been as successful. Constraint-based approaches to drawing have been limited by difficulty in creating constraints, solving them, and displaying them to users. These interface issues, coupled with implementation complexity and performance problems, have prevented the widespread acceptance of constraint-based systems.

There have been examples of research systems for constraint based drawing such as Juno [Nel85], IDEAL [VW82], HILS [Whi88], CoDraw [Gro89], PictureEditor [KNK89], HotDraw [FB93], and Magrite [Gos83]. A very different use of constraints is shown in the PED picture beautifier [PW85] that automatically places constraints on a rough drawing and solves them to clean up the drawing. Another use of constraints is in the Visio [Sha93] diagramming program which permits defining object semantics with equations with a spreadsheet interface.

Recent developments such as constraint inferencing, more widely available solving technology, and the faster computers capable of solving constraints at interactive rates have renewed interest in constraint-based drawing. Systems such as Chimera [Kur93], Grace [Alp93], IntelliDraw [Ald92], Rockit [KLW92], DesignView [Com92], Converge [Sis91] and my own Briar (Section 9.1) all use constraint inferencing to couple constraints and direct manipulation.

Constraint techniques have also been applied in 3D systems. Although Sketchpad III [Joh63], the first interactive 3D application, did not have constraints, they are suggested as a requirement for future systems. The Variational Geometry systems of Lin, Gossard and Light [LGL81] renewed interest in the use of constraint techniques for designing 3D objects. Bruderlin [Bru86] and Rossignac [Ros86] presented constraint-based solid modelers. Constraint-based solid modelers that use direct manipulation input are presented by Sohrt and Bruderlin [SB91] and by Fa et al [FFD93]. David Pugh's Viking system [Pug92], uses constraints to maintain geometrical relationships

defined by sketching. Converge [Sis91] models 2D and 3D objects with constraints.

Constraint methods have been applied to surface modeling, allowing users to manipulate surfaces without seeing the underlying representations. Fowler [Fow92] and Welch, Gleicher and Witkin [WGW91] present simple constraint methods for controlling points on B-Spline surfaces. Celniker [CG91] describes methods that optimize a shape interactively, which are extended in [CW92] to a broader class of constraints. Welch and Witkin [WW92] extend this work to a wider variety of constraints that permit the user to stitch together pieces of surfaces.

The "energy constraints" work of Witkin et al.[WFB87] introduces the idea of modelling using arbitrary functions of objects as controls. Barzel [Bar92c] discusses the philosophical attractiveness of using physical constraints for modelling.

Specialized interactive graphics systems use constraints to help users manipulate complex objects. Mark Surles' SCULPT system [Sur92a, Sur92c, Sur92b] permits the interactive manipulation of molecules. The Jack system [PB88a] uses constraint methods to interactively position a human figure. In [PB91], the authors extend Jack with more complicated constraints on human figures.

The differential approach and the tools created to implement it were heavily motivated by the desire to build constraint-based applications and to study the issues involved. Seeing 30 year old films of Sketchpad inspired the desire to understand how these techniques might apply in modern systems. The energy constraints work was also particularly inspiring because it demonstrated the utility of a wide range of controls, a central theme in the differential approach. Of the systems discussed, Briar, Converge and Chimera best typify the applications motivating the differential approach.

2.1.2 Constraint-Based Tools for Building Graphical Applications

Tools for building graphical applications have employed constraints to ease the construction process, for example, by automatically maintaining consistency multiple representations of data or between views and data. The use of constraint methods to simplify the construction of graphical applications was pioneered by Borning's ThingLab system [Bor81]. The early successors to Thinglab for constructing interfaces using constraints are surveyed in [BD86]. Barth's GROW toolkit [Bar86] was another early use of constraints to help the programmer lay out the various elements of the interface.

The common use of constraint methods for maintaining consistency of data can trace its origins to non-constraint-based methods. The Smalltalk Model-View-Controller model [KP88] for direct manipulation implementation provided the influence for many other systems, despite its late appearance in the published literature. The model uses separate objects to handle input and output for application objects. A critical piece to implement the model is a mechanism for dependencies: objects must be notified of changes to other objects, for example to update the display when appropriate. The dependency mechanism is a simple form of constraints known as *one-way* because the

data only flows one way in the constraint. Distinctions among types of constraints will be explained in the next section.

More sophisticated algorithms for creating one-way constraints were introduced later, and lead to more general dependency mechanisms for interface toolkits. For example, Hudson's incremental attribute evaluation [Hud91] was used to create the Apogee toolkit [HH88]. Similar one-way mechanisms were used in other toolkits such as Coral [SM88], MEL [Hil91], and Garnet [MGD+90].

Despite the simplicity and limited expressibility of one-way constraints, they are an extremely useful feature in interface toolkits. They are sufficient to update views when data changes, keep dependant data consistent, and help designers lay out interfaces. Simple solvers are extremely popular because efficient and simple mechanisms for creating them have been widely available. As developments in solvers make more powerful techniques practical, newer toolkits explore their advantages. Examples include Rendezvous [HBP+93], ThingLab II [Mal91], VB2 [GBT93], and Multi-Garnet [SB92]. Rendezvous even permits creating constraints across multiple displays, maintaining consistency between multiple users of a shared application.

The existing constraint-based tools for developing graphical interfaces are inadequate for constructing the constraint-based applications and examining the interface questions I wanted to study. The lack of support for numerical constraints in existing tools provided a niche to be filled with the work of this thesis. It is important to explore whether numerical constraint methods could be encapsulated and provided in a toolkit.

2.2 Constraint Solving Technologies

The wide array of uses of constraint techniques has led to the creation of an even larger selection of constraint solving technologies. Here, we provide a brief survey. Most systems have only used a single solving technique. Other systems, including the early Sketchpad [Sut63] and ThingLab [Bor81], used hybrids where multiple solvers were used to solve different parts of problems.

2.2.1 Propagation and Symbolic Methods

The simplest constraint methods allow the specification of dependencies among elements of the data. Some mechanism is provided in order to make sure that dependent values are updated appropriately. These mechanisms can operate either by replacing accesses with function calls to recompute the data, or by having changed data notify their dependents. This latter approach is known as *local propagation* because new results first propagate to elements closely connected (i.e. local) in the dependency graph. Dependency schemes are called *one-way* because information flows only one way across the dependencies. Despite their simplicity, one-way local propagation techniques are

extremely popular because they are easy to implement efficiently and because they can provide some important needs in user interface software, Efficient methods for handling one-way constraints by minimizing the number of evaluations are discussed by Hudson [Hud91], and extensions to one-way constraints for indirectly referencing variables are provided by Vander Zanden et al.[VZMGS91, VZMGS94].

One way constraints describe dependencies on data. For example, consider the constraint C=A+B. A one-way constraint would declare that C depends on A and B, and when either A or B changes, C is updated accordingly. Multi-way constraints permit the dependency to be determined based on the data, allowing C to be computed when A and B are provided, or B to be computed when A and C are provided. Consequently, multi-way solvers are more complex than one-way solver, which has hindered their acceptance. Sophisticated multi-way local propagation solvers such as DeltaBlue [FBMB90], SkyBlue [San94], and the methods of Vander Zanden [VZ88, VZ89], are now becoming more readily available. Sannella et al [SMFBB93] argue that they are as efficient as the simpler one-way solvers.

Local propagation solvers can be optimized by making them incremental, so that only the elements affected by changes are recomputed. Efficient algorithms that recompute minimal numbers of dependencies include the DeltaBlue solver [FBMB90] and its successors such as SkyBlue [San94]. These solvers also have the interesting property that they are hierarchical [BFBW92]: they permit declaring certain constraints to be more important than others. The more important constraints are solved first, and less important constraints are used only when more important constraints leave unspecified degrees of freedom.

The popularity of local propagation solvers owes to their utility, their efficiency, and the fact that arbitrary functions can be computed in the dependencies. However, even the most sophisticated local propagation solvers have an important limitation: the methods are local. Propagation constraints solve systems of constraints by treating the constraints one at a time. Therefore, they can solve only sets of constraints for which there is an ordering such that constraints depend only on previous results. In graph terminology, propagation constraints can solve only systems that do not have cycles in their dependency graphs; in terms of equations, local propagation solvers can only solve triangular systems. Sophisticated local propagation solvers, such as SkyBlue [San94] can detect when their methods are insufficient, but cannot solve simultaneous equations.

For geometric problems, local propagation is insufficient. For example, it is unable to handle a pair of constraints that specify that a point is equidistant from two other points. This requires solving two constraints simultaneously. Solving two constraints simultaneously is the backbone of geometric constructions, as it permits intersecting figures as done in compass and straight-edge constructions. Compass and straight-edge constructions need only to handle pairs of constraints simultaneously as only two objects are ever intersected. However, these objects may depend on the results that are

propagated from previous computations.

Solving pairs of constraints simultaneously, for example to permit compass and straight-edge constructions, is an important special case that has been added to some propagation systems. Ruler and compass construction systems allow the user to explicitly order dependencies on constructions. Examples include Noma's system [NKK+88], LEGO [FP88], DoNALD [Ben89], and GIPS [CFV88]. More sophisticated systems have solvers that automatically plan the propagation paths. An example is the 2-forest propagation solver used in PictureEditor [KNK89]. Even more sophisticated solvers use rule-based systems to find sets of constraints that must be solved simultaneously and build propagation plans that use special case solutions for the simultaneous cases. Examples include Glenn Kramer's TLA solver [Kra90] and Aldefeld's system [Ald88]. Augmented term rewriting, introduced in Bertrand [Lel88] and also used in Siri [Hor91, Hor92], generalizes and formalizes the rule based propagation approach.

The inadequacy of propagation methods was a motivation for the differential approach. The success of the methods showed that constraints could be a useful tool in interactive systems. However, to achieve the desired flexibility of types controls and simultaneous combinations, a new approach to using numerical constraints would be needed.

2.2.2 Numerical Constraint Solving Techniques

Solving systems of linear equations for real numbers is a very well studied problem. Methods must address a wide variety of issues, including precision, stability, robustness, and efficiency. An excellent introduction to the field is provided in the text by Golub and van Loan [GL89]. Solving systems of non-linear constraints is much more difficult. In fact, for an argument that no general guaranteed method can exist, see Chapter 9 of Press et al.[PFTV86]. Generally, non-linear methods are designed for optimization, rather than equation solving[1]. Good, general tutorials on optimization methods are given by the texts by Fletcher [Fle87] and Gill, Murray and Wright [GMW81].

Some numerical methods operate like propagation methods in that they operate only on one constraint at a time. One example is relaxation which successively solves each constraint. Relaxation has been used in several early constraint-based graphical systems including ThingLab [Bor81] and Sketchpad [Sut63]. With relaxation, solving a constraint may break previously solved ones. The process iterates over all the constraints until a solution is found, or the solver gives up. Other methods that treat constraints individually include gradient (steepest) descent and penalty methods, surveyed by Platt [Pla92]. These simple methods do not work reliably for constraint problems and offer slow convergence even on problems that they do solve. The poor performance

[1] Chapter 9 of Press et al.[PFTV86] explains why the two problems are not equivalent, and argues why optimization is a more tractable problem.

of these simple solvers has discouraged many people from using numerical constraint methods for interactive graphics.

An important class of equation solving and non-linear optimization techniques operate by solving a sequence of linear systems. These iterative methods take a sequence of steps (hopefully) converging on a solution. At each iteration, a linear system is solved to determine what step should be taken. Numerical analysis texts, such as [PFTV86], introduce the basic varieties of these methods. The best known are Newton-Raphson methods, which have been used in a number of constraint-based graphics systems including Juno [Nel85] and Converge [Sis91].

Methods that use linear system solving are susceptible to problems when the constraints are redundant, inconsistent, or ill-conditioned. A standard method to cope with these problems is the technique known as regularization or damping. The technique will be discussed in Section 3.2.1, but briefly, it alters the linear system by limiting how much any particular equation can contribute to the solution. The method is the basis for the Levenberg-Marquardt method for solving non-linear equations [GMW81]. It has also been applied to the animation of articulated figures by Maciejewski [Mac90], and to the related problem of robotic control by Wampler [Wam86]. Damping methods are equivalent to the robust pseudo-inverse techniques of Nakamura [Nak91].

The "snakes" work of Kass et al.[KWT88] used numerical optimization to perform computer vision tasks. This work pioneered the use of optimization in interactive graphical applications. The system permitted a user to directly manipulate curves by resolving the optimization between each redraw. User interaction is created by including the user's input as part of the optimization objective, a technique that will be used in Section 3.5.

It is possible to view the methods of this thesis as a form of non-linear constrained optimization solving in which each iteration is displayed to the user. Unlike most solvers, the methods are more tuned towards generating smooth trajectories towards the goals, rather than getting to the goals as quickly as possible. Because the user can interact with the optimization process, a system can be interactively guided out of local minima. Mark Surles used a similar approach in his SCULPT system [Sur92a, Sur92c]. He used a different alternate Lagrange multiplier formulation than the one presented in Section 3.2.

2.2.3 Physical Simulation

The computer graphics community is becoming increasingly interested in using techniques of physical simulation for animation and modelling. Physically-based modelling and animation typically provide constraints in order to mimic the mechanical and structural relationships found in the real world.

A simple method for implementing physical constraints is by using springs to attach things together. This is called the *penalty method* because broken constraints are

penalized to pull them back to a solved state. To model more rigid constraints, the stiffness of the springs must be increased, making the equations of motion harder to solve numerically. The penalty method and its problems are reviewed by Platt [Pla92].

Lagrangian dynamics provides a constraint method that derives new equations of motion for constrained objects. A standard text used to introduce the methods is Goldstein [Gol80]. The methods effectively permit switching to a representation where the constraints are implicit. Unfortunately, the methods are impossible to automate for general cases as they require the ability to find algebraic solutions to systems of non-linear equations.

A method more applicable to computer graphics is the Lagrange multiplier method. In this method, constraints create reaction forces that cancel out any applied forces that would cause the constraints to be broken. The constraint forces are computed by solving a system of linear equations. Constraint stabilization methods, introduced by Baumgarte [Bau72], also use the constraint forces to inhibit the accumulation of numerical error due to drift.

Methods derived from constraint stabilization have been used by the computer graphics community to find initial solutions to constraints as well as to simulate their behavior. Barzel and Barr's dynamic constraints [BB88] use the stabilization forces to cause models to self assemble from various configurations. Platt and Barr's augmented Lagrangian constraints for flexible surfaces [PB88b] also used constraint stabilization, but attempted to avoid solving the linear system for the Lagrange multipliers by estimating them from previous values. In a later paper [Pla92], Platt explains why this was a bad idea, and provides a more standard Lagrange multiplier derivation of dynamic constraints.

Issues in using the Lagrange multiplier and constraint stabilization methods in interactive systems were discussed by Witkin, Gleicher and Welch [WGW90]. The system of Witkin and Welch [WW90] used the basic methods of [WGW90] to provide an interactive system for animating deformable objects. These techniques evolved into the differential methods of this thesis, first presented in [GW91a] and [GW92]. For the methods described here, constraint stabilization is accomplished by choosing controllers that continually "go towards" a value, rather than simply attempt to maintain a constant value by creating a 0 derivative. This will be described in Section 6.3.

The animation system of Witkin and Welch [WW90] provided a number of innovations that influenced the differential approach. The system permitted specification of objects' mass distributions in order to control an object's default behavior, an idea generalized into the use of metric definition in Section 3.4.1. The system also presented a predecessor to the controllers of the differential approach. The paper describes a vocabulary of controllers used to describe animation by specifying forces and impulses on objects over time.

Non-interpenetration or collision constraints are a special type of physical constraint. They differ from other mechanical connections in that they are represented

by inequality rather than equality equations. Methods for simulating collisions were first provided by Moore and Wilhelms [MW88] and Hahn [Hah88]. David Baraff has treated the physical simulation of collisions extensively [Bar92a], first introducing methods that properly handle collision and contact of polyhedral objects [Bar89], and then extending this result to curved surfaces [Bar90], surfaces with friction [Bar91a], and deformable surfaces [BW92]. Gascuel [Gas93] provides collision constraints for other types of deformable objects.

As discussed in Section 1.2.2, the differential approach of this thesis is a descendent of previous work in physical simulation, discussed in [WGW90]. The differential approach can be viewed as a form of physical simulation where the world has a different set of laws than the real world. Rather than follow Newton's $f = ma$ laws of motion, objects obey Aristotle's $f = mv$. Objects move only when pushed, rather than having inertia.

2.2.4 Inverse Kinematics and Dynamics

The problem of determining the configuration of parameters required to achieve desired values of object attributes is called inverse kinematics. The inverse kinematics problem is important to robotics as it is used to compute configurations of robots actuators required to achieve needed end-effector positions. The problem is, therefore, well studied, especially for the special case of most interest in robotics: articulated figures. An articulated figure is an object made of rigid links connected by joints.

Basic robotics texts, such as those by Craig [Cra86] or Paul [Pau81] present methods for solving inverse kinematic problems for articulated figures. Craig splits solution strategies into two broad classes, closed form solutions and numerical solutions. His text, like many others, dismisses numerical solutions "because of their iterative nature, numerical solutions generally are much slower than the corresponding closed form solution; in fact, so much so that for most uses we are not concerned with the numerical approach."

Inverse Kinematics techniques are becoming well known within the computer graphics community. Commercial systems, such as Softimage [Sof93] and Wavefront [Wav94] now permit users to manipulate articulated figures by positioning their end effectors. Badler and et al.[BMW87] describe extensions to standard inverse kinematics that permit positioning articulated figures by placing multiple constraints on them. Welman [Wel93] surveys inverse kinematics methods and discusses how to interactively position articulated figures using them.

More general methods for inverse kinematics use iterative numerical algorithms to solve the non-linear equations. Nakamura presents such an approach in his text [Nak91]. Nakamura's techniques are very similar to those of the differential approach, including his use of damping to handle singular systems.

Inverse dynamics, or robot control, is a related problem to inverse kinematics.

Rather than solving for configurations, the methods determine forces and torques required to achieve desired effects. Inverse dynamics has been explored for use in computer animation. Armstrong et al.[AGL87] and Wilhelms [Wil87] present systems that use inverse dynamics to aid in the animation of articulated figures. Issacs and Cohen's DYNAMO system [IC87] combines inverse dynamics with kinematics using a general formulation that can handle objects other than articulated figures.

2.2.5 Numerical Methods for Interactive Graphics

As will be further discussed in Chapters 4 and 5, there are several issues in employing numerical techniques in interactive systems. The two main ones are fast solving and dynamic definition of the problems.

One issue in employing numerical computations in interactive systems is that the derivatives of the functions representing constraints must be computed. While there are several methods for computing derivatives, such as symbolically creating the equations or estimating the values with finite differences, the methods of Automatic Differentiation have been shown by [Gri89] to be at least as efficient and accurate. An introduction to Automatic Differentiation provided by Iri [Iri91], and a survey of tools is provided by Juedes [Jue91].

Research in Automatic Differentiation focusses on the development of compile time tools for large problems [BGK93]. For computer graphics, Automatic Differentiation techniques were developed to operate on expression graphs explicitly represented in program data structures, as will be discussed in Chapter 5. These methods permit the functions being differentiated to be dynamically defined. An implementation of the techniques was employed in the system built for Spacetime Constraints [WK88]. A later system using the methods is Kass' CONDOR [Kas92] which permitted the user to interactively specify constrained optimization problems by direct manipulation of expression graphs.

My implementation of Automatic Differentiation, called Snap-Together Mathematics, encapsulated the methods into an application independent toolkit and is discussed in Chapter 5. The first version of Snap-Together Mathematics was introduced as part of work on interactive physical simulation [WGW90]. The first C++ toolkit for Snap-Together Mathematics was detailed in [GW91b]. Based on this paper, Kaufman reproduced the system [Kau91]. A variant of the original Snap-Together Mathematics was used inside of the Briar drawing program (Section 9.1), and evolved into the current implementation introduced in [GW93] and described in Chapter 5.

The critical performance issue in most numerical constraint methods is solving a linear system, as discussed in Chapter 4. Exploiting sparsity, the fact that a matrix contains many 0 elements, is a standard technique for speeding the solution of linear systems. The text by Duff et al.[DER86] provides an introduction to the techniques. For the differential approach, the direct methods, like those discussed Duff et al, are

less appropriate than iterative methods. Detailed discussions of iterative methods, and specifically the Conjugate-Gradient methods used in this thesis, are provided by [PS82] and [She94].

Steven Sistare's thesis describing Converge [Sis90] provides an analysis of the performance issues in using numerical techniques in an interactive drawing system, and includes methods for dynamically selecting linear system solvers and partitioning the constraint problems. Mark Surles' work on interactive manipulation of protein molecules extensively treated the performance issues in solving the linear systems involved in solving the optimization problems [Sur92b, Sur92c]. Because his task was to manipulate predefined models that had a very specific structure, his methods do extensive pre-analysis. The structure of the matrix found in chemistry problems permits solutions with linear time complexity.

2.3 Graphics Toolkits

For a variety of reasons, constructing interactive applications is an extremely difficult task [Mye94]. In order to aid with this process, a variety of tools, surveyed in [Mye93], have been developed. The most often used are graphical interface toolkits.

Basic graphics toolkits, such as GL [Sil91], PHIGS [Com88], and X [SG86], provide drawing primitives and basic elements for interaction, such as events and windows. Graphical interface toolkits support graphical applications by providing high level support for interaction techniques and graphical object management, aiming to insulate the programmer from low level details such as window management as much as possible. Such toolkits have become a part of the construction of almost all graphical applications. However, most toolkits leave the majority of the work of graphical editing to the applications programmer.

Some research tools, such as ArtKit [HHN90], Garnet [MGD+90] and Coral [SM88] provide support for graphical editing in addition to the more typical buttons and sliders. Tools specifically designed to support 2D graphical editors include Unidraw [VL89], ArtKit [HHN90], MEL [Hil91] and GRANDMA [Rub91]. Rendezvous [HBP+93] is specifically designed for creating multi-user graphical editing applications. All of these tools provide mechanisms for creating direct manipulation operations.

More recently, toolkits have been developed to support 3D graphical applications at a higher level than low level graphics packages such as GL or PHIGS. Such toolkits are almost always object oriented, and provide high level abstractions of interaction techniques. Examples of such toolkits include MR [SLGS92], Inventor [SC92], UGA [CSH+92], BAGS [ZCW+91], Alice [PT94], VB2 [GBT93], and GROOP [KW93].

Many of the toolkits mentioned contain support for advanced interface techniques, such as ArtKit's snapping, GRANDMA's gesture recognition, or Inventor's 3D manipulators. However, no previous high-level toolkits provide non-linear constraints or

interaction techniques for both 2D and 3D applications. Similarly, many user interface toolkits use constraint techniques to help programmers build interactive applications, as discussed in Section 2.1.2. In all cases, constraint methods are limited to propagation, and the focus is on abstractions to help programmers, not necessarily to provide constraints to the users.

Providing an embedded interpreter in an interactive application is not an uncommon technique. The utility of such extension languages is discussed in [BG88], which describes the success of the EMACS editor. Graphics toolkits which center around such interpreters include Tk [Ous91], MR [SLGS92], Alice [PT94], UGA [CSH+92], and ezd [Bar91b].

Previous toolkits have attempted to aid in the development of interaction techniques, and their incorporation into systems. For example, Garnet provides a basic set of interactors [Mye90] from which more complex behaviors can be constructed. UGA [CSH+92] and Alice [PT94] allow prototyping 3D interaction techniques procedurally. GITS [OA90] defines interaction techniques with constraints; however it is limited to the design of 2D widgets and it precompiles constraint solutions. In [ZHR+93], interaction techniques are interactively linked together in a constraint-like fashion to build more complex 3D widgets.

2.4 Interaction Techniques and Applications

Development of 3D interaction techniques was a major motivation for the differential approach and is the source of most of the examples in the thesis.

2.4.1 Manipulating 3D Objects

Sketchpad's 3D successor, Sketchpad III [Joh63], introduced graphical manipulation of 3D objects and first faced the issues of manipulating 3D objects with 2D pointing devices. Since then, many researchers have explored the issues. Catalogs of interaction methods are provided by Evans et al.[ETW81], Nielson and Olsen [NO86] and Osborn and Agogino [OA92]. The problem of specifying a 3D rotation using a mouse has received close attention, such as the work of Chen et al.[CMS88] and Shoemake [Sho92]. Techniques which rotate and translate objects using references to other points of interest in the scene are explored by Bier [Bie86] and [Bie90]. Methods based on interactions between pairs of objects are provided by [Ven93].

In order to make interfaces easier to learn and use, designers explore how to make them self-revealing. Houde [Hou92] considers iconic handles and movements based on the objects' meanings. 3D Widgets [CSH+92] use graphical objects which disclose potential behavior in the same view as the objects they manipulate. The authors have subsequently built an interactive tool for rapidly prototyping these widgets [ZHR+93].

Inventor [SC92] is a popular toolkit for constructing 3D applications that employ a widget style interface.

2.4.2 Controlling Virtual Cameras

The problem of specifying a viewing transformation or virtual camera configuration is an important problem for 3D graphics. This work deserves mention here not only because it is a problem to which the differential approach will be applied to yield interesting results (Section 8.1.4), but also because the work typifies the general problems that the differential approach is designed to address.

Most camera formulations are built on a common underlying model for perspective projection under which any 3-D view is fully specified by giving the center of projection, the view plane, and the clipping volume. Within this framework, camera models differ in the way the view specification is parameterized. These parameterizations are typically designed to provide controls that are useful for either interaction or interpolation.

Much of the work on interactive camera placement in computer graphics has been concerned with direct control of these standard parameters. Several researchers have addressed the problem through the use of 3-D interfaces, including six degree-of-freedom pointing devices [WO90, TBGT91, BMB86] and more specialized devices such as steerable treadmills [Bro86]. Issues involved in using the standard LOOKAT/LOOKFROM model to navigate virtual spaces are considered by [MCR90]. In [DGZ92], the LOOKAT/LOOKFROM model is embedded in a procedural language for specifying camera motions.

The difficulty with using camera parameters directly as controls is that no single parameterization can serve all needs. For example, sometimes it is more convenient to express camera orientation in terms of azimuth, elevation and tilt, and other times in terms of a direction vector. These particular alternatives are common enough to be widely available, but others are not. A good example involves the problem, addressed by Blinn [Bli88b] of portraying a spacecraft flying by a planet. Blinn derives several special-purpose transformations that allow the image-space positions of the spacecraft and planet to be specified and solved for the camera position. The need for this kind of specialized control arises frequently, but we would rather not derive and code specialized transformations each time it does. The differential approach permits using these interaction techniques without deriving the inverse transformations.

Registering graphical objects and a real image by recovering camera parameters is considered in Section 8.2.4. Problems involving the recovery of camera parameters from image measurements have been addressed in photogrammetry[2], computer vision, and robotics. All of these are concerned with the recovery of parameter values, rather than time derivatives. Algebraic solutions to specific problems of this kind are given

[2] Also see chapter 6 of [Sch59] for amazing mechanical solutions to photogrammetry problems.

in [Mof59] and [Gan84], while numerical solutions are discussed in [Low80, Gen79, McG89]. In [TTA91], constrained optimization is employed to position a real camera, mounted on a robot arm, for the purpose of object recognition. Factors considered in the optimization include depth of field, occlusion, and image resolution. The use of constrained optimization for camera placement in animation is proposed by Witkin et al.[WKTF88].

2.4.3 Controlling Lighting and Surface Properties

Shadows play a particularly important role in 3d images. They contribute greatly to viewers' abilities to perceive depth [Wan92]. Techniques for displaying special cases of shadows can be implemented in real time on graphics workstations [Bli88a, Hud92], and the most sophisticated graphics hardware is even capable of drawing more general shadows in real time [SKvW$^+$92].

Controlling scene parameters by directly manipulating illumination effects has been explored by several researchers. A desire for appearance-based manipulation is expressed in [vWJB85]. Poulin and Fournier [PF92] describe techniques for positioning light sources by specifying the positions of specular highlights and shadows. Dragging drop shadows on the floor and walls is used to position objects in [HZR$^+$92]. Hanrahan and Haeberli [HH90] discuss techniques which allow users to paint on images and have the surface's colors updated appropriately. Painting with Light [SDS$^+$93] permitted controlling intensities of light sources in a similar fashion. Kawai, Painter and Cohen's Radioptimization [KPC93] permitted controlling light sources by specifying the desired lighting on various surfaces. The methods used a constrained optimization on the results of a radiosity computation.

The science is in the technique, all the rest is just commentary.

— Allen Newell
SCS Distinguished Lecture, Dec, 1991

Chapter 3

Differential Techniques

This chapter introduces the basic techniques required to implement the differential approach. We begin by reviewing how graphical manipulation can be viewed as an equation solving problem. To solve these equations differentially, we will solve a constrained optimization that computes rates of change of the parameters given the desired rates of change of the controls. Basic methods for solving these optimizations will be discussed.

Constrained optimization computes the rate of change of object parameters. To determine the objects' trajectories, an ordinary differential equation (ODE) must be solved from an initial boundary value. Some of the basic issues in solving such equations as well as some methods will be introduced.

Additional flexibility is provided by adding additional terms into the constrained optimization problems. This is used to provide default behaviors for objects and to permit the creation of soft controls that can be used to express preferences.

The chapter concludes with an alternate solving method that sacrifices generality for simplicity, a simple example worked through in detail, and a summary of the symbols and methods discussed. The subsequent chapters describe how these methods can be implemented in an efficient and flexible manner.

3.1 The Differential Optimization Problem

In the introduction, the basic notion of treating graphical manipulation as an equation solving problem was introduced. We control graphical objects by specifying what happens to the values of selected attributes called controls. These controls are defined by functions,

$$\mathbf{p} = \mathbf{f}(\mathbf{q}), \tag{3.1}$$

where $\mathbf{q}$ is the state vector of the objects, $\mathbf{p}$ is the vector of values of the controls, and $\mathbf{f}$ is the function that defines the controls. A full table of all mathematical symbols used in this chapter is provided on page 62.

As was discussed in Section 1.1.5, it is not practical to solve Equation 3.1 for $\mathbf{q}$ given $\mathbf{p}$. Instead, we take a differential approach to the problem, as described in Section 1.2. Rather than specifying values for the controls, we will specify how they are changing over time. At particular instants in time, we compute how the state vector must change in order to achieve the desired changes in the controls.

Given a particular instant in time, the value for the state vector at that instant ($\mathbf{q}$), and the desired values for the rate of change for the controls ($\dot{\mathbf{p}}$), we must compute the necessary rate of change of the state vector($\dot{\mathbf{q}}$). We call this problem the *differential optimization*. Since the value for the state and the control function are given, we also know the value of the controls at the instant the optimization is to be solved.

Our need to deal with the time derivatives of the controls and state variables leads us to take the derivatives of each side of Equation 3.1 to yield

$$\dot{\mathbf{p}} = \frac{d\mathbf{p}}{dt} = \frac{d\mathbf{f}(\mathbf{q})}{dt}. \tag{3.2}$$

Applying the chain rule yields

$$\dot{\mathbf{p}} = \frac{\partial \mathbf{f}}{\partial \mathbf{q}} \frac{d\mathbf{q}}{dt}. \tag{3.3}$$

For the general case of a vector of control functions, the derivative is a matrix called the *Jacobian,* which is the matrix $\partial \mathbf{f}/\partial \mathbf{q}$ and is denoted by $\mathbf{J}$. Using this notation, we get

$$\dot{\mathbf{p}} = \mathbf{J}\dot{\mathbf{q}}. \tag{3.4}$$

Like the controls themselves, the Jacobian $\mathbf{J}$ is a function of the state variables. The matrix of functions that compute the elements of the Jacobian can be determined by differentiating the control functions with respect to the variables. Since the values for the state variables are given, the value of the Jacobian is effectively given as well. Methods for computing the Jacobian efficiently will be discussed in Section 5.2, but for now, we simply assume we have some method to compute the Jacobian from the variables, and treat is as if it were a given as well.

Since $\mathbf{J}$ is a known matrix, Equation 3.4 is a *linear* equation, even though $\mathbf{p}$ is a non-linear function of $\mathbf{q}$. The differential approach has replaced the multi-dimensional non-linear root-finding problem with a linear system and an ordinary differential equation. Unlike non-linear equations, for which good solving techniques are unlikely to exist, linear systems are relatively easy to solve.

3.1.1 Underdetermined Cases

Unless enough controls are specified to uniquely determine a solution, Equation 3.4 will be underdetermined. There will be many possible ways for the state vector to change to achieve the desired changes in the controls. As discussed in Section 1.1, the system must select one of the ways for things to change. We must use some heuristic to

pick a solution, because we lack any information as to what is desired. The rule chosen for the differential approach is to minimize the amount that the configuration changes, or, more precisely, to minimize the rate of change of the configuration. That is, if the user's controls don't ask for something to change, the system should avoid changing it. This leaves open a variety of ways to measure change that will be explored in Section 3.4. Since the rate of change of the configuration will be a linear function of the rate of change of the variables, its magnitude will be a quadratic function, which we denote by g.

When the linear system of Equation 3.4 does not uniquely determine $\dot{\mathbf{q}}$, it provides constraints on its possible value. To determine the particular value of $\dot{\mathbf{q}}$, we must solve the problem

$$\text{minimize } E = g(\dot{\mathbf{q}}) \text{ subject to } \dot{\mathbf{p}} = J\dot{\mathbf{q}}. \tag{3.5}$$

That is, we have cast the problem as a constrained optimization: minimize the value of a quadratic objective function of $\dot{\mathbf{q}}$, subject to the linear constraints that the controls are met. In the following sections, we discuss solution methods using standard techniques that meet the needs of differential manipulation.

3.2 Solving the Differential Optimization

The linear/quadratic constrained optimization problems are a standard class of problems for which a wide range of techniques have been developed. Good surveys can be found in texts such as [Fle87] and [GMW81]. A standard technique is the Lagrange multiplier method. A form of it is reviewed here for use in the differential approach.

To begin, we consider minimizing a specific quadratic objective function, simply minimizing one half the magnitude of $\dot{\mathbf{q}}$ squared. The value of $\dot{\mathbf{q}}$ that minimizes that is the same value that minimizes the magnitude of $\dot{\mathbf{q}}$. The specific constrained optimization problem we consider in this section is then

$$\text{minimize } E = \frac{1}{2}(\dot{\mathbf{q}} \cdot \dot{\mathbf{q}}) \text{ subject to } \dot{\mathbf{p}} = \mathbf{J}\dot{\mathbf{q}}. \tag{3.6}$$

We will consider the general case of quadratic objectives in Section 3.4.

To provide an intuition for how Lagrange multiplier methods work, consider an extremely simple case: a particle in 2 dimensions, with its state represented as its Cartesian coordinates, $\mathbf{q} \equiv \{x, y\}$. We will place a control on the particle that is its distance to the origin, $p = f(\mathbf{q}) = x^2 + y^2$. Suppose we specify $\dot{p}$ to be 1.

As shown in Figure 3.1, there are many possible values for $\dot{\mathbf{q}}$ which achieve the desired value for $\dot{p}$. In this case, it is clear to see that the one with smallest magnitude is the one which is in the same direction as the gradient of f. Any component of $\dot{\mathbf{q}}$ not along this line will not be helping to achieve the desired controls. We can therefore restrict $\dot{\mathbf{q}}$ to be some multiple of the gradient, that is, $\dot{\mathbf{q}}$ can be expressed as a scaling factor times the gradient. This scaling factor is called the *Lagrange Multiplier.*

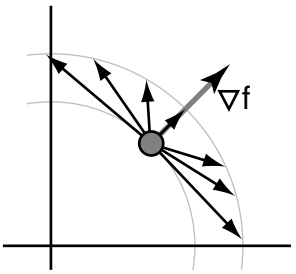

Figure 3.1: A point on the plane with a radial control. Many possible values of $\dot{\mathbf{q}}$ will yield the desired value of $\dot{p}$. The one with least magnitude has the same direction as the gradient of f.

If there were multiple controls, each would make a contribution to $\dot{\mathbf{q}}$. For each control, the contribution is some multiple of its gradient. We therefore have a vector of Lagrange multipliers, which we denote by $\boldsymbol{\lambda}$. $\dot{\mathbf{q}}$ is determined by the linear combination $\mathbf{J^T}\boldsymbol{\lambda}$.

More formally, to be a solution to the constrained minimization problem, $\dot{\mathbf{q}}$ must satisfy two criteria. First, $\dot{\mathbf{q}}$ must satisfy the linear constraints, given by Equation 3.4. Secondly, $\dot{\mathbf{q}}$ must minimize E as much as possible, subject to the constraints. In the case of an unconstrained minimization, we would require that the gradient $\partial E/\partial\dot{\mathbf{q}}$ vanish, meaning that there is no direction to change $\dot{\mathbf{q}}$ that would result in a lesser value for E. With constraints, there might be a way to change $\dot{\mathbf{q}}$ to further minimize E, but only if these changes are prohibited by the constraints. That is, if the gradient of the objective function is not zero, it must lie in the row space of the constraint gradients. This requirement is expressed by defining the objective function gradient to be a linear combination of the constraint gradients,

$$\frac{\partial E}{\partial \dot{\mathbf{q}}} = \mathbf{J}^T\boldsymbol{\lambda}, \tag{3.7}$$

for some value of $\boldsymbol{\lambda}$. The vector $\boldsymbol{\lambda}$ is an intermediate result which we call the *Lagrange multipliers.*

In the case of the simple objective function of Equation 3.6, the gradient $\partial E/\partial\dot{\mathbf{q}}$ is simply $\dot{\mathbf{q}}$, giving

$$\dot{\mathbf{q}} = \mathbf{J}^T\boldsymbol{\lambda}. \tag{3.8}$$

Substituting this into Equation 3.4, gives

$$\dot{\mathbf{p}} = \mathbf{J}\mathbf{J}^T\boldsymbol{\lambda}, \tag{3.9}$$

a linear system which can be solved for its one unknown, $\boldsymbol{\lambda}$. This intermediate result can be substituted back into Equation 3.8 to yield the desired final result, $\dot{\mathbf{q}}$.

3.2.1 Over-determined Cases

To this point, we have focussed on techniques that handle the cases where an insufficient set of controls are specified to uniquely determine a solution. We now must consider the problem of handling cases where too many controls specify the solution. Such cases may involve redundant controls, where multiple controls all specify the same solution, or conflicts, where there are no solutions to all of the controls.

Conflicting controls are obviously a problem as there is no solution which will meet the controls. However, redundant controls manifest themselves in exactly the same way. Consider a system subject to two identical controls p_1 and p_2. The net result should be that $\dot{\mathbf{q}}$ moves in the manner specified. But $\dot{\mathbf{q}}$ is created by the sum of the contributions of p_1 and p_2. How much does each contribute? Does p_1 contribute a little and p_2 a lot? Does p_1 contribute a huge positive amount and p_2 a huge negative amount?

When controls are over-specified, whether their values conflict or not, they cause the Lagrange multipliers to be under-specified. The matrix $\mathbf{JJ^T}$ will be singular. Redundant or conflicting controls are inevitable, and are notoriously hard to detect. It is important that our solution method be robust in the face of these singularities, and that it will do something reasonable with conflicts.

One approach to handling the over-constrained cases would be to employ a linear system solver which could handle Equation 3.9 even when the matrix is singular. For example, singular value decomposition (SVD) [PFTV86] could be used. The SVD has many attractive properties, for example it provides information as to which controls are redundant. Unfortunately, SVD is expensive to compute. In contrast, if we can restrict the problem so that the solver only needs to solve non-singular systems, we can exploit this property to solve them efficiently, as will be discussed in Chapter 4.

Rather than force the solver to handle singular matrices, we will instead modify the matrices so that they have a unique solution. We will not get an exact solution to the original linear system, but we are trading accuracy for improved behavior in bad cases. Because we are interested in interaction, rather than high accuracy quantitative methods, we will make such tradeoffs often, as discussed further in Section 4.5.

The technique for making the matrices non-singular is called *damping*. The basic intuition is that we generally prefer to avoid large values for the Lagrange multipliers, therefore, in cases where the Lagrange multipliers are undetermined, we should minimize their magnitude. The derivation here most closely follows that of Nakamura's derivation of a robust pseudo-inverse [Nak91].

In cases where the controls are over-determined, the solver will not be able to achieve all the desired values for them. Instead, we must settle for getting as close as possible, that is, to satisfy them in a least squares sense. To find Lagrange multipliers that achieve this minimum, we minimize $1/2(\mathbf{J}^T\boldsymbol{\lambda} - \dot{\mathbf{q}}) \cdot (\mathbf{J}^T\boldsymbol{\lambda} - \dot{\mathbf{q}})$, In addition, we would like to minimize the magnitude of $\boldsymbol{\lambda}$, although since this is not as important, we can scale this term by a small amount, which we will call μ. The function we wish

to minimize is

$$E = \frac{1}{2}(\mathbf{J}^T\boldsymbol{\lambda} - \dot{\mathbf{q}}) \cdot (\mathbf{J}^T\boldsymbol{\lambda} - \dot{\mathbf{q}}) + \mu(\boldsymbol{\lambda} \cdot \boldsymbol{\lambda}). \tag{3.10}$$

The minimum of this quadratic is found by differentiating with respect to $\boldsymbol{\lambda}$, and setting that equal to 0, yielding

$$0 = \mathbf{J}\mathbf{J}^T\boldsymbol{\lambda} - \mathbf{J}\dot{\mathbf{q}} + \mu\mathbf{I}\boldsymbol{\lambda}, \tag{3.11}$$

where $\mathbf{I}$ is the identity matrix. Recalling Equation 3.4, a little rearrangement yields

$$\dot{\mathbf{p}} = (\mathbf{J}\mathbf{J}^T + \mu\mathbf{I})\boldsymbol{\lambda}, \tag{3.12}$$

a variant of Equation 3.9 which has small amounts added to the diagonal of the matrix.

Rather than having a single scaling factor for the magnitude of the vector of Lagrange multipliers, we could have an individual one for each individual multiplier. This would enable damping selectively, or to damp some controls more than others. Selective damping allows the creation a a limited constraint hierarchy: if two constraints conflict, and one is damped but the other is not, the undamped constraint will dominate the damped one. When two conflicting controls are both damped, their effects are blended. By individually adjusting their damping values, the controls can be weighted. The larger the damping value, the less weight the control receives.

The damping technique presented here has three major drawbacks. First, it penalizes large values of the multipliers whether they are underdetermined or not. This can cause a problem when the multipliers legitimately need to be large to satisfy the desired controls. Secondly, damping is applied whether there are conflicting controls or not. Finally, it introduces a new dimensionless parameter μ. Because it has no real meaning to the original problem, values for it are difficult to determine. For the differential approach, damping values must be determined empirically.

3.3 Solving the Differential Equation

The methods of the previous section permit us to compute the rates of change of the state vector. We now consider how to use the rates to find the trajectories of the configurations over time. We must solve the problem of computing the trajectory of the state given its initial value and time derivatives, a problem of solving an ordinary differential equation (ODE) from an initial boundary condition. Here, we provide a brief introduction to handling this problem in the context of the differential approach. For a more complete, but still practical, introduction to ODE solution methods, see Chapter 15 of [PFTV86].

The value of $\mathbf{q}$ is actually is a function of time, defined by a function that computes its time derivative. The form that we have this function defined in is

$$\dot{\mathbf{q}} = \mathbf{f}'(\mathbf{q}, t), \tag{3.13}$$

where $\mathbf{f}'$ is found by solving the differential optimization. Given the value for $\mathbf{q}$ (which corresponds to $\mathbf{q}(t)$ for some time t), we can compute $\dot{\mathbf{q}}$. This is a standard form for an ODE.

For the types of problems we encounter with the differential approach, we cannot solve the ODE in closed form. Instead, we must solve it numerically by discretizing time into a series of small steps. In computing a step, the following problem must be solved: given the state at the current time, $\mathbf{q}(t)$, find the state at some time in the future, $\mathbf{q}(t+\Delta t)$. The time derivative $\dot{\mathbf{q}}$ (i.e. the result of the differential optimization) does not directly provide the solution to this problem. It only specifies how the state is changing at the instant that it is computed. The problem of updating the state given the ability to find its time derivatives is solving an ordinary differential equation from an initial boundary condition.

Solving the ODE is difficult because when we perform an evaluation to find $\dot{\mathbf{q}}$ for a particular $\mathbf{q}$, we are only finding out about a particular instant in time . We have no information about the future. $\dot{\mathbf{q}}$ might remain constant for the duration of the step, but it might also change drastically over the course of the step.

The simplest method for solving an ODE is to find $\dot{\mathbf{q}}$ at the beginning of the step and assume it remains constant over the course of the step. This is known as Euler's Method, and has the simple update rule of

$$\mathbf{q}(t + \Delta t) = \mathbf{q}(t) + \Delta t\dot{\mathbf{q}}(t). \tag{3.14}$$

Euler's method approximates $\mathbf{q}(t)$ with as a piecewise linear function. The size of each piece is the step size. If the step is too large, the approximation will not be good. Notice that each step requires computing a new $\dot{\mathbf{q}}$ by solving the differential optimization.

To understand what is meant by "good" in ODE solving within the context of the differential approach, consider a simple example. Once again, we will use a point in the plane, however, this time, we will select a control that is its angular position about the origin. Suppose we provide a desired velocity for this control of 1 unit per unit time, the starting configuration has the point a unit distance from the origin along the positive x axis, and the specified derivative always points tangent to the circle. As time progresses, we will expect the point to move around the origin in a circular path.

If an Euler's method ODE solver is applied to this example, the problems of ODE solving are quickly apparent. At the initial position on the x axis, the gradient of the control points vertically. Any step in this direction will lead the point off the circle it is expected to follow around. As more and more steps are taken, the point will continue to spiral away from the circle, speeding off the page, as shown in Figure 3.2.

Because of error in approximation, the point spirals outward over time. However, if smaller step sizes are taken, the behavior is better. That is, the point spirals outward more slowly, better approximating the expected circle. This is shown in Figure 3.3. In fact, by going more slowly, we may reach a desired destination more quickly because we are less likely to overshoot or drift away from the target. Going slowly can be

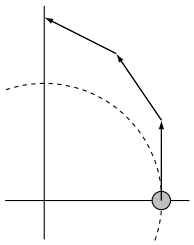

Figure 3.2: A point on the plane with a control that drives it tangent to a circle around the origin. Although it should (ideally) travel in a circular path, ODE solver error causes it to spiral off the page.

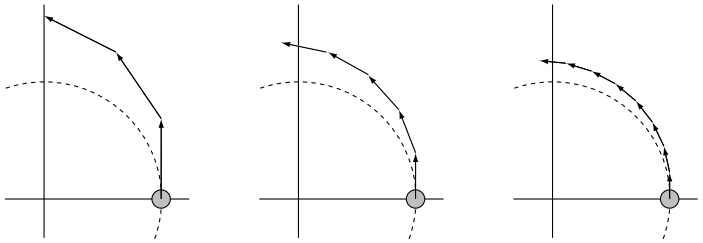

Figure 3.3: A point is pushed in a direction tangent to a circle about the origin, using an Euler ODE solver and various step sizes. In all cases, the point spirals away from the target circle, although with small step sizes, the point tracks the circle better.

accomplished two ways, either by reducing the duration of the step or by reducing the velocities.

In solving the ODE, there is effectively a speed limit. If an object attempts to change faster than this, it may speed out of control or even miss its destination entirely. For a given velocity, solving can be made more stable by reducing the step size, in the limit of infinitesimally small step sizes, ODE solving will be exactly correct. However, since each step may require significant computation, the number that can be executed is limited. Alternatively, this speed limit can be viewed with a constant step size: for a given step size, how fast can an object go without becoming unstable. If we think of steps as taking some fixed amount of time to compute, this translates directly to an apparent velocity in the image.

The speed limit given by ODE solving varies according to a number of factors. Most significantly, it depends on the path that the objects take. The more non-linear the function is, the more poorly the linear approximation will fit it. At the extreme, if an object is truly moving in a line, Euler's method achieves the exact motion, and there is no speed limit. In a sense, the speed limit can be viewed as a restriction on the types of controls used to define the motion: given that the step size is fixed, how badly non-linear a control can be used and still have the object move at a reasonable rate.

If we take smaller steps to achieve better performance, we might use multiple ODE solver steps for each redraw. For example, if we would like to maintain 10 frames per second and updating the image or solving the differential optimization takes 30ms, we might use two Euler steps between redraws. As we will see in Chapter 4, typically the differential optimization is the most time consuming part of the process, and becomes more so as the problems grow larger. With a faster computer, the time to compute each step will be decreased, so more steps can be computed per redraw, effectively raising the speed limit.

Using multiple samples per step is what is called *multi-step solving*. We might phrase the problem as follows: the starting point (t and $\mathbf{q}(t)$), compute $\mathbf{q}(t + \Delta t)$ as well as possible using n samples. Using two Euler steps has $n = 2$, and uses a 2 piece piecewise linear approximation.

Given that we have some number of samples that we can make in a step, we can consider how to best use these samples to approximate $\mathbf{q}(t)$. For example, if we can take 2 samples, we might use a 2 piece linear approximation by taking two Euler steps. Alternatively, we might use these same two samples to fit a parabola. This would be called a *2nd-order* method. The particular case of a parabola is simple to create, since a parabola has a linear function for its derivative. This is effectively done by using the initial step as a trial step, evaluating the derivative at this point, and using this for the duration of the step. This is called the *midpoint method* or the 2nd-order Runge-Kutta method. It is applied to the example problem in Figure 3.4.

According to Press et al.[PFTV86], the most popular multi-step method is the 4th-order Runge-Kutta method. As implied by the name, it uses 4 evaluations per step. According to the literature, this method is generally perceived to be superior to higher order methods. The 4th order Runge Kutta method has been the preferred solver for the prototype implementations in this thesis.

A higher order method is only better than taking a larger number of lower order steps if it provides a higher speed limit for the same number of evaluations. This will, of course, depend on the problem to be solved. However, in practice, the 4th-order Runge-Kutta method seems to be a good method for implementing the differential approach. Empirically, it usually performs at least as well, but sometimes substantially better, than taking 4 small Euler steps, or 2 Runge-Kutta 2 steps.

There are many other multi-step methods. Predictor-Corrector techniques [PFTV86] are another popular strategy. Such methods use past steps to predict future values and then correct for the error of the prediction. However, these methods are difficult to apply in dynamic settings because the dynamic nature of the problems make it difficult to maintain a history to use in prediction. Often, there will be no history so some technique like Runge-Kutta will be needed to start the process.

For the prototype implementations of this thesis, fourth order Runge-Kutta and Euler's methods solvers are used.

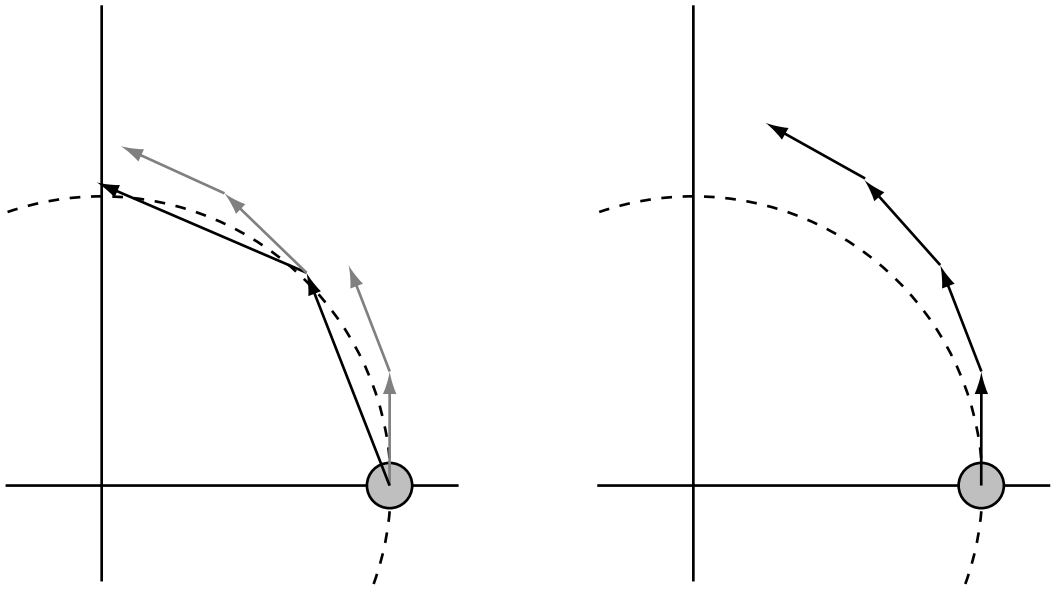

Figure 3.4: Left, a second order Runge-Kutta method, and right, an Euler's method are used in the example problem of Figure 3.2. The evaluations used by the Runge-Kutta solver are shown in grey. Notice how the Runge-Kutta solver stays closer to the circle using the same number of evaluations as the Euler solver.

3.3.1 Adaptive Step-Sizes

So far, we have considered solving an ODE with fixed step sizes. Instead, we might consider adapting the step size to the problem. When a step is made, it could be checked to see how good it was. If it caused an unacceptably large amount of error, it could be redone with a smaller step size. Adaptive step size methods have the advantage that they can slow down to accurately handle problems when they become difficult. Also, because they check their results, they are less likely to cause bad errors.

Adaptive step size methods have some severe drawbacks when used with the differential approach. The most significant problem is that they are continually adjusting the step size which alters the amount of computation required to advance simulation time a specified amount. If the computation rates are fixed, the apparent velocities of objects will fluctuate. This can be disconcerting to the user. Also, the extra evaluations to perform checks and computing alternate steps might be better spent on making more steps since error correction is built into the controls, as controllers can adjust their values in response to what is happening as will be discussed in Section 6.3. For example, in the example of the previous section, if the user really cared about the point staying on the circle, an additional control that maintained this would be used.

The differential approach provides some interesting opportunities for employing adaptive step sizes. Standard methods for ODE solving treat the equation as a black

box, that is they cannot get any information about the problem other than asking for evaluations of $\dot{\mathbf{q}}$. Standard adaptive ODE solvers employ methods such as taking the same step with a higher order method to compare with.

With the differential approach, we have more information about the problem we are solving. In particular, the functions that define the controls provide measures of error. For example, if a controller is meant to keep a control at a particular value, at the end of the step it can be checked to insure that the control has not changed too much. Each different control might have its own way of defining what an acceptable amount of error is. I call this *semantic adaptation* because it adapts based on the meaning of the problem. I have experimented with some simple semantic adaptation of step size, simply reducing the step size when a control has a value that the system finds unacceptable. The method works as follows: a certain set of controls are monitored. When a step is computed, the monitored controls are examined. If any exceed a specified error limit, the step size is shortened.

A different type of semantic adaptation is to use a different step when problems occur. One useful variant of this is the cleanup step. In an application like constraint-based drawing or mechanism simulation, there is typically some small number of controls that cause motion and a potentially larger number that represent constraints. A cleanup step is an extra step that is run only with the constraints. It is used when the pulling controls have broken the other constraints to cause them to get back to their desired configuration.

3.4 Generalized Objective Functions

The optimization objective determines which solution will be given in under-determined cases. By selecting different optimization objectives, different default behaviors can be given to objects. To this point, we have only consider one optimization objective: one half the magnitude of the state vector derivative squared. This section considers other objective functions.

The types of default behavior that we consider in the Differential Approach can be summarized by the idea that objects should not change unless a control causes them to change, and that when an object changes to achieve what is specified by a control, it should do so by changing as little as possible. By altering how we measure change, we can control the default behavior or feel of an object. For an example, consider manipulating a line segment by moving one of its points, as shown in Figure 3.5. Depending on the metric of change, the line segment will behave differently. In all cases, the line segment achieves what is specified by the controls. However, by selecting an appropriate metric, the programmer can create a desirable default behavior. An appropriate metric is not essential since if there was something that was important, it could be specified with a control. But, properly defined objectives can alleviate the need for extra

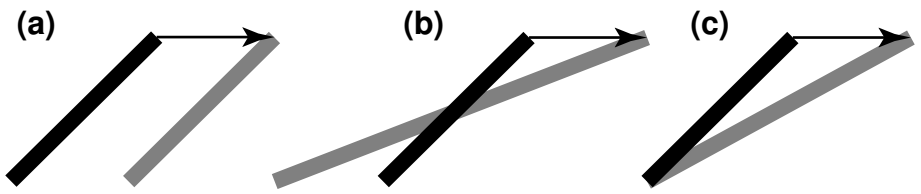

Figure 3.5: A line segment is dragged by controls that specify the position of the upper point. Different objective functions provide different behaviors: a) change in length and orientation are minimized; b) change in the position of the center is minimized; c) change in the position of the lower point is minimized.

specifications.

The simplest metric of change is the magnitude of the rate of change of state vector, $\dot{\mathbf{q}}$, as was used in Section 3.2. This simple objective has been used up to this point, and is sufficient for a wide variety of applications. It has a drawback: it causes the parameterization to affect the behavior of the object. Using the parameterization violates the goal of separating manipulation from representation. Even simple decisions, such as whether to represent an angle by degrees or radians, can affect an object's behavior [Wit89a]. This may be a serious problem, or an opportunity. It means that we can choose which interactive behavior we would like by carefully choosing the representation. However, if we are not careful about choosing the representation, we might get a less desirable behavior. The severity of this problem is limited, because the user could always provide additional controls if they really cared what happened.

3.4.1 The Metric

In order to spare the user the increased effort of more completely specifying their intent, and to give programmers more freedom to select representations that are convenient, we must use a different objective function. Rather than measuring change in values of the parameters, we could measure change in something that did not depend as closely on the representation. We have used the functions that compute objects' attributes to serve as controls that are independent of representation. Similarly, we prefer to define an object's metric in terms of its attribute functions as well. We select a subset of the attributes to define the metric. Just as we denote the subset of the attributes that serve as the controls as $\mathbf{f}$, we will denote the function used to define the metric by $\mathbf{g}$ which is also a function of $\mathbf{q}$. We denote the Jacobian $\partial \mathbf{g}/\partial \mathbf{q}$ by $\mathbf{G}$.

The optimization objective will be to minimize the magnitude of the change in the attributes. This rate of change is

$$\dot{\mathbf{g}} = \mathbf{G}\dot{\mathbf{q}}. \tag{3.15}$$

Since we are searching for the minimum, we can minimize $1/2$ the sum of squares of

$\mathbf{g}$, rather than the magnitude. Since we might wish to emphasize some of the attributes more than others, we also introduce a scaling factor for each to make the objective a weighted sum of squares. Writing the scaling factors as the diagonal elements of a matrix for notational convenience, the objective function is

$$E = \frac{1}{2}\dot{\mathbf{q}}^{\mathbf{T}}\mathbf{G}^{\mathbf{T}}\mathbf{S}\mathbf{G}\dot{\mathbf{q}}, \tag{3.16}$$

We will call the matrix that defines this quadratic ($\mathbf{G^TSG}$) term the *metric* because it defines a way to measure $\dot{\mathbf{q}}$.

The simple objective function of Section 3.2 used the parameters of the objects as the attributes that defined the metric. Since $\mathbf{g} = \mathbf{q}$, $\mathbf{G} = \mathbf{I}$ and $\mathbf{M}$ also is the identity matrix. Viewed this way, the advantage of using a correct metric can be seen. The identity metric defines the behavior of the object in terms of its parameters, rather than in terms of something that is potentially meaningful to the user.

A Particularly Useful Metric

The metric provides a method for an interface designer to define a default behavior for an object. In effect, it allows for hand-tuning the behavior that a user sees when the object is manipulated. However, this leaves the problem that the interface designer must hand-tune the behavior in order to hide effects of the parameterization. Often, this is not an issue, as the parameters provide a reasonable default behavior, or, if a specific behavior is required, it can serve to describe a metric. However, some applications demand an automatic method for determining a metric that provides a consistent, parameterization independent, feel for a variety of objects. Such a method is particularly useful in cases where a user may define object behavior. An example is the parametric curve manipulation of Section 8.1.1 and Section 9.4.

An analogy to physics provides an automatic, consistent metric for a broad class of objects. As first suggested by Witkin [Wit89b], we can imagine an object as a physical entity with an uniform mass distribution. In effect, we can view each pixel of the object as an atom, each with a tiny bit of mass. This mass distribution defines how the object changes as forces are applied in particular places. Inertia causes each particle to move as little as possible. The mass distribution serves as the metric does, defining the behavior of objects in response to controls. Witkin and Welch [WW90] used specification of the mass distribution to allow animators to specify the the default behaviors of simulated objects that were acted upon by point controls.

When solving the equations of motion of a physical object in generalized coordinates in order to simulate it, the mass distribution is encoded into a matrix known as the inertia tensor or mass matrix [Gol80]. This matrix is found by accounting for the effects of each particle on the objects' behavior by integrating over the mass distribution. When such a matrix is determined numerically, the integral is approximated by sampling a set of particles.

Analogously, a metric can be defined by viewing a graphical object as a collection of "particles" and minimizing the motion of these particles. In practice, the distribution is estimated by a set of points. We define the metric functions to be the positions of an evenly spaced set of points on the object. We call such a metric the *mass matrix* because of its physical analog.

The mass matrix is an important metric because it can be defined independently of the object. For any graphical object, a set of points can be evenly distributed either along its length (for a curve) or within its area (if it is solid). Section 8.1.1 will illustrate the utility of this, allowing default behavior to be automatically provided for a wide variety of objects.

3.4.2 Solving the Generalized Quadratic Objective

Using a different quadratic optimization objective requires a slightly different set of methods for solving the constrained optimization problems. In this section, we derive the method in its full generality for the case of any quadratic objective function and linear constraints.

The standard form of a vector quadratic equation is

$$E = \frac{1}{2}\mathbf{x}^\mathbf{T}\mathbf{M}\mathbf{x} + \mathbf{b}^\mathbf{T}\mathbf{x} + k, \tag{3.17}$$

where $\mathbf{x}$ is the vector parameter ($\dot{\mathbf{q}}$ for this chapter), $\mathbf{M}$ is the quadratic or matrix term, the vector $\mathbf{b}$ is the linear term, and k is a scalar constant. Since we are not interested in the value itself, but rather only the value of $\mathbf{x}$ that minimizes it, we can ignore the constant as it goes away when we take the gradient, and we can multiply the quadratic by $1/2$ as it cancels out other values later, simplifying the equations. The linear term permits measuring change from a point other than 0, and will be used in Section 3.5. The methods of Section 3.2 solve the special case of this objective with an identity matrix for $\mathbf{M}$, and 0 for the linear component $\mathbf{b}$.

The Lagrange multiplier derivation can again be applied, this time to Equation 3.17. We denote the linear constraints as $\mathbf{A}\mathbf{x} = \mathbf{a}$. We require the gradient of the objective function to be a linear combination of the of the constraints,

$$\frac{\partial E}{\partial \mathbf{x}} = \mathbf{M}\mathbf{x} + \mathbf{b}\mathbf{x} = \mathbf{A}^\mathbf{T}\boldsymbol{\lambda}. \tag{3.18}$$

Solving this for $\mathbf{x}$ and denoting the inverse of the metric $\mathbf{M}^{-1}$ by $\mathbf{W}$, gives

$$\mathbf{x} = \mathbf{W}\mathbf{A}^\mathbf{T}\boldsymbol{\lambda} - \mathbf{W}\mathbf{b}. \tag{3.19}$$

Inserting this into the constraint equation $\mathbf{A}\mathbf{x} = \mathbf{a}$ gives

$$\mathbf{A}\mathbf{W}\mathbf{A}^\mathbf{T}\boldsymbol{\lambda} = \mathbf{a} + \mathbf{A}\mathbf{W}\mathbf{b}, \tag{3.20}$$

a linear system that can be solved for $\boldsymbol{\lambda}$. Once $\boldsymbol{\lambda}$ is computed, it can be inserted into Equation 3.19 to compute $\mathbf{x}$.

The damping techniques of Section 3.2.1 are not taken into account by the generalized quadratic objective. Using a derivation similar to that of Equation 3.12, yields

$$(\mathbf{A}\mathbf{W}\mathbf{A}^{\mathbf{T}} + \mu\mathbf{I})\boldsymbol{\lambda} = \mathbf{a} + \mathbf{A}\mathbf{W}\mathbf{b}. \tag{3.21}$$

Using the notation of the rest of the chapter, $\mathbf{M}$ is the metric as defined in Section 3.4.1 and the linear constraints are given by Equation 3.4 so $\mathbf{A} = \mathbf{J}$ and $\mathbf{a} = \dot{\mathbf{p}}$.

3.4.3 An Approximation to the Metric

Using the metric to define the default behavior of an object has several advantages. It permits separation of manipulation and representation, and provides an abstraction for defining the feel of an object. However, it has a significant cost: we must find the metric and invert it. This is problematic because the metric is large. Inverting a matrix this large would be prohibitive. One advantage to using the identity matrix as the metric is that it is trivial to invert.

What we aim for in this section is an approximation to the metric that is inexpensive to invert, yet provides some of the features of the full metric. The approximation we consider is simply using the diagonal elements of the metric. This diagonal matrix is trivial to invert — we merely take the reciprocal of all its elements — and cheap to use in solving Equation 3.20. It still addresses some of the important issues that the full metric addresses, particularly the selection of units.

Consider again the example of dragging a line segment in Figure 3.5. Suppose its configuration is represented by the position of its center, its length and its orientation, and that the simple identity metric objective function is used. If the upper left corner of the segment is move a quarter of an inch to the left, the line segment might have its center move, scale and rotate, or some combination of the two. Suppose that the position of the center of the line segment was represented in micrometers from the corner of the page, the orientation represented as radians from horizontal, and the length in inches. To achieve the movement of the upper left point by simply moving the center would require the parameters to change very quickly as there are many micrometers to be covered, while achieving the movement by scaling and rotating would require considerably smaller changes in the parameters. Because the simple optimization objective minimizes change in the parameters, the latter would be chosen. If the position of the center were measured in miles instead, a very tiny change in the position of the center would create the needed motion, so this would be selected by the optimization criteria.

In the example, the simple selection of units with which to represent the position of the center of the line segment determined the dragging behavior. The problem is

parameters having different units. One way around this is to define an objective function which minimized the amount of change in the parameters after converting them to some standardized units. Suppose we knew the conversion factors between the units of the parameters and the standard units. We would have a scaling factor for each parameter. For notational convenience, we can write the scaling factors as the diagonal elements of a diagonal matrix $\mathbf{S}$, so the component-wise scaling of parameters would simply be the multiplication $\mathbf{Sq}$.

Rather than simply minimizing the magnitude of $\dot{\mathbf{q}}$, we would instead minimize the time derivative of the scaled parameters, $\mathbf{S}\dot{\mathbf{q}}$, giving

$$E = \frac{1}{2}(\mathbf{S}\dot{\mathbf{q}} \cdot \mathbf{S}\dot{\mathbf{q}}), \tag{3.22}$$

or, to use the generalized notation of Equation 3.17

$$\mathbf{M} = \mathbf{S}^{\mathbf{T}}\mathbf{S}. \tag{3.23}$$

We see that we have a diagonal metric.

The problem is to determine $\mathbf{S}$ to convert the parameters to the standard units. One way to define standard units would be to require that equal changes in each variable should affect the attributes the same amount. As for the metric, we pick a subset of the attributes to define the objective function, and denote the function that computes these attributes by $\mathbf{g}$. However, we are only interested in the derivatives with respect to a single variable at a time. That is we want to measure the change in all of the attributes of $\mathbf{g}$ with respect to each variable independently. For a particular variable, the scaling factor is the magnitude of the derivatives of each element of $\mathbf{g}$ with respect to the variable, that is,

$$\mathbf{S}_{ii} = \sqrt{\frac{\mathbf{g}}{\mathbf{q}_i} \cdot \frac{\mathbf{g}}{\mathbf{q}_i}}. \tag{3.24}$$

The diagonal terms in Equation 3.23 are the scaling factors squared,

$$\mathbf{M}_{ii} = \frac{\mathbf{g}}{\mathbf{q}_i} \cdot \frac{\mathbf{g}}{\mathbf{q}_i}. \tag{3.25}$$

These are exactly the diagonal terms of the metric in Equation 3.16.

The diagonal metric cannot take into account interactions between variables. While it can remove differences in units between similar terms, it cannot make two different representations seem alike. In the line segment example, it eliminates the effects of the choice of units, however it does not remove the effect of a completely different parameterization. It could not, for example, express an objective function that minimized the motion of the endpoints. Therefore, no matter what diagonal metric are chosen, a line segment parameterized by position of center, length and orientation will feel different than a line segment parameterized by the positions of its endpoints.

3.5 Soft Controls

The generalized objective functions of the last section allow the creation of objective functions that can be used to control the object. To this point, we have discussed default behaviors for objects that causes objects to minimizes their motion if controls are causing them to change, they do not change. In this section we consider an alternative: having objects change by default unless a control specifies otherwise. The non-zero defaults lead to linear terms in the general quadratic optimization objective function of Equation 3.17. Using this term will enable soft controls: controls which are overridden by the regular controls. These are important because they allow us to create behaviors such as dragging subject to constraints, where an object is manipulated by the user but constraints will not be violated. While these techniques do not provide general constraint hierarchies as described by Borning et al.[BFBW92], the dragging subject to constraints that they can provide is useful in many graphical applications. The use of optimization objective terms to provide user control was pioneered in the vision research of Kass et al.[KWT88].

Suppose that we had some desired default value for $\dot{\mathbf{q}}$, denoted $\dot{\mathbf{q}}_\mathbf{0}$. Rather than simply minimize the magnitude of $\dot{\mathbf{q}}$, we would instead minimize its difference from the default value, so

$$E = \frac{1}{2}(\dot{\mathbf{q}} - \dot{\mathbf{q}}_\mathbf{0}) \cdot (\dot{\mathbf{q}} - \dot{\mathbf{q}}_\mathbf{0}). \tag{3.26}$$

In terms of the generalized objective function of Equation 3.17, the coefficients of the linear term $\mathbf{b}$ in this case is $\dot{\mathbf{q}}_\mathbf{0}$. Similar metrics can be worked out to include a metric as well. While the generalized solution of Section 3.4.2 can be applied, we review the derivation for this important special case here as it provides insight.

To provide intuition for how this works, consider again the simple point example from Section 4.3. Notice that the control specifies the behavior only in the direction of its gradient, so the optimization objective is free to do whatever it wants in an orthogonal direction. Suppose that we have specified $\dot{p}$ to be 0. This restricts $\dot{\mathbf{q}}$ to lie along the line perpendicular to the gradient, as shown in Figure 3.6. To find $\dot{\mathbf{q}}$ closest to $\dot{\mathbf{q}}_\mathbf{0}$, we must project $\dot{\mathbf{q}}_\mathbf{0}$ onto this line. We do this by adding in a component of $\dot{\mathbf{q}}$ which cancels out the disallowed portion. This *constraint component* must be a multiple of the gradient. This multiple is the Lagrange multiplier.

We compute $\dot{\mathbf{q}}$ as the sum of two components, its default value $\dot{\mathbf{q}}_\mathbf{0}$ and the contributions of the controls, $\dot{\mathbf{q}}_\mathbf{c}$, so

$$\dot{\mathbf{q}} = \dot{\mathbf{q}}_\mathbf{0} + \dot{\mathbf{q}}_\mathbf{c}. \tag{3.27}$$

Since $\dot{\mathbf{q}}$ must satisfy the controls, we substitute this into Equation 3.4, to yield

$$\dot{\mathbf{p}} = \mathbf{J}(\dot{\mathbf{q}}_\mathbf{0} + \dot{\mathbf{q}}_\mathbf{c}). \tag{3.28}$$

Since the contribution of the constraints is a linear combination of the control gradients,

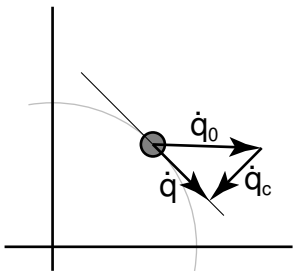

Figure 3.6: A hard controls constrains the distance from the point to the origin. Any movement of the point must be orthogonal to the gradient of this control. When a default velocity ($\dot{\mathbf{q}}_\mathbf{0}$) is given for $\dot{\mathbf{q}}$, it must be projected into the space that meets this restriction. To achieve this, a component is added to $\dot{\mathbf{q}}_0$ that projects it onto the space where $\dot{\mathbf{p}} = 0$.

we define the Lagrange multipliers as

$$\dot{\mathbf{q}}_\mathbf{c} = \mathbf{J}^T \boldsymbol{\lambda}. \tag{3.29}$$

Which, with a little rearrangement yields

$$\dot{\mathbf{p}} - \mathbf{J}\dot{\mathbf{q}}_\mathbf{0} = \mathbf{J}\mathbf{J}^T \boldsymbol{\lambda}, \tag{3.30}$$

a linear system, which like Equation 3.9 can be solved for $\boldsymbol{\lambda}$, which in turn determines $\dot{\mathbf{q}}_\mathbf{c}$ by Equation 3.29, from which $\dot{\mathbf{q}}$ can be computed by Equation 3.27. Notice that when $\dot{\mathbf{q}}_\mathbf{0} = 0$, the method of this section is exactly the same as that of Section 4.3. The damping techniques of Section 3.2.1 also apply.

3.5.1 Determining the Values for Soft Controls

We now must figure out how to obtain $\dot{\mathbf{q}}_\mathbf{0}$. Our goal is to provide soft controls that work as the hard controls do, except that the hard controls are given precedence over them. Soft controls are defined as $\mathbf{p_s} = \mathbf{f_s}(\mathbf{q})$, but like the hard controls, would be specified by their derivatives, $\dot{\mathbf{p}}_\mathbf{s}$.

If the soft controls do not conflict with the hard controls, they can simply be treated as hard controls. The more interesting cases, however, are when the hard controls limit the soft controls. The ultimate goal is to have have soft controls work exactly as hard controls do, except in the cases where there are hard controls that take precedence over the soft controls.

We would like to satisfy the soft controls as closely as possible subject to the restriction that the hard controls are specified exactly. We can define the objective function to minimize the squared error of the soft controls meeting their desired values

$$\text{minimize } E = \frac{1}{2}(\mathbf{J_s}\dot{\mathbf{q}} - \dot{\mathbf{p}}_\mathbf{s}) \cdot (\mathbf{J_s}\dot{\mathbf{q}} - \dot{\mathbf{p}}_\mathbf{s}) \text{ subject to } \dot{\mathbf{p}} = \mathbf{J}\dot{\mathbf{q}}. \tag{3.31}$$

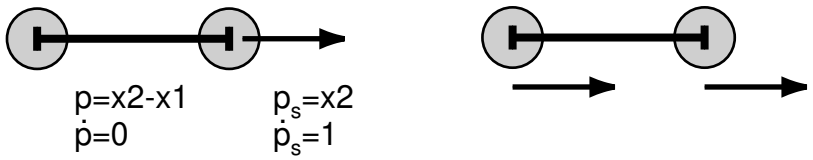

Figure 3.7: Two points are connected by a hard control constraining their distance. The right point is also pulled by a soft control. If the soft controls are computed independently, a non-optimal solution may be found, as shown on the left. The soft control may specify that the right point should move, but part of this motion might be removed by the constraints. As shown on the right, there is a solution that both satisfied the constraints and meets the desired values for the soft controls.

Such an objective function is the similar to the definition of the metric in Section 3.4.1, except that rather than minimizing the magnitude squared of the change of the derivatives of the functions value, we minimize the magnitude squared of the difference between the derivative's value and a default value. Generalized sets of soft controls (e.g. $\mathbf{J_s}$) will lead to $\mathbf{M}$ terms in Equation 3.17. The problems in using general metrics also apply to soft controls: A sufficient number of soft controls must be specified to uniquely determine $\dot{\mathbf{q}}$ in all cases, even when there are no hard controls. If an insufficient number of soft controls are specified, $\mathbf{M}$ will be singular. Problems with ill-conditioning and efficiency in inverting $\mathbf{M}$ make this soft control scheme impractical. In this section, we concentrate on methods for simpler achieving soft controls by computing values for $\dot{\mathbf{q}}_0$.

Two-Pass Solver

One way to find $\dot{\mathbf{q}}_0$ is to ignore the hard controls, and simply use the method used for hard controls for the soft controls. Using this approach, two linear systems are solved: first, a linear system is solved to compute the Lagrange multipliers that will determine $\dot{\mathbf{q}}_0$, then a linear system is solved to project $\dot{\mathbf{q}}_0$ into the subspace allowed by the hard constraints.

The method of computing the soft controls independently has a serious drawback: it does not achieve the desired solution. Consider a case where two points are connected with a hard control constraining their distance, and a soft control pulling one of the points to the right, depicted in Figure 3.7. The soft control alone would move one of the points, violating the hard control. When this is projected into the legal subspace, part of the motion world be cancelled out. However, if all the controls were treated equally, the other point could be moved to satisfy the hard control.

An alternative is to account for the hard controls in computing $\dot{\mathbf{q}}_0$. We compute $\dot{\mathbf{q}}_0$ using the methods we would use for the hard controls, except that we include both the hard controls and the soft controls in the computation. Damping must be used in case

the controls conflict. We then solve an optimization problem again using the result of the first solution as $\dot{\mathbf{q}}_0$, and just using the hard controls as constraints. I first used this technique in the *Briar* drawing program described in Section 9.1, and therefore call it the *Briar-style solver.*

In cases where all the controls, both soft and hard, are consistent, the *Briar-style* solver has the nice property that both hard and soft controls behave the same. It is also the case that the effort in solving the linear system twice can be avoided by first checking to see if $\mathbf{J}\dot{\mathbf{q}}_0 = \dot{\mathbf{p}}$, in which case $\dot{\mathbf{q}}_c$ will trivially be 0. When using an iterative linear system solver, as described in Section 4.3.1, this check happens automatically.

However, if we knew that the hard and soft controls did not conflict, then there would be no need to have soft controls. In the cases where there are conflicts, the *Briar-style* solver has a few drawbacks. Most obvious is that it requires solving the linear system twice, which can be expensive. Also, since solving the larger system will have conflicting constraints, damping must be used. The solver does not actually solve Equation 3.31, but instead minimizes the difference between the damped result, which already partially accounts for the hard constraints.

Spring Controls

An alternate method to compute $\dot{\mathbf{q}}_0$ is to use gradient descent to drive the soft controls to their desired values. We compute $\dot{\mathbf{q}}_0$ to have the direction of the gradient of the soft control functions, and a magnitude proportional to how far from the target it is,

$$\dot{\mathbf{q}}_0 = k\mathbf{J}_\mathbf{s}(\mathbf{p}_\mathbf{s} - \mathbf{f}_\mathbf{s}(\mathbf{q})), \tag{3.32}$$

where k is a scaling constant. This causes the controls to be pulled towards their desired values with a decaying attraction: as the control nears its desired value, the rate at which it is being pulled is decreased. In the physical analogy of Section 1.2.2, this attraction is a spring. Equation 3.32 is the generalized force version of Hooke's law.

I will call these spring-like controls *spring controls* or simply *springs.* Their method can be viewed as a cheap way to estimate the Lagrange Multiplier, an attempt to use gradient descent to achieve desired values for the soft controls, or as generalized springs, if we view the optimization as a physical simulation. Despite the fact that they are a little harder to justify, they do work very well.

3.6 An Alternate Technique

The methods presented in the last sections described Lagrange multiplier techniques for solving constrained optimization problems with linear constraints and a quadratic objective function. The methods build a linear system and solve for an intermediate result, the Lagrange multipliers. The methods have the advantage that they permit the use of any quadratic objective function.

Often, we do not exploit generality of the Lagrange multiplier formulation. For example, the simple objective functions of Equation 3.6 or Equation 3.26 may be sufficient. In such cases, alternate, special purpose solution methods are sufficient.

The objective function that simply minimizes the magnitude of $\dot{\mathbf{q}}$ gives an important special case called a linear least squares problem. This is a very standard problem in numerical analysis. Example solving methods include singular value decomposition (SVD), QR factorization, and pseudo-inverses. These methods, and many others, are discussed in [GL89]. Unfortunately, these methods are almost all extremely expensive to compute, as they are unable to exploit properties of the problems such as sparsity that we will use in the next chapter to speed performance.

Iterative solvers may also be used to solve the linear least squares problem. In particular, conjugate-gradient solvers, discussed in Section 4.3.1, are relevant to implementing the differential approach. Variants of conjugate-gradient find a solution to the linear system, but have the property that the solution they provide to a linear system is solution closest to the starting point. Therefore, if the solver is begun with a zero starting point, the solution with least magnitude is found. The conjugate-gradient linear system solver in the Numerical Recipes text [PFTV86] is such a solver. To implement soft controls using the conjugate-gradient, Equation 3.28 is solved for $\dot{\mathbf{q}}_\mathbf{c}$ using the conjugate-gradient solver.

Using a least squares solver to implement the differential approach effectively solves Equation 3.4 directly, without first computing the Lagrange multipliers. For instance, implementing the differential approach by using a conjugate gradient algorithm works very well. The solver given in the text of Press et al.[PFTV86] permits the two most often used objective functions, and handles over-determined cases by providing a minimum norm residual solution to the linear system. Using the algorithm is a very practical way to implement the differential approach. It performs extremely well in practice. This method served as the backbone of my early implementations, and is available by a run-time switch even in my most current versions.

The obvious question is why bother developing a more complex technique when the simpler approach works so well. The three main reasons for using the Lagrange multiplier techniques in this chapter over the simpler "direct" approaches such as using conjugate-gradient are: the intermediate result of the Lagrange multiplier techniques (the Lagrange multipliers) will be useful in certain interaction techniques such as the active set methods of Section 6.4; they place few restrictions on the linear system solver that is used, so that fast algorithms can be found; and, they extend to other quadratic objective functions. However, in cases where these advantages are not required, the simpler approach is worth considering. The approach handles the two most often used objective functions, those of Equation 3.6 and Equation 3.26, so it is often sufficient.

3.7 A Concrete Example

To review the basic techniques of this chapter, we now consider a complete example in detail. Our object will be a fixed length line segment with a unit radius, represented by the position of its center and its orientation. The control we will create is the position of its endpoint. The state variables are $\mathbf{q} = \{\mathbf{q}_{cx}, \mathbf{q}_{cy}, \mathbf{q}_{theta}\}$, and the controls are $\mathbf{p} = \{\mathbf{p}_x, \mathbf{p}_y\}$.

The control functions are:

$$\begin{aligned} \mathbf{p}_x &= f_x(\mathbf{q}) = \mathbf{q}_{cx} + \cos \mathbf{q}_\theta \\ \mathbf{p}_y &= f_y(\mathbf{q}) = \mathbf{q}_{cy} + \sin \mathbf{q}_\theta. \end{aligned} \tag{3.33}$$

The core of the implementation will be the differential optimization that will compute a value for $\dot{\mathbf{q}}$ given a value for $\mathbf{q}$ and $\dot{\mathbf{p}}$. This routine must first compute the Jacobian of the controls, **J**, as a function of $\mathbf{q}$:

$$\mathbf{J} = \begin{bmatrix} 1 & 0 & -\sin \mathbf{q}_\theta \\ 0 & 1 & +\cos \mathbf{q}_\theta \end{bmatrix}. \tag{3.34}$$

It then can compute values for the Lagrange multipliers by solving the linear system

$$\dot{\mathbf{p}} = \mathbf{J}\mathbf{J}^{\mathbf{T}}\boldsymbol{\lambda} \tag{3.35}$$

for $\boldsymbol{\lambda}$, and then computing $\dot{\mathbf{q}}$ as

$$\dot{\mathbf{q}} = \mathbf{J}^{\mathbf{T}}\boldsymbol{\lambda}. \tag{3.36}$$

Suppose at the current time, the line was at a 45 degree angle with its center at the origin ($\mathbf{q} = [0, 0, \pi/4]$), and that the control specifies the endpoint to move right with unit velocity ($\dot{\mathbf{p}} = [1, 0]$). The Jacobian of the controls is then

$$\mathbf{J} = \begin{bmatrix} 1 & 0 & -.707 \\ 0 & 1 & .707 \end{bmatrix}. \tag{3.37}$$

To compute the $\dot{\mathbf{q}}$, we must first solve the linear system

$$\begin{bmatrix} 1.5 & -0.5 \\ -0.5 & 1.5 \end{bmatrix} \boldsymbol{\lambda} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \tag{3.38}$$

then use Equation 3.36 to determine $\dot{\mathbf{q}}$. At this particular instant, $\boldsymbol{\lambda} = [.75\ .25]$, so $\dot{\mathbf{q}} = [.75\ .25\ -.35]$. If we were to use an Euler step with step size .1, the configuration at the end of the step would be $\mathbf{q} = [.075\ .025\ .75]$.

The basic control for interaction will be to repeatedly take ODE solver steps, interleaving redraw between the steps to give the illusion of motion. This solver will call the differential optimization routine, possibly several times per iteration. The ODE solver

will also have to provide values for $\dot{\mathbf{p}}$ to the optimization routine. These values are what accounts for the user's motions; for example, they might be tied to the input device. Methods for determining desired velocities will be discussed in Section 6.3, however, one simple way of getting $\dot{\mathbf{p}}$ from the input device is to use decaying attraction: we compute the vector from the position of the control to the pointer, and use a multiple of this for $\dot{\mathbf{p}}$.

In this example, we are already using multiple controls, one for each axis. Even more controls could be added. For example, suppose we wanted to add two more controls that position the other end of the line segment. These controls are computed by

$$\begin{aligned} \mathbf{p}_{x2} &= f_x(\mathbf{q}) = \mathbf{q}_{cx} - \cos \mathbf{q}_\theta \\ \mathbf{p}_{y2} &= f_y(\mathbf{q}) = \mathbf{q}_{cy} - \sin \mathbf{q}_\theta . \end{aligned} \tag{3.39}$$

The Jacobian would now be a 4 by 3 matrix.

Since we most likely will not have two mice, rather than permitting the user to control the position of the second point, we might want to constrain it to remain at the origin. This would require specifying $\mathbf{p}_{x2}$ and $\mathbf{p}_{y2}$ to have values that caused the point to move towards the origin, e.g. to be a negatively scaled multiple of the position of the point.

Clearly, these controls will conflict: the mouse might attempt to pull the other endpoint away from the origin. This will cause the matrix $\mathbf{JJ^T}$ to be singular. In order to solve the linear system, we might add damping by adding a small amount to the diagonal elements of the 4 by 4 matrix.

We might instead wish to drag the endpoint subject to the constraint that the other endpoint remains at the origin, that is, the mouse should not be able to rip the other endpoint from its resting point, but other than that, should be able to drag its endpoint as well as possible. To do this, we use soft controls.

To compute the optimization with the soft controls, we first must compute a value for $\dot{\mathbf{q}}_\mathbf{0}$ by computing the Jacobian of the soft controls with Equation 3.40, and multiply this by the desired value of the soft controls. The regular controls are now just the opposite endpoint, so the Jacobian is simply the 2x3 matrix of their derivatives. To compute $\dot{\mathbf{q}}$ we first compute the Lagrange multipliers by solving the linear system

$$\dot{\mathbf{p}} - \mathbf{J}\dot{\mathbf{q}}_\mathbf{0} = \mathbf{J}\mathbf{J}^T\boldsymbol{\lambda}. \tag{3.40}$$

We use that to compute $\dot{\mathbf{q}}$ by

$$\dot{\mathbf{q}} = \dot{\mathbf{q}}_0 + \mathbf{J^T}\boldsymbol{\lambda}. \tag{3.41}$$

3.8 Summary

In this section we review the techniques presented in this chapter for solving the differential optimization problem, summarize the procedure for implementing it, and de-

n	number of controls
m	number of state variables
$\mathbf{q}$	state vector (m-vector)
$\mathbf{p}$	values of the controls (n-vector)
$\mathbf{f}$	function to compute the controls, $\mathbf{p} = \mathbf{f}(\mathbf{q})$
$\dot{\mathbf{q}}$	time derivative of $\mathbf{q}$ (m-vector)
$\dot{\mathbf{p}}$	time derivative of $\mathbf{p}$ (n-vector)
$\mathbf{J}$	Jacobian of $\mathbf{f}$ ($n \times m$ matrix), $\mathbf{J} = \partial \mathbf{f} / \partial \mathbf{q}$
E	optimization objective
$\boldsymbol{\lambda}$	Lagrange multipliers (n-vector)
μ	damping factor (scalar or n-vector)
$\mathbf{q_0}$	default value for $\mathbf{q}$
$\mathbf{p_s}$	soft controls
$\mathbf{f_s}$	function computing soft controls
$\mathbf{J_s}$	Jacobian $\partial \mathbf{f_s} / \partial \mathbf{q}$
$\mathbf{M}$	metric ($m \times m$ matrix)
$\mathbf{W}$	inverse metric $\mathbf{M^{-1}}$ ($m \times m$ matrix)
$\mathbf{g}$	functions whose change is minimized to define $\mathbf{M}$
$\mathbf{G}$	Jacobian $\partial \mathbf{g} / \partial \mathbf{q}$

Table 3.1: Symbols defined in this chapter, and used throughout the thesis.

scribe the caveats as to solving the more general, numerical optimization problem. The symbols used throughout this chapter, and for the rest of the thesis are reviewed in Table 3.1.

The differential optimization takes the current value of the state $\mathbf{q}$, and the functions that define the controls and objectives as givens. From these givens, the current values of the controls and the Jacobians of the controls and objective metric functions can be computed, so they too are considered givens. The procedure is as follows:

1. Compute the $\dot{\mathbf{q}}_0$ value of the force controls, if any, using the damped spring formula of Equation 3.32, or some other methods.

2. Find the metric, $\mathbf{M}$, and its inverse. Often, the identity matrix is used instead. Computing a metric involves computing the Jacobian of $\mathbf{g}$.

3. Compute the Jacobian of $\mathbf{f}$, $\mathbf{J}$.

4. Compute the Lagrange multipliers, using some variant of Equation 3.20. Most likely, some damping will be used on some of the controls.

5. Compute $\dot{\mathbf{q}}$, for example by Equation 3.8.

6. If this computation was to compute both the hard and soft controls in a Briar-style solver (as in Section 3.5.1), remove the soft controls, set $\dot{\mathbf{q}}_0 = \dot{\mathbf{q}}$, and return to step 3. For the second time through, less damping might be used.

The differential optimization solves for $\dot{\mathbf{q}}$, given $\dot{\mathbf{p}}$ and $\mathbf{q}$. It makes use of the control function ($\mathbf{f}$) and its Jacobian (which is a function of $\mathbf{q}$). If a metric is to be defined, a set of functions ($\mathbf{g}$) and its Jacobian will be needed as well. The process has a single tunable parameter, which is the amount of damping (μ). μ might be a vector if damping values are to be provided for each control.

To emphasize, being able to solve the differential optimization problem

$$\dot{\mathbf{q}} = \mathcal{O}_{(t,\mathbf{f})}(\mathbf{q}, \dot{\mathbf{p}}). \tag{3.42}$$

is not the same as being able to solve the general control equation

$$\mathbf{q} = \mathbf{f}^{-1}(\mathbf{p}). \tag{3.43}$$

In fact, being able to solve Equation 3.42 does not necessarily allow solving Equation 3.43. There are two main reasons for this: we need to know what $\dot{\mathbf{p}}$ will achieve the changes necessary to get the desired values of $\mathbf{p}$; and, once we have $\dot{\mathbf{q}}$, we don't necessarily know what $\mathbf{q}$ is at some future time.

Given a desired value for $\mathbf{p}$, the most obvious way to proceed differentially is to use a $\dot{\mathbf{p}}$ that changes the value as needed, e.g. if the value of a control is too high, make it decrease. However, heading straight for the goal is only a heuristic that can often fail, for example, if there is a local minimum. Also, there is no certainty that there is velocity $\dot{\mathbf{p}}$ that will achieve the desired controls, either if there is no way to achieve the desired controls at all, or if there is simply no continuous path through state space.

Even if a direction for $\dot{\mathbf{p}}$ is known, and the corresponding $\dot{\mathbf{q}}$ that achieves it can be found, there is no guarantee that the correct $\mathbf{q}$ can be found. Finding values of $\mathbf{q}$ requires solving the ordinary differential equation of Equation 3.42 for $\mathbf{q}$.

The secret to walking on water is knowing where the rocks are.

— Herb Cohen
Vail Symposium 14 poster

Chapter 4

Efficient Solution Techniques

In the previous chapter, we introduced methods for implementing the differential approach. In this chapter, we now consider how to solve the differential optimization problems efficiently.

4.1 The Demands of Interactive Systems

Interactive systems place a different set of demands on numerical techniques than more traditional, batch computation applications might.

One unique demand of the numerical problems in interactive systems is that they are dynamic. Because the equations are created in response to the users' actions, they are not known when the system is created. More significantly, the set of equations to be solved is continually changing in response to the user. This dynamic nature of the numerical problem means that we must be able to define equations at run time, which will be addressed in Chapter 5. Solution methods that rely on extensive pre-analysis are to be avoided as the problem being solved may change before the cost of the analysis can be amortized.

A common practice in numerical computations is to adapt the solution methods on a per problem basis. This ranges from experimenting with different algorithms to see which best solves a given problem, to adjusting parameters to make solvers converge. Such per problem tweaking is unacceptable in the setting of an interactive system. Not only is the problem continually changing, but we would like to insulate the user from the mathematics. We do not want the user to have to learn about constrained optimization just to draw a picture.

Speed is an important consideration for numerical routines. For the differential approach, it is critical. If the computations are not fast enough, the system will not be able to provide the smooth motion which is demanded by direct manipulation. We also must be concerned with scalability, that is how the methods will perform as the problems grow larger.

Accuracy, typically an important concern in numerical analysis, is less critical to interactive systems. This is important since there is often a tradeoff between the time a computation takes and how accurate it is. The accuracy required is typically limited by factors such as device resolution. In the cases where users demand sub-pixel accuracy, for example when designing an object that is to be manufactured, the accuracy demands are typically known. Even in applications where high accuracy is required, fast, inaccurate methods are useful if these results can be refined.

While accuracy is not essential for interactive applications, stability is. Numerical instabilities can cause such undesirable effects as objects wobbling or flying off the screen. Stability is, therefore, an important concern for interactive systems.

For the purpose of this thesis, there is an additional goal for the numerical routines. We would prefer techniques that are simple and widely available. For several of the numerical problems we face, for example solving linear systems and ordinary differential equations, a vast array of sophisticated software packages are available, both from public sources and commercial vendors. Development of such algorithms is beyond the scope of this thesis. Similarly, relying on a particular software package would make the approach harder to reproduce and port.[1]

4.1.1 Basic Methods for Achieving Performance

There are a few general strategies for improving the performance of the computations. These will be applied in various ways throughout this chapter.

Trade Accuracy for Performance — As discussed earlier, in an interactive system, we are often willing to trade accuracy for performance. Techniques for doing this will be discussed in Section 4.5. It is generally *not* acceptable to trade stability for performance.

Trade Convergence for Iteration Rate — As a control moves towards a goal value, it is more important that it moves with smooth motion than that it gets to its goal in a minimum amount of time.

Exploit Sparsity — The matrices involved in the differential optimization problems are filled mainly with zeros. It is crucial to exploit this fact, both for speed and memory usage.

Reduce the Problem Size — In the next section, we will see that the computational complexity of the differential methods is linear in the number of variables and quadratic in the number of constraints. Therefore, to handle larger problems, the size of the numerical problems actually solved must be reduced, while giving the user the illusion that the system is solving the larger problem.

[1] The work of Mark Surles[Sur92a, Sur92b] has such a problem. Anyone wishing to reproduce the results must purchase an expensive sparse matrix package on which the work relies.

Reuse Previous Results — Many intermediate results are used by several later computations. Caching such intermediate results can avoid redundant computation. Caching will be discussed in the next chapter.

One other important method for speeding numerical computations is to exploit special cases. For example, extremely efficient algorithms exist for solving n-body dynamics problems, finite elements, diagonal matrices, and the kinematics of articulated chains. However, the goals of the differential approach demand general purpose solutions. We therefore focus on general purpose methods for enhancing performance.

4.2 Scalability of the Differential Approach

With the differential approach, it is important that the redraw steps happen fast enough to give the illusion of continuous motion. As the number of objects and controls grows, so does the time required to make a step. In this section, we consider how the performance of solving in the differential approach scales as problems get larger, and identify the bottlenecks in performance. We are primarily concerned with parts of the computation that scale worse than linearly.

The problem size of the differential approach can grow in two ways: the number of controls (n), and the number of variables or objects (m). For complexity analysis purposes we consider variables and objects equivalent because each object will have a small constant number of variables. For all interesting cases, $m > n$, otherwise there will certainly be redundant or conflicting controls.

We consider only the complexity of solving the differential optimization problems. Other parts of the system might scale badly: for example, an input technique might need to examine all pairs of objects (requiring $O(m^2)$ time), or a rendering computation might require solving for interactions among all objects. However, such issues would need to be addressed in non-differential approaches as well. ODE solving, the other part of the differential approach's computation, will require a small constant number of calls to the differential optimization solver for the kinds of ODE solvers we might consider.

With arbitrary controls, the computation costs of the differential approach are almost unbounded. For example, we might have a control on the average center of each combination of 4 objects, requiring a combinatorial explosion just to enumerate the terms in the expression. However, for analysis we make some assumptions that are rarely violated in practice:

1. Objects are independent, therefore, the addition of another object does not change the number of variables an object has, or the amount of time that it takes to compute attributes.

2. Controls are independent, therefore, the addition of another control does not change the number of variables a control depends on or adversely change the amount of time to compute a control's value. Some optimizations, such as notably common subexpression sharing, may speed evaluations.

3. Controls depend on a fixed number of variables, independently of the total number of variables or controls in the system. This restriction eliminates controls on the aggregate of all objects, for example the center of mass of all objects in the world.

From these three assumptions, it follows that n and m are independent. It also follows that the time to compute the values for the controls is $O(n)$, because computing each of n controls cannot depend on either n or m. By a similar argument, the Jacobian of the controls can also be computed in $O(n)$ time.

The fact that the Jacobian of the controls $(\mathbf{J})$ can be computed in $O(n)$ time is significant, and non-obvious. $\mathbf{J}$ is an $n \times m$ matrix, so it would take $O(nm)$ time just to fill the matrix with 0s. The key observation is that we do not have to store all the values in the matrix because many of them will be zero, that is, the matrix is *sparse.* Each row depends only on the variables that its control depends on, which is independent of n or m. Therefore, the entire matrix will contain only $O(n)$ entries. By exploiting sparsity, this can be stored and accessed in $O(n)$ time. Exploiting sparsity is an important tool in implementing the differential approach.

Computing the differential optimization requires solving a linear system with the $n \times n$ matrix $\mathbf{JJ^T}$. This matrix can be built in $O(n^2)$ time because each element is computed by the dot product of two constant length vectors. Solving the linear system, with a standard method such as Gaussian Elimination, would be an $O(n^3)$ process. This unacceptable asymptotic performance can be improved by exploiting sparsity.

For certain classes of sparse matrices, linear systems can be solved in much less than $O(n^3)$ time. For example, if the matrix has constant bandwidth, solving time is $O(n)$. For certain configurations of controls, the matrices will have this structure. Surles[Sur92a] describes why important problems in molecular biology and other domains have this structure, and describes techniques for solving such constraint systems using methods very similar to the differential approach [Sur92b]. Unfortunately, if we permit constraints among arbitrary objects, as we must for the general differential approach, we do not know the structure of the matrix a priori. In fact, there is no guarantee that the matrix $\mathbf{JJ^T}$ will be sparse.

The way to exploit sparsity without making restrictions on the way objects can be connected is to avoid constructing $\mathbf{JJ^T}$. Many types of iterative linear system solvers, such as the Conjugate-Gradient techniques discussed later, access the matrices only by multiplying them by a vector. Using the associativity of matrices, the multiplication $\mathbf{JJ^Tx}$ can be achieved by doing two matrix by vector multiplies. Each of these would take $O(n)$ time because that is the number of entries are in the matrix $\mathbf{J}$. Technically,

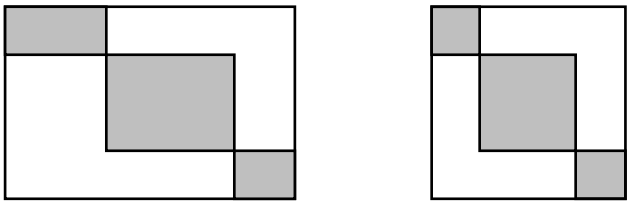

Figure 4.1: Sparsity patterns depicted by filling potentially non-zero elements with grey. Left: Since each object defines a few functions to contribute to the metric, and these functions depend only on the object's variables, the Jacobian will be *block-sparse* with a rectangular region for each object. Right: When the Jacobian is multiplied by its transpose, the blocks do not interact with each other, leading to independent squares along the diagonal of the matrix. This form of sparsity is called *block-diagonal.*

these multiplies will take $O(m)$ time because the intermediate result $\mathbf{J^T x}$ is a vector of length m, and $m > n$. However, if $m \gg n$, the vector will be sparse, so sparse vector techniques reduce things back to $O(n)$. Using a solver that requires only $O(n)$ of these matrix vector multiplies means that the linear system can be solved in $O(n^2)$ time. Empirical results using a Conjugate-Gradient solver confirming this are discussed in Appendix B.

4.2.1 Complexity of the Metric

The above discussion ignored the metric. $\mathbf{M}$ is an $m \times m$ matrix, so in the general case, simply filling it or inverting it would dominate the asymptotic complexity of the differential solving. This complexity prohibits using arbitrary metrics or using the optimization objective to find the soft controls as described in Section 3.5.1. With some reasonable restrictions, metrics can be supported without adversely effecting the computational complexity. To the restrictions of the previous section, we add

4. Each object defines its metric (e.g. its contributions to $\mathbf{g}$) independently.

All of the arguments for evaluating the controls and their Jacobian apply to $\mathbf{g}$ as well. However, since each row of $\mathbf{G}$ can only depend on the variables of one object, we know that the matrix must have a block structure, depicted in Figure 4.1. When this is multiplied by its transpose to form the metric, the matrix will be block diagonal with a block for each object, and block sizes equal to the number of variables that each object has.

The block sparsity of the metric is important. It has only $O(m)$ entries, and can be inverted in $O(m)$ time because each of the blocks is independent. Using the same associativity argument as for $\mathbf{JJ^T x}$, the matrix multiplication $\mathbf{JWJ^T x}$ also will take $O(n^2)$ time. The same arguments apply for the diagonal metric.

4.3 Solving the Linear System

To compute the Lagrange multipliers, we must solve a linear system which can be written in the standard form

$$\mathbf{A}\boldsymbol{\lambda} = \mathbf{b}, \tag{4.1}$$

where $\mathbf{A}$ is a square matrix of size equal to the number of constraints, and is determined by the Jacobians of the control functions, and $\mathbf{b}$ is a vector computed from the control values. For example, in the simplest form of the differential optimization, $\mathbf{A}$ is $\mathbf{J}\mathbf{J}^T$, and $\mathbf{b}$ is $\dot{\mathbf{p}}$.

In choosing a numerical algorithm to solve the linear system, we first must consider the properties of $\mathbf{A}$. First, it will always be positive semi-definite, and in the cases with damping, positive definite. It will also always be symmetric. These properties hold because it is created by multiplying a matrix by its transpose and altering the diagonal.

The most important property of $\mathbf{A}$ with regard to solving it efficiently is that it will be sparse. Or, more precisely, it will be created by multiplying a set of sparse matrices. This is significant because in this section we will show an algorithm which does not actually ever build $\mathbf{A}$. The structure of $\mathbf{A}$ relates to the constraint problem at hand. In particular, an element of $\mathbf{A}$ will be non-zero if the two constraints (one corresponding to the row, one corresponding to the column) share a variable.

Solving a linear system is an $O(n^3)$ process in general. Exploiting the sparsity of the matrices is important to achieving better performance. Sparse matrix techniques generally fall into two categories: direct and iterative methods. Direct methods take advantage of the structure of the matrix problem to solve the linear system as quickly as possible. For matrices where the structure is unknown, the algorithms do a pre-analysis to find the structure of the matrix so it can be solved quickly. Direct methods are not well suited to the purposes of this thesis for several reasons. First, because the structure of the matrix is continually changing, the pre-analysis must be done often making its cost difficult to amortize. Secondly, direct methods' computational complexity is proportional to the bandwidth of the matrix, so it is possible that even for an extremely sparse matrix, the cost will still be $O(n^3)$. Finally, direct methods are complicated to implement.

Iterative methods solve linear systems by repeatedly performing a calculation that eventually converges on the solution. Such methods offer an opportunity to trade accuracy for performance by controlling the tolerance to which the solver is required to achieve. By setting a larger tolerance, the algorithm is permitted to stop before it achieves an exact solution. In Section 4.2, an argument was given that with an iterative solver that does only a constant number of matrix vector multiplies per iteration, $O(n^2)$ performance could be achieved for the differential approach. The particular type of algorithm suggested for use in the differential approach, Conjugate-Gradient methods, offers this performance and several other advantages.

4.3.1 Conjugate-Gradient Linear System Solving

Conjugate-Gradient is a class of iterative algorithms for solving linear systems, non-linear systems, and optimization problems. Surveys of Conjugate-Gradient methods for solving linear systems are provided in [PS82] and [She94]. [GL89] also provides a good introduction to the techniques. The actual solver I have used is adapted from the one presented in [PFTV86]. We briefly review some of the important attributes of the algorithm here.

Conjugate-Gradient algorithms operate by repeatedly refining an estimate to the solution of the system of equations. Consider the current estimate as a point in n-dimensional space. At each iteration, the algorithm chooses a direction in which to move the estimate, computes a distance to travel in this direction, and finally updates the estimate accordingly. This process is repeated until the estimate is sufficiently close to being a solution, which can be quickly checked by inserting the estimate into the equation and measuring the error.

The key piece of a Conjugate-Gradient algorithm is how it selects directions to move its estimate in. For each iteration, a direction is chosen that is conjugate (orthogonal) to the preceding directions. Since a set of mutually conjugate vectors in n-space has n elements, a conjugate gradient algorithm, under ideal situations, would require at most n iterations to get an exact solution. In practice, numerical inaccuracies may cause the solver to require more iterations on ill-conditioned problems. Because we are not as concerned with accuracy, we will settle for stopping the solver before it completely converges in ill-conditioned cases, limiting it to $O(n)$ iterations.

At each iteration, a conjugate gradient algorithm must compute a new direction, find a step length in this direction, revise the estimate, and compute the error residual. The only part of this which actually must access the matrix are the first and last step. What is most significant for our purposes here is that in those steps, the only accesses to the matrix are to multiply it by a vector.

The Conjugate-Gradient technique leads to a family of algorithms. Many of the more sophisticated algorithms, such as LSQR algorithm introduced in [PS82], provide greater precision and more tolerance of numerical errors. As discussed in the paper introducing LSQR, the more sophisticated algorithms offer advantages only when high accuracy is required and when the matrix is ill-conditioned. However, as discussed in Section 4.1, the standard tradeoffs of numerical analysis do not apply to our applications. Since getting an answer quickly is more important than obtaining a high-accuracy answer, the more traditional Conjugate-Gradient methods may be actually be more desirable for our purposes.

4.3.2 Selection of a Linear System Solving Algorithm

Solving the linear system dominates the computational complexity of the differential approach. Selecting the algorithm is, therefore, an important decision.

From my experience, a Conjugate-Gradient solver is the best candidate for use in the differential approach because:

1. It exploits sparsity irrespective of the form of the matrix to be solved, leading to $O(n^2)$ performance in typical applications that meet the assumptions of Section 4.2.

2. It is simpler to implement than other general sparse matrix solvers such as direct methods.

3. It does not need to form the actual matrix that defines the linear system, which may not be sparse. Instead, it simply uses the Jacobian matrices that make it up, avoiding multiplying the Jacobian by its transpose.

4. The only operations it requires from the Jacobian is the ability to multiply by it and its transpose by a vector, providing freedom in choosing the representation for $\mathbf{J}$.

5. The stopping criteria can be adjusted to trade accuracy for performance by accepting solutions within a larger tolerance.

Most other approaches to solving the linear system in the differential optimization problem fail to provide one of these advantages. Other solvers potentially offer other advantages, for example lower overhead, more accuracy, or better tolerance of numerical errors, however, these advantages are often outweighed by those listed for Conjugate-Gradient. For example, a Cholesky factorization, as presented in [PFTV86], is a very efficient way to solve linear systems with positive-definite symmetric matrices, just what is needed for the differential approach. Such a solver has $O(n^3)$ performance, but with a very small constant, and is numerically stable. For small problems, the small constants of the Cholesky algorithm might make it a faster method. However, even in these cases, performing the matrix multiply $\mathbf{JJ^T}$ is often expensive enough to outweigh the performance gains in the linear system solver.

Other iterative solvers may compete with Conjugate-Gradient in some applications. The performance of iterative solvers is very problem dependent. My experimentation shows Conjugate-Gradient to be vastly superior than simpler Jacobi iterative solvers. Implementing Gauss-Seidel iteration or Successive Over-Relaxation (SOR) is difficult with the matrix representations used in my implementation (discussed in the next chapter), as column operations cannot be implemented efficiently.

If the matrices to be solved have a known sparsity pattern for which an efficient, special purpose solver exists, such a solver would probably be preferable to using conjugate-gradient. For example, if the matrix is known to be banded with a narrow bandwidth, linear time algorithms can be used. However, selective use of special purpose solvers has not been explored in this thesis as I have tried to emphasize general purpose techniques.

4.3.3 Partitioning the matrix

One very important type of sparsity that will often be useful to exploit in the differential approach is partitioning. In some cases, the rows of the linear system may not all depend on one another, that is, they may be partitioned into separate, smaller pieces, similar to those shown in Figure 4.1. With the differential approach, a partionable matrix occurs whenever there are groups of objects that do not share any controls. Partitioning breaks the large matrix into smaller pieces when they are independent.

The reasons for partitioning the matrix include:

1. Solving several smaller problems will be faster than a single large one if the complexity is greater than linear.

2. If one of the partitions is ill-conditioned, it can have bad effects on the other partitions.

3. Some of the partitions may be trivial to solve. This is especially true in cases like constraint-based drawing where one partition will be receiving user input, and the other partitions will be sitting idle.

Reason 1 is not as important with the conjugate-gradient method, described in Section 4.3.1. In a sense, the Conjugate-Gradient algorithm solves the disconnected partitions in parallel. However, the solver must take the number of iterations required to solve the largest block. The savings is, therefore, not as great as when a straight $O(n^2)$ or $O(n^3)$ solver is used. However, the savings can be considerable when one block requires many iterations, for example if it is large or ill-conditioned, and many other blocks can be solved quickly, either by trivial checks or because they are small.

Reason 2 is particularly important with the conjugate gradient methods described in Section 4.3.1. If any partition of the matrix is ill-conditioned, the steps the solver will take will involve very small direction vectors and very large scaling factors. The parts of the direction vector that correspond to the well-determined partitions of the matrix will contain extremely small numbers, so the large steps should not have any effect. However, because of floating point inaccuracy in representing the very large and very small numbers, much error is introduced. The net effect of this in differential manipulation is that if there are any controls which are ill-conditioned they will cause completely disconnected graphical objects to jiggle.

An algorithm for partitioning

One of the features of partitioning is that it is simple and fast to implement. To partition $\mathbf{JJ^T}$, it is sufficient to order $\mathbf{J}$.

We begin with each variable in a disjoint set. For each constraint, we union the disjoint sets that correspond to each variable that the constraint affects. When completed, we can then gather each set together into the state vector.

The key piece to performing the partitioning is that we can do the disjoint set union and find operations very quickly. In fact, using an extremely simple algorithm, the unions and finds can be performed in nearly linear time[2]. The disjoint set union and find algorithms, along with a complexity analysis, are provided in [Cor89].

The partitioning algorithm runs in time linear with the number of variables and the number of non-zero elements of the matrix. Since the algorithm must actually have the matrix to partition, and filling the matrix takes time proportional to the number of non-zero elements in it, computing **J** is often the most expensive step in partitioning.

4.4 Reducing Problem Size

As described in Section 4.2, solving the linear system in the differential optimization is the dominant factor in the computational complexity of the differential approach. Without placing restrictions on the problems, it is unlikely that we can achieve better than $O(n^2)$ complexity for general constraint problems. We must find ways to keep n small, without restricting the size of the problems that the user actually works on. That is, to find methods which give the user the illusion that the system is working on a larger problem, while in fact, the problem's size has been reduced.

Partitioning, described in the previous section, is an example of a method for transparently reducing the problem size by solving a set of smaller problems rather than the larger problem.

If we know how some variables are going to change by some other means, the expense of solving the differential problem is not required. For example, If we know that an object is frozen in place, we know that it will not move, and that the time derivatives of its variables are simply zero. We can implement the constraints by removing the object's variables from the set of variables solved for, rather than by adding more equations.

For complexity purposes, m and n are really the number of *active* variables and constraints, that is the number that might actually have an affect on the current step. We can discount objects if there are no controls that may cause them to move or if there is something else which requires that they do not change. We can forget a control if it does not effect any changeable objects. We call the set of variables and constraints that are actually participating the *working set.*[3]

In general, adding a new control or constraint adds to n and therefore makes the differential optimization more expensive to compute. However, constraints realized by removing variables from the working set instead reduce m rather than increasing n, speeding computations. Such constraints are represented implicitly in the structure of

[2] Actually, it is inverse Ackerman worse than linear, but since the inverse Ackerman is a small constant (< 5) for any quantity we are likely to encounter, we can consider it to be linear.

[3] The other obvious term, active set, already means something else.

the problem, rather than explicitly by an equation.

Freezing an object is a simple example of a constraint that can be implemented implicitly. Specifying that an object is to be frozen, e.g. that it must not change, could be represented by explicitly placing a control on each variable. However, the effects of these controls are known: they will cause the variables not to change. Since the variables will not change, they can be removed from the working set. Without variables in the working set, the object is constrained not to move, however, this constraint is represented implicitly in the structure of the problem.

Implicit constraints generalize to sets of individual variables. For example, a line segment has four degrees of freedom. Freezing its length can be implemented by explicitly placing a controller on a length connector. However, if the line segment's representation included a separate independent variable for length, this variable could be removed from the working set to implicitly represent the length constraint. This was how the fixed length line segment of Section 3.7 was created.

Implicit constraints are representation dependent. In the example above, if the programmer had chosen a different representation for the line segment, for example to represent it by the positions of its endpoints, the length constraint could not be implemented as an implicit constraint.

Often, it is worthwhile to choose representations to maximize the number of implicit constraints. For example, in a planar mechanisms simulator like the one described in Section 9.2 most line segments represent rigid linkage rods. Therefore, a representation is used that has length as a variable. This way the commonly needed constraint that the line is a rigid length can be realized as an implicit constraint.

Finding new representations of objects is a difficult problem, especially when we cannot anticipate the types of constraints and combinations that will be desired. In fact, the whole differential approach is a response to the problem that representations cannot simply be derived on demand. In terms of physical simulation methods, finding new representations is equivalent to deriving new equations of motion with Lagrangian dynamics techniques.

Finding new parameterizations is equivalent to symbolically solving the non-linear systems of equations. This task is not automatable for any general class of problems. Although it is not possible to create new representations either dynamically or in a general, automatic way, it is possible to create multiple representations for important cases of controls on objects.

It is conceivable to build a system that changes the representation of objects to maximize the number of implicit constraints. This requires solving a combinatorial optimization problem. Globally optimizing for the maximum number of constraints is most likely difficult.[4] Incremental methods might provide different results depending on the order the controls are added, which may be a problem since the behavior of implicit constraints and standard controls differ.

[4] I believe it to be NP-hard, although I do not have a proof.

Another form of implicit constraint is merging, that is having multiple parameters access the same variable, as seen in ThingLab [Bor81]. Merging is an implicit constraint for equating parameters. Because merged parameters share a single variable, they have exactly the same value. A more general variant of merging views a variable as an input. It can either be connected to a slot in the state vector, or connected to some output. This effectively mixes local propagation into the numerical methods.

Implicit constraints are exact. If an object is frozen, it stays exactly fixed. Two merged quantities are exactly equal. While this exact equality can be an asset, it can also be a problem as it means the constraint behaves differently than its explicit counterpart. This can be particularly troublesome in cases where the solver might break the equality constraint slightly, for example to achieve a least squares solution to an over-constrained problem. This distinction becomes significant when the system switches between the two types of constraints.

4.5 Trading Accuracy for Performance

Trading accuracy for performance is an important method for improving the performance of the differential approach.

Using simple ODE solvers with fixed step sizes is one way to trade accuracy for performance. The simpler ODE solvers compute rough solutions quickly, and then permit feedback terms to clean up the results in subsequent steps. This is useful because it gives a rough answer quickly, but provides a more accurate answer over time. For example, in dragging an object, the accuracy needed might be low. When the user stops to examine a situation more closely, the solution has a moment to become more accurate, and by the time the user has decided that a solution is acceptable enough to print or render at high resolution, the constraints are fully converged.

Varying the step size of the ODE solver is another way to trade accuracy for performance. Larger step sizes cause larger apparent velocities on the screen, assuming that control velocities are constant. As discussed in Section 3.3, larger step sizes may be less accurate.

The use of a Conjugate-Gradient linear system solver provides another way to trade accuracy for performance. By using a larger tolerance for the stopping criterion, the solver can accept an answer more quickly if it finds an approximate one.

Many aspects of the methods used to handle inequalities effectively trade accuracy for performance as well. For example, the simple scheme for selecting active sets of Section 6.4.4 trades accurate solutions for faster solving. Not backing up the ODE solver when an inequality boundary is crossed, as will be discussed in Section 6.4.5, may also be viewed as another method to achieve performance by giving up accuracy.

Do not worry about your problems with mathematics. I can assure you that mine are far greater.

— Albert Einstein

Chapter 5

Snap-Together Mathematics

The mathematical techniques of the previous chapters permit controlling graphical objects by specifying the derivatives of functions of their parameters. In this chapter, we consider techniques for defining and evaluating these functions. The challenge stems from the dynamic nature of interactive systems: objects change in response to system actions as the system runs. This means that the functions that define controls must be created on the fly, in response to user actions. In order to effectively implement the mathematical calculations, we must evaluate the functions and their derivatives rapidly. This chapter presents *Snap-Together Mathematics,* a toolkit for dynamically defining functions and rapidly evaluating them and their derivatives.

With the differential approach, objects provide their attributes as output connectors for other objects to use, and as attachment points for controllers that will control the objects. These connectors compute functions of the objects' parameters and input dependencies, and must support the operations of attaching other object inputs and controllers. Snap-Together Mathematics provides a mechanism for realizing the connectors.

"Wiring" connector outputs to inputs builds new, more complicated functions from the elements being assembled. Building a new function may happen any time a new object, constraint or control is defined. It would be unacceptable if building a function required an extensive symbolic math computation or for the program to be recompiled and re-linked. Snap-Together Mathematics explicitly represents the expression graph of connected functional elements as C++ data structures. A connector is simply the output of a node in the expression graph. We will call the nodes *blocks.* To wire an input, it merely needs to be given a reference to some output. Snap-Together Mathematicshas efficient mechanisms for evaluating the values and derivatives of nodes by traversing the graphs.

For example, consider an expression graph that represents connecting the endpoints of two line segments together with an attachment constraint, as shown in schematically Figure 5.1. In this figure, the line segments have a state vector to store their parameters (x, y, θ, l), and provide the positions of their endpoints as connectors. The inputs to the attachment constraint are plugged into these output connectors. The output of the

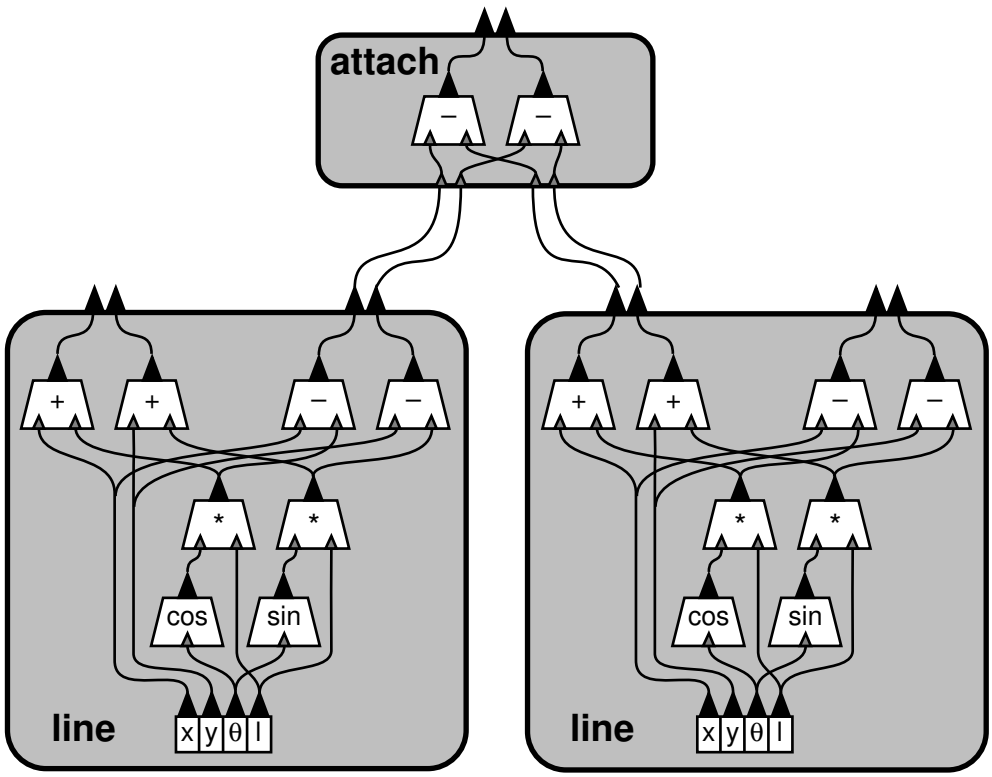

Figure 5.1: An expression graph representing two line segments connected by an attachment constraint. The outputs of the line segment objects, their attribute connector, serve as inputs to the attachment constraints. Function composition is also used inside the objects to build up the functions representing graphical object attributes.

constraint is a function of the variables of both line segments, built by composing the two attribute outputs with its own function. Snap-Together Mathematics allows this plugging to happen dynamically, for example if the user specified the constraint with a mouse click.

Snap-Together Mathematics provides a common protocol for block outputs, permitting any output to be wired to any input. This is important as it allows objects with inputs and those with outputs to be designed independently and dynamically snapped together at run time as needed.

Inside the graphical objects, blocks are wired together to compute the attribute functions. While these object functions could have been explicitly programmed because they would have been known ahead of time, constructing them by wiring together simpler blocks can simplify programming as it allows the programmer to avoid writing the code to evaluate the derivatives.

Several other systems, such as CONDOR [Kas92] and the SPAR Modeling Testbed [FW88], have explicitly represented expression graphs to the user. In contrast,

Snap-Together Mathematics provides the graph data structures as a general purpose tool for the programmer. The programmer could implement a graph viewer and permit the user to have direct access to the data structures. However, such an interface has not been implemented with Snap-Together Mathematics. The style of applications which it has been used to support intend to hide the mathematics from the user. Snap-Together Mathematics has been used for a number of purposes other than the differential approach including physical simulation and optimization-based motion planning.

The elements of Snap-Together Mathematics are not novel. Explicitly representing data flow graphs has been around for decades, and the techniques of automatic differentiation (AD), required to rapidly evaluate the derivatives, are becoming a common practice in the numerical analysis community. Snap-Together Mathematics addresses a very different need than previous AD systems have. Dynamic composition and evaluation of functions and their derivatives was introduced in a system by Witkin and Kass [WK88]. Snap-Together Mathematics is refines these basic ideas in an a simple, general purpose toolkit, allowing direct support for the abstractions of the differential approach. Snap-Together Mathematics was originally developed to support work in interactive physical simulation [WGW90], but has evolved into a more general purpose tool for encapsulating numerical computations.

5.1 Evaluating Functions

Evaluation is the most basic computation to be performed on expression graphs. In the interactive applications which we are considering, expression graphs will be evaluated many times per second, so performance is critical. The most efficient way to repeatedly evaluate an expression is to compile it into machine code. Unfortunately, compiling and linking code for each dynamically created expression is prohibitively expensive in the programming environments presently available.

Other approaches to evaluating the expression graph are interpretive: traverse the graph for each evaluation. Each node of the graph computes its output value, given the values of its inputs. A set of primitive function elements are predefined at compile time to do this. Evaluation of a node involves asking its predecessors for their output values then computing the "local" function of the node.

Performance can be enhanced using caching to exploit two types of redundancy: within an evaluation, common subexpressions need be evaluated only once (these subexpressions may be shared within one expression or between different expressions); between evaluations, certain old values might still be correct if some of the inputs did not change. Re-computation can be avoided by storing the results of a calculation and, for a later request, deciding whether this stored value is still correct. There are many possible ways to implement this cache validation; elaborate schemes might avoid some re-computation, but will require additional computation and storage to make the de-

termination. The more expensive the evaluations become, the more effort should be expended to avoid excess evaluations.

5.2 Evaluating Derivatives

We will need to evaluate the derivatives of expressions with respect to some subset of their inputs. Although the techniques extend to higher derivatives, for this discussion, we will consider computing first derivatives since this is what the differential approach's methods require. Derivatives are taken with respect to a set of variables that we call the *working set of variables.* The working set is a subset of the state vector. We denote the concatenation of this set of variables into a vector by $\mathbf{w}$. For a vector expression $\mathbf{f}$, the Jacobian, or first derivative, $\mathbf{J}$ is the matrix $\partial\mathbf{f}/\partial\mathbf{w}$. In this matrix, each row corresponds to an element of $\mathbf{f}$, while each column corresponds to a variable in $\mathbf{w}$.

There are three basic approaches to computing derivatives: approximate them numerically, derive a symbolic expression for the derivatives, or compose them using a process called *automatic differentiation.* The latter approach has been shown to be superior both in performance and precision of the results [Gri89].

To understand the process of automatic differentiation, consider how derivatives are computed manually. The chain rule allows us to decompose complicated functions into smaller pieces. For example, if our expression is $f = f(a, b, \ldots)$, then the chain rule yields

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial w} + \cdots. \tag{5.1}$$

Differentiation involves recursive applications of the chain rule. If we are able to evaluate the derivative of each of the primitive functions with respect to their inputs then we can apply Equation 5.1 recursively to build the compound expressions. The recursion bottoms out at the constants, whose derivatives are 0, and at the variables, whose derivatives are 1 with respect to themselves and 0 with respect to others.

Symbolic differentiation applies the chain rule to an expression graph to transform it into a new expression graph that evaluates the derivative. The resulting expression must then be simplified to take advantage of the sparsity of the derivatives. Even then, the symbolic differentiation of a vector with respect to a vector yields a matrix of *expressions* which is unwieldy to manage.

Automatic differentiation also applies the chain rule to expressions; however, rather than symbolically composing more complicated expressions, the intermediate results are combined numerically. For any node in the graph, if the inputs to equation 5.1 are concatenated into a vector, the equation multiplies two matrices: the "local" Jacobian of the outputs with respect to the inputs, and the derivatives of the inputs with respect to the working set.

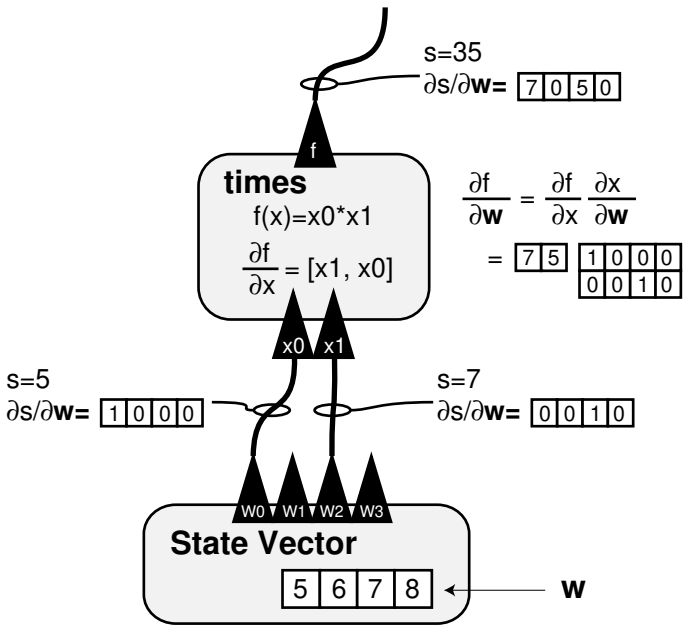

Figure 5.2: A simple example of derivative composition. Signals carry both values (s) and their derivatives ($\partial s/\partial \mathbf{w}$). The function block computes its internal Jacobian and composes the global Jacobian by multiplying the matrices.

We implement automatic differentiation by augmenting the expression graph with the ability to pass derivatives as well as values along edges. In addition to computing its output values, each node of the expression graph must also be able to compute the value of its local derivative, also a function of its inputs. The composition process builds the "global" Jacobian by multiplying this matrix with intermediate result matrices. By passing the entire intermediate result matrices along the edges of the graph, the derivative matrix can be built in one traversal of the expression graph. The same mechanisms for sharing intermediate results by caching as discussed in the previous section apply.

A recursive descent of the expression graph computes the derivative matrix. Each node in the graph is able to respond to requests for the derivative of its output with respect to the current working set. Constants and variables not in the working set return zero in response to this query. A state variable in the current working set returns a vector with one in the position corresponding to the variable, and zeros elsewhere. After determining that its cached value is not valid, a non-terminal node recursively asks its children for their derivatives, computes its local Jacobian, and multiplies these together to produce its derivative with respect to the current working set. Figure 5.2 demonstrates a simple example. Edges of the expression graph pass not only values, but also their derivatives.

This method of automatic differentiation assembles the Jacobian bottom-up, and is called the forward-mode. The alternative reverse-mode, or top-down, approach is presented by [Gri89] and implemented for an interactive system by [Sap93]. This algorithm reverses the order of the matrix multiplies, building the Jacobian matrix from the top down. It has the advantage that the intermediate result matrices are of small, fixed size. In the bottom-up approach, the size of intermediate matrices depends on the number of variables which contribute to that derivative. Because the intermediate results are fixed-sized, the top-down approach can achieve linear asymptotic complexity in places where the bottom-up approach has $O(n^2)$ complexity. However, this increased worst-case performance on dense problems comes at the expense of considerable bookkeeping, inability to fully exploit sparsity, inability to share intermediate results, much higher time constants, and difficulty in changing working sets.

It is important to recognize the generality of either of these derivative composition processes. Each node of the expression graph need only to be able to compute its *local Jacobian,* the derivative of its outputs with respect to its inputs. This matrix is a function only of the input values, not their derivatives. Given the local Jacobians, the composition process merely multiplies the matrices together to build the global derivatives.

5.3 Sparse Representations

The critical performance issue in building the Jacobian matrix, as well as most calculations that use this result, is exploiting sparsity. Bottom-up matrix passing schemes exploit sparsity by using sparse representations for the intermediate matrices. There are many possible ways to represent a sparse matrix, with many tradeoffs to consider in selecting a representation [DER86]. This decision is central to the design of an implementation. One particular representation with which we have had success is the half sparse matrix: a full vector of sparse vectors, as depicted in Figure 5.3. We call a system based on these data structures a *sparse vector* scheme.

In the sparse vector scheme we consider every output in the expression as an independent scalar, even if higher levels will interpret them as pieces of larger structures. A function block can have multiple scalar outputs. The gradient of each scalar is a sparse vector ($\partial x/\partial \mathbf{w}$).

Sparse vectors can be represented as a list of pairs $(index, value)$, taking space linear in the number of non-zero elements. If this list is sorted by index, we can perform the essential vector operations in time linear in the number of non-zero elements. For single vector operations, such as multiplying by a scalar or finding the magnitude, the algorithms simply run through the list. For multi-vector operations, such as addition, linear combination, or dot product, we exploit the sorted order of the lists and step through both in parallel, advancing which ever has the least index. These algorithms

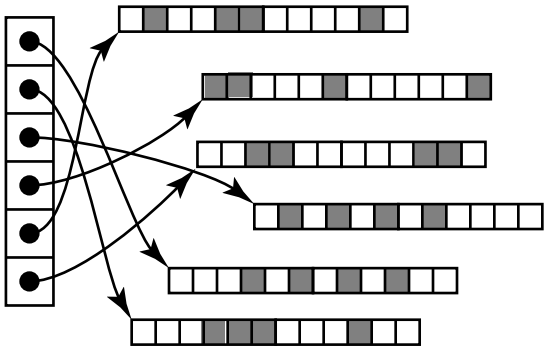

Figure 5.3: A *half-sparse* matrix representation store the matrix as a full vector of sparse vectors. Each entry in an array points to a sparse vector, depicted here by a sparsity pattern.

maintain the representation invariant so re-sorting is not needed.

Each derivative in the expression graph is represented as a sparse vector, the derivative of a scalar with respect to a set of variables. For each graph, one set of variables is denoted as the current working set: all derivatives are with respect to this set. For each variable, we must know an index to the corresponding column of the Jacobian.

Sparse vectors are collected into matrices which are half-sparse. While this is a non-standard representation, it does permit the operations required by the numerical methods we employ. In particular, half-sparse matrices can be multiplied by a vector or by its transpose rapidly. These methods are the essential computations in iterative linear system solvers like conjugate gradient [PFTV86]. The multiplication routines are among the most important in the entire implementation of the differential approach as they form the inner loop of $O(n^2)$ portion of the computation. However, the routines are very simple and can be coded efficiently. The algorithms are:

$\mathbf{v} = \mathbf{H}\mathbf{x}$
multiply `HalfSparseMatrix` **H**, `Vector` **x** $\Rightarrow$ `Vector` **v**
    v = 0
    `for` $i = 0 \ldots \mathbf{H}.rows$
        `for` $j = 0 \ldots \mathbf{H}.row[i].elements$
            $\mathbf{v}[i] += \mathbf{H}.row[i].value[j] * \mathbf{x}[\mathbf{H}.row(i).index[j]]$

$\mathbf{v} = \mathbf{H^T x}$

```
multiplyTranspose HalfSparseMatrix H, Vector x ⇒ Vector v
    v = 0
    for i = 0...H.rows
        for j = 0...H.row[i].elements
            v[H.row[i].index[j]] += x[i] * H.row[i].value[j]
```

5.4 The Snap-Together Math Library

The machinery of the differential approach is encapsulated into a C++ toolkit called Snap-Together Math. The library implements the function composition and evaluation discussed in this chapter, as well as the differential solver.

5.4.1 The Protocol for Function Outputs

One of the essential elements to implement the differential approach is the standard protocol for connectors, the outputs of objects. Connectors correspond directly to Snap-Together Math block outputs. When a controller or object "plugs in" to a connector, it merely stores a reference to a Snap-Together Math output.

In order to be a Snap-Together Mathematics output, and object merely must provide two functions: one that computes its output value, and one that computes the derivatives of its values with respect to the current working set. In my implementation, the class of objects that have Snap-Together Mathematics outputs is the class `Port`. Each `Port` may provide a number of scalar outputs. The two methods that all `Port` subclasses must provide take an index that specifies which scalar value is being referred to, and return either the real number value of the output, or a pointer to a sparse vector (class `SpVec`) containing the derivative of the value with respect to the current working set. The signature for the minimum set of methods is:

```
class Port {
  public:
     virtual Real   o(const int);
     virtual SpVec* grad(const int);
}
```

In order to be a mathematical output, an object is required to provide only these two methods. The protocol supports many optional methods and fields such as the width

(the number of scalar values the object provides), strings that provide names for the signals, and nominal value ranges. To identify a particular scalar output, a pairing of a `Port` and an integer index is required. This is referred to as a `Signal`. Signals are constructed by pairing a pointer to a `Port` and an integer index.

Graphical objects could be subclasses of `Port` so that they could provide connectors for their attributes. However, this gives little structure when there are many outputs on a single objects. Instead, a graphical object will typically define other surrogate `Port` objects which provide smaller numbers of scalar outputs. For example, a polygon object might keep a list of vertices, each of which would be a `Port` object. This is important because these surrogate objects can be standardized. For example, all polygons could use the same vertex objects, so attaching to a vertex could be done independently of the polygon type.

Standardizing `Port` types adds further modularity. Snap-Together Mathematics permits connecting any scalar input and output. However, often such connections are most useful when they are grouped together and typed. For example, A 2D distance object has 4 scalar inputs, however these are most meaningful when they are thought of as two 2D point inputs. Graphical objects would provide specialized `Ports` which meet this standard.

Several standard types of `Ports` will be discussed in the following sections.

5.4.2 Functional Elements

The most basic element of Snap-Together Mathematics are objects which can be used in function composition. As explained in Section 5.4.1, these objects are subclasses of `Port` and must provide a few basic methods. Part of the beauty of the Snap-Together Mathematics scheme is the `Port` class is minimal enough that its functionality can be added to application classes. However, the Snap-Together Mathematics toolkit provides a variety of types of `Ports` for general mathematical elements that can be combined to build more complex structures.

The most fundamental `Port` subclass is `FBlock`, the class of function blocks. These objects compute mathematical functions of their inputs. The standard library includes various primitive functions, including almost all of the functionality of a scientific calculator. The class `FBlock` is implemented to have a fixed number of inputs for each subclass. Other special subclasses of `Port` can do things such as sum a variable number of inputs or take the magnitude of a vector of inputs. This permits creating controls on aggregate collections of objects, as in Garnet [VZMGS94].

Wiring functions together simply requires inserting an output `Signal` into the input field of a function block. The class `FBlock` stores its inputs as an array of `Signals`. Connecting the output of one function block to the input of another looks like

```
block1->ins[0] = Signal(block2,0);
```

where `block1` is a pointer to a function block (`FBlock*`), but `block2` can be a pointer to any subclass of `Port`. This code fragment connects the first output of the object pointed to by `block2` into the first input of the function block pointed to by `block1`. Notice that this wire is not explicitly represented, nor does the output port receive any indication that it is being attached to.

Defining a new function block requires specifying two methods. One that computes the function, and another that computes its internal Jacobian. It is important to note that this is really the only place where a programmer might have to take a derivative.

Writing the derivative routines is a simple matter of mechanically applying the rules of freshman calculus. However, this can be tedious as the functions get complicated, especially since we have a strong desire to have optimized code. Because the process is mechanical, it can be automated.

The *BlockMaker* tool automatically generates code for new function block types from mathematical expression. The tool, is written within the Mathematica symbolic algebra system [Wol88]. The tool takes as input an expression that describes the function block to be created, and a small amount of auxiliary information such as the number of inputs to the block and the name for the C++ class for the function block. The output of the tool is a C++ program file that contains the code for the block's methods, as well as a C++ header file describing the new class. The generated code is optimized using Mathematica's simplification tools as well as a common subexpression remover that I developed with Stephen Schwab. Because the tool runs within Mathematica, the full power of the symbolic algebra system is available to define the expressions used to create blocks.

New function block types are not often required. The Snap-Together Mathematics library includes many basic functions, such as those found on a scientific calculator, and more complicated functions can be created by composing these elements together. The main reason to create new blocks is efficiency. Composing a function out of other function blocks is more expensive than compiled code if the compiled code is optimized to exploit internal sparsity within the block and to share common subexpressions. For derivative evaluations, properly optimized code will perform the same evaluations as done by automatic differentiation. However, because it is explicitly compiled, there is less overhead. Automatic differentiation works better for more complex functions whose symbolic derivatives would be difficult to optimize.

5.4.3 Representing State Variables

The leaves of the expression graphs are constants and variables. The two are distinguished from one another in that the derivatives for the variables are not zero when taken with respect to itself, while the derivatives for the constants are always 0. Implementing a class to represent constant values is, therefore, simple; its methods just return constant values. The derivative of a variable must have a 1 in the column of the

gradient that corresponds to that variable.

The simple protocol for Snap-Together Mathematics does not address the issue of defining the working set of variables and the mapping of its members to columns of the Jacobian. This is indicative of the larger issue of managing collections of variables. On one hand, building systems in an object-oriented manner requires the state of the system to be distributed into the objects themselves. But, mathematical algorithms typically require this state in the form of contiguous vectors, which are gathered, ordered collections. This ordering also gives meaning to the columns of the Jacobian matrices.

There are many potential schemes for representing variables in a Snap-Together Math implementation ranging from having objects allocate space in a global state vector to modifying our numerical algorithms so that they operate on distributed, non-contiguous, vectors. Several of these were explored in earlier tools. The Snap-Together Mathematics library uses a combination of centralized and distributed representation. Objects each have their own state, however these variables are "gathered" into a centralized state vector for numerical computations. When an object's variable has been gathered, it knows where in the global vector to find it so it can still retrieve its value as well as index it for creating derivatives. In the context of Snap-Together Mathematics, derivatives can be taken only when variables are gathered, as this is the only time when variables correspond to matrix columns. When the numerical computations are complete, the values are scattered back into the corresponding smaller vectors.

The ability to scatter and gather variables has an important advantage over always keeping the variables centralized. It allows for the set of variables to be changed rapidly. This not only simplifies adding and deleting objects from the working set, but also makes it easy and efficient to operate on subsets of the variables. Techniques that make use of this latter feature are discussed in Section 4.4.

The scatter/gather scheme uses two main data structures: one to represent the smaller individual state vectors, and one to store the gathered global state vector. In Snap-Together Mathematics these classes are called `StObj` and `StVec` respectively. Each of these classes is a subclass of `Port,` although `StVec` objects rarely have connections made to them.

`StObjs` are objects that store a number of state variables. Each graphical object would have one that stores its configuration. `StObjs` are also used to store constants by marking their variables so that they are never gathered.

The gathering operation takes a list of `StObjs` and assigns designated variables in them to elements in an `StVec,` as depicted in Figure 5.4. Variables store their assigned location. If their value or derivative is requested when they are in an assigned state, they forward the request on to the `StVec.` If they are not assigned, they return the value stored internally and 0 for their derivative. A scatter operation returns each variable in the `StVec` to its corresponding `StObj,` updating the `StObj`'s internal value, and removing the assignment.

The `StVec` provides a contiguous vector for mathematical computations. Routines

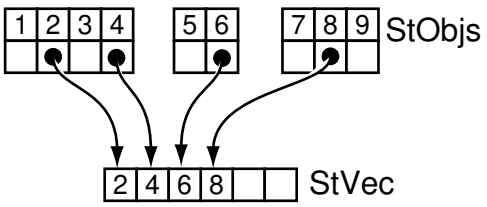

Figure 5.4: Selected variables from `StObj` objects are gathered into the `StVec` object. Variables in the working set point to slots in the `StVec` object.

such as ODE solvers look at the data stored here. The indices of the `StVec` define the columns of the Jacobian: requests for derivatives of the `StVec` return vectors with a 1 value in the corresponding column.

An important part of the scheme for storing state variables is the ability to gather[1] selectively. This provides the ability to switch a variable off, or turn it into a constant. Uses for disabling variables are described in Section 4.4.

During a gather operation, a function is provided that decides if a variable is to be gathered or not. Typically, the default function is used. This function makes its decisions based on a bit vector stored with each variable. The bit vector and function provide the following mechanisms for selection:

- Each variable belongs to a *caste* that identifies its type. Whether or not a given caste is to be gathered can be decided independently. This permits operations like "gather only the variables that affect lighting."
- Each variable has several type bits, which permit denoting the expected use of the variable. For example, if a variable is denoted as a constant, it will not be gathered.
- Each variable has a counter that freezes it whenever the count is non-zero. A problem with using a single bit for this purpose is that it is impossible to determine how many constraints are freezing a variable. If two constraints freeze the same variable, deleting one of the constraints would re-enable the variable. A counter re-enables the variable only when all freezes have been removed.

Selective gathering also provides a mechanism for merges, or equating two variables. During a gather, two variables can be made to share one space in the `StVec`. This will effectively merge them, constraining them to have the same value. A similar technique can be use to constrain a variable to have the same value as any other `Signal`. Such features are not used much in the Snap-Together Mathematics implementation, because they make the optimization of the next paragraph impossible.

[1] The corresponding scatter operations always scatter what was last gathered, so there is no selection involved in them.

An important special case of a function block is one that has each of its inputs connected to the same `StObj`. This is a very common case, used often for graphical objects where the function blocks each compute some attribute. Because all of the inputs are connected to variables, the Jacobian of the inputs of the block is the identity matrix, possibly with some columns missing. By exploiting this, the matrix multiply to compute the block's Jacobian can be considerably faster. This leads to substantial speedups in many applications, as these direct-connected blocks are very common, and very often constitute the majority of the "wide" input blocks, which are the most expensive to compute.

Partitioning, as introduced in Section 4.3.3, reorders the elements of the `StVec` so that variables in a common partition are next to one another, facilitating solving each subset independently. Partitioning is done only when a gather operation is performed, not each time the linear system is solved. This is done because re-ordering the state vector would confuse the process of differential equation solving.

Because the partitioning algorithm does not actually look at the values of $\mathbf{J}$, they do not need to be computed when finding the initial matrix to partition. In fact, rather than using the real values of the matrix, it can be better to use binary values which represent that the matrix element might be non-zero, rather than that it is non-zero for the current value of the state vector. This is easily achieved by having each `Port` provide a method which works like the `grad(int)` method, but does a logical or instead of a linear combination of its input vectors. For objects that do not provide this optional method, the gradient may be used instead.

5.4.4 Caching in Snap-Together Math

As hinted at in Section 5.4.1, caching is an important tool for enhancing performance in Snap-Together Mathematics evaluations. Whenever a value or derivative is computed by a function block, or other `Port` type that implements caching, it is stored inside the block in case the result is used again. Each time a value or derivative is requested, the block first decides if its cached value is valid; if it is, it avoids re-computation and simply returns the cached value.

The Snap-Together Mathematics toolkit employs a simple cache validation scheme. A single global timestamp is used. Whenever any of the inputs (state variables) are changed, this timestamp is incremented. A block must recompute if its internal timestamp is older than the global timestamp. This scheme is very simple but has the obvious problem that it invalidates much more than needs to be. Changing a single value invalidates all caches in the entire system.

In the context of the differential approach, invalidating all the caches simultaneously is not as catastrophic as it might seem since the variables are all updated simultaneously with each ODE solver call for evaluation of the differential optimization. Schemes that do substantially better require some combination of substantial amounts

of graph traversal, explicitly representing links bidirectionally, doing numerical and sparse data structure comparisons, or exploiting knowledge about the particular problem. For most applications, such complication is not warranted as, at best, it serves only to reduce the constants on the linear complexity portions of the differential approach.

5.4.5 The Differential Solver

The Snap-Together Mathematics toolkit encapsulates the abstractions of the differential approach in a set of classes that implement the techniques of the previous chapters. An object is used to represent the differential optimization problem, storing information about the controls and the variables they effect. This object is called a constraint engine or `ConstEngine`.

The `ConstEngine` class has fields that contain an `StVec` and a list of `StObj` that are to be gathered for computations. A `ConstEngine` object also stores information to define the objective function for the differential optimization, such as a list of `Signals` that make up $\mathbf{g}$.

The controls for the differential optimization problem are stored in a `ConstEngine` by a list of `Controller` objects. `Controllers` are objects that specify desired derivatives for connectors, as discussed in the next chapter. The Snap-Together Mathematics class for a controller specifies a `Signal` to control, a controller type, and several parameters.

Solving the differential optimization problem, e.g. using the techniques of Chapter 3 and Chapter 4, is executed by a method of the `ConstEngine` class. This is the only part of the system in which constraint solving needs to be done. This method is implemented in a manner that interfaces with the ODE solver implementations of the underlying mathematics toolkit. The `ConstEngine` class also interfaces with non-linear iterative solvers.

Snap-Together Mathematics is built on top of an object-oriented mathematics toolkit that I wrote. The toolkit includes an object-oriented framework for defining ODE problems and solvers. The class `Integrand` represents an ODE problem by defining a single method that defines a function to compute $\dot{\mathbf{q}}$ given $\mathbf{q}$ and t. An `ODESolver` object stores an integrand and an initial condition and offers a method to step time forward. The `ConstEngine` class is a subtype of `Integrand`.

When used as an `Integrand` to solve a differential optimization problem, a `ConstEngine` must first load the $\mathbf{q}$ vector provided by the ODE solver into its `StVec`. Loading the state allows the solver to try different values for the state in the process of taking an ODE step. The `ConstEngine` keeps all of the intermediate results of the solving process, such as the Lagrange multipliers, as internal fields.

5.4.6 The Whisper/Snap-Together Math Interface

While Snap-Together Mathematics is a C++ toolkit, an interface is provided to it as an extension to the Whisper interpreter described in Appendix A. The extension adds several new data types to Whisper, as well as many new primitive functions. `WhSTM` provides a convenient way to access to the functionality of the Snap-Together Mathematics, and can be used on its own to develop simple applications completely in Whisper. The Whisper/Snap-Together Mathematics interface, called `WhSTM,` shows how the functionality of Snap-Together Mathematicscan be provided in a more convenient form, and will be used in later portions of the thesis.

`WhSTM` provides primitive Whisper data types for most the Snap-Together Mathematics classes `Port`, `FBlock`, `StObj`, `Signal`, and `Const`. Other classes, generated by primitives written in C++ are generally treated by the more generic class that is appropriate. For example, a summation block, which is not an `FBlock` because it allows variable numbers of inputs, simply appears as a `Port` in Whisper. No facility for defining new subtypes of Snap-Together Mathematics classes is provided in Whisper. `WhSTM` defines 88 primitives, including creation functions for 30 different types of function blocks.

`WhSTM` supports automatic promotion of types as needed. Any function requiring a `Port` can take anything that is a subtype, including `FBlock` and `StObj` types, even though Whisper has no subtyping mechanism. Real numbers are also promoted to `Port` where necessary; a constant valued port object is automatically created.

The constructor for `Signal` takes a `Port` and an index. It permits specifying the index either as an integer or as a string name if the block being connected to supplied the optional names for its outputs. If the index is omitted, it is assumed to be 0. This allows a `Port` to be promoted to a `Signal` when needed.

Function block creation routines all optionally take initial values for the signal input. The convenience of this, coupled with the automatic promotions, is demonstrated in this simple example

```
(set b (plus-block (times-block (signal point-port 2) 5)
                   (signal point-port 'y)))
```

that computes the sum of 5 times the z value of a 3 output port representing a Cartesian coordinate and its y value. The ease with which functions can be built in Whisper will be used in the Bramble toolkit, described in Chapter 7. A more extensive example of using `WhSTM` to build function graphs is described in Section A.2.

The Whisper interface does not provide constraint engines or ODE solvers as basic types. However, `WhSTM` does have default instances of these objects, so that commands like

```
(add-const (controller sig '= 2.0))
```

implies the use of the "built-in" constraint engine, and

```
(rk4-step .1)
```

uses the built-in 4th order Runge-Kutta solver instance to step the default constraint engine forward a time step. Other packages further extending Whisper can alter the defaults. This will be used extensively in Bramble.

If you don't know where you're going, you will wind up somewhere else.

— Yogi Berra

Chapter 6

Controllers

The previous three chapters provided the machinery required to implement the differential approach. This machinery permits specifying desired derivatives for function outputs. Appropriate changes to the state of the objects are computed from these derivatives. In this chapter, we consider how to specify the derivatives in order to create graphical interaction techniques.

With the differential approach, graphical objects are manipulated by specifying how particular attributes of the objects should change. Graphical objects provide their attributes as output connectors. A connector serves as a control when its behavior is specified. The specification of the derivative of a connector is encapsulated into an object called a *controller.* These objects are plugged into any connector output just as dependency inputs are. A controller can look at the output it is connected to and the external world (e.g. input devices) in order to produce a desired derivative value for the connector.

The capabilities of a controller are limited: they can specify only a rate of change. Objects move by having controllers attached to their connectors over a period of time, continuously specifying derivative values which are converted by the differential optimization to derivatives of the object parameters.

The basic building blocks of graphical interaction techniques are controllers and connectors. For the purposes of interaction, the set of graphical objects appears as a set of connectors, any of which can serve as controls by having a controller attached to them. The chapter begins by showing how basic methods such as dragging are created by attaching controllers to connectors for periods of time. Subsequent sections explain how the continuous model of time is coupled with events and describes the details of the controllers themselves. Some details of creating complex controllers that switch controls on and off are discussed in the chapter's final section.

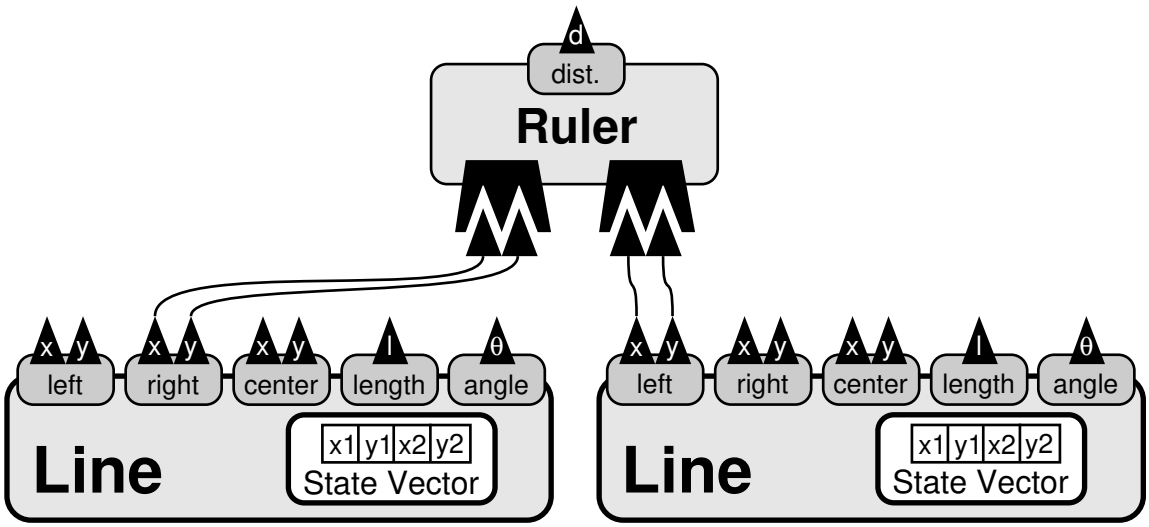

Figure 6.1: A schematic "wiring diagram" of two line segments with an attachment constraint between their endpoints. For the purposes of defining manipulations, these objects appear as a set of connectors awaiting controllers to be plugged into them.

6.1 Example Interactions

To show how the machinery of the differential approach applies to manipulating graphical objects, we consider some concrete examples. For these examples, our model will consist of 3 graphical objects: two line segments, and an attachment object that is connected to one endpoint of each line segment. A schematic of the data structures is given in Figure 6.1.

6.1.1 Specifying Values

A basic operation is to specify a value for a connector, for example to specify a position for an endpoint of a line segment. This cannot be done directly with the differential approach: only rates of change can be specified. In order to achieve a desired value, we must specify the derivative over a period of time, and wait for the object to achieve the desired value.

The `GoTowards` controller makes a connector's value move towards a target. At each instant, it computes a derivative that moves the value towards the desired goal, as will be discussed in more detail in Section 6.3. A `GoTowards` controller does not specify its attached connector's value. However, by pushing its value in the right direction, over time the connector will reach its target unless something impedes its progress. In order to specify a target value, we place a `GoTowards` and wait for the

value to reach its target.

If the `GoTowards` controller is left attached after the attached connector reaches its target, the controller will continue to drive the value towards the target, holding it in place. This is how constraints are created. Because the `GoTowards` continually pushes toward the goal, the constraint will be restored if it drifts. The `GoTowards` controller, therefore, include constraint stabilization like that proposed by Baumgarte [Bau72].

To create a constraint, a controller must be applied to some connector. The attachment object which simply provides the connector is not a geometric constraint by itself. However, a geometric connection constraint object would be a version of the attachment object that created `GoTowards` 0 controllers on its connectors when it was placed, and removed these controllers when it was removed.

6.1.2 Dragging

To drag an object, we can permit the user to specify where a particular point on it should be positioned. When the mouse button is pressed to grab a point on an object, a differential controller is attached to the output that computes the position of the point. This controller guides the point to follow the mouse. When the button is released, the controller is removed from the output connector of the dragged point. The addition of an optimization objective term with the mouse position to provide manipulation of a graphical object is first presented in [KWT88].

The `Follow` controller provides derivative values that cause its attached controller to move towards tracking a moving target. It is similar to a `GoTowards`, with the exception that rather than having a fixed target, it has a moving target. This behavior could be created with a `GoTowards` controller that periodically has its target value updated. However, `Follow` controllers can sometimes provide better tracking behavior by using information about the motion of its target. In the case of tracking an input device, where derivative information is difficult to estimate for lack of a good predictive model, a `Follow` controller is typically implemented as a `GoTowards`. More details about the `Follow` controller are provided in Section 6.3.

For many reasons, it is unlikely that the point being dragged will track the cursor exactly. When the cursor moves quickly, it might take the system a few steps for the dragged object catch up, and even then, if the cursor moves far during the step, the two will separate. Also, the rates at which graphical objects can move is sometimes limited by how well the ODE solver can solve for the motion; if the object moves too quickly, the ODE solver will be unable to accurately compute its motion. Faster steps and better algorithms can reduce the separation, however the mouse might still separate from the object being dragged if the motion is restricted by a constraint.

When the cursor and dragged point diverge, it is important to provide visual feedback to connect them. Typically, a line is drawn between the cursor and the point being

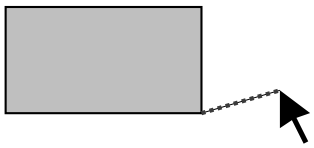

Figure 6.2: A small line is used to connect the mouse pointer to the object being dragged. This feedback is important because the cursor location will diverge from the location of the object because of lag, integration error, or restrictions on the object's motion.

dragged, as shown in Figure 6.2. Metaphorically the cursor is connected to the dragged point with a rubber band. This metaphor is a pretty accurate one for the spring attraction techniques introduced in Section 3.5.1.

The generality of the differential approach is apparent in the simple example of dragging. Graphical objects provide point position aspects. The dragging behavior can be plugged into any objects' output. The differential solving process will compute how to control the objects' parameters in order to achieve the desired motion. If a different pointing device, for example a 3D tracker, is available, it too could be plugged in to the same places.

6.1.3 Constrained Dragging

The ability to mix constraints and controls is important to the effective use of constraints. The constraints permit the user to drag objects without violating any previously established relationships. User defined persistent constraints can provide many services, including helping to avoid redundant work, helping to explore a constrained space, or helping to construct compound objects.

Because constraints are easy to attach and detach, they can be used with dragging controls to define interaction techniques. In addition to creating a `Follow` or `GoTowards` controller to cause the mouse to be tracked, additional constraints are simultaneously added. These constraints are removed when the dragging controller is removed.

For an example, consider rotating a graphical object. To create a rotation, the object could be dragged subject to a constraint that nails the position of its center in place. Dragging points on the object will then cause the object to rotate, assuming the object has that degree of freedom. Dragging might also cause the object to stretch, if the dragging motion is not circular about the center of rotation. This can be combatted by keeping the cursor on a circular track or by using another constraint to keep the size constant. In the latter case, some mechanism that makes the dragging constraint break rather than the others will be required. The dragging control must be subject to the other constraints. Techniques for achieving this are discussed in Section 3.5.

The ability to combine constraints and controls allows a set of building blocks to be

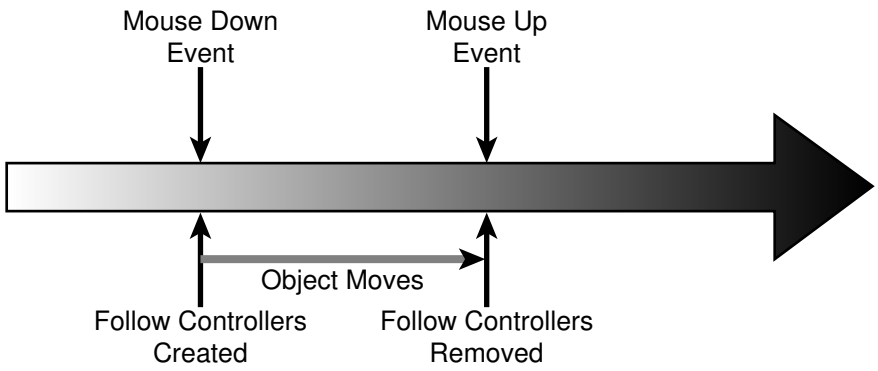

Figure 6.3: A timeline of an interaction in the differential approach. The arrow symbolizes the flow of time. At discrete instants, such as when the mouse events are received, the set of differential controllers is altered. The effects of these over time causes changes in object configurations.

provided for constructing interaction techniques. For example, it provides a constraint-based strategy for developing 3D manipulation techniques, as discussed in Chapter 8. The basic idea is that enough constraints are provided so that the input device is sufficient to fully determine the motion. This strategy is not unlike what we employ in the real world where we use things such as jigs and braces to help us manipulate hard to handle objects.

6.2 Continuous Time

As illustrated by the above examples, object configurations are altered by having controllers specify their behaviors over a period of time. The state of objects cannot be changed instantaneously. Instead, it changes over time, the way an object in the real world does. This is quite different than traditional methods for constructing interactive systems, where the state of objects is updated at discrete instants corresponding to events or polling increments.

In concept, time in the differential approach continuously moves forward. While time progresses, any active differential controllers are causing the objects they affect to change. At discrete instants, such as a window system event, controllers may be created, destroyed or altered, however, the state of objects is not changed. To change the configuration of an object, a controller must be created and time must pass so that the object can adjust itself accordingly. A time line of such a process is depicted in Figure 6.3.

Since a controller can be plugged into any connector at any time, placing a controller can be asynchronous with the placement of other controllers. This makes it easy to

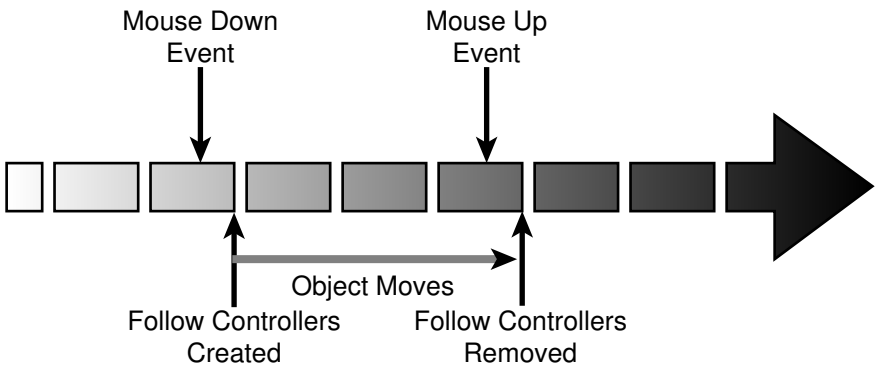

Figure 6.4: A timeline of interaction in the differential approach, as actually implemented. The arrow symbolizes the flow of time. Time is discretized into steps. A discrete event is deferred until the end of the step it occurs in.

perform concurrent operations, such as allowing for asynchronous two-handed input.

In practice, we can only approximate the continuous flow of time discretely. As discussed in Section 3.3, we must treat the ordinary differential equations describing object trajectories as a sequence of steps in order to solve them. During each step, time is advanced a small amount. By making these steps small enough, the user is given the illusion of continuous motion.

As implemented in the differential approach, events are deferred until the end of the step, as depicted in the timeline of Figure 6.4. If the duration of the steps is very small, the delays will not have an effect. If avoiding this lag is crucial, the approach can be modified slightly to trade total number of steps for reduced lag. For example, when an event occurs, the current step could be aborted and later restarted after the event is processed. Also, time may be stopped during periods when no objects are moving and user events are expected. For example, in a standard direct manipulation system, when the user is not dragging an object, nothing is moving so differential time may be stopped and the system may spend its full effort responding to events. An event may need to start time, for example if it initiates a dragging operation.

After each step, the system must redraw the displays. For the prototypes of this thesis, screen update always redraws the entire display. Other systems might attempt to speed redraw by selectively updating only objects that have changed. I have not taken such an approach because:

1. Selective redisplay is more difficult with the differential approach since many objects may be moving at once. In fact, the system must be able to determine when selective redraw is appropriate, and when it would simply be faster to redraw the entire view.

2. Selective redraw is difficult to implement for 3D applications when using a Z-

buffer hidden surface removal algorithm, because the state of the Z-buffer must be restored when objects are removed from the display.

3. The differential approach has been based on the assumption that fast drawing hardware is available.

6.2.1 Differential Time and Clock Time

The time in the differential approach is advanced at each ODE solver step. There is no assurance that the "clock" in the differential approach, that is the time that passes from solver steps, will correspond to wall-clock real time. In fact, making differential and real time correspond would be difficult: a system would have to be able to accurately predict how long (in real time) it will take to do the computations required for each step, and make sure the proper amount of differential time advances on each step. This rules out the possibility of the system adaptively controlling time steps when things grow difficult, and would require complicated synchronization when the real world clock and the differential time differ.

Rather than coupling differential time and real time, we instead let the clocks float. The differential "clock" can be thought of as a dimensionless quantity: it's mapping to "wall clock" time is unknown, and unimportant. Whether the clock moves quickly and velocities are low, or the clock moves slowly and velocities are high, the same effects can be achieved. It becomes impossible to express controls in terms of real time, for example to say that an object moves across the screen at 3 inches per second, or that a point reaches its target in 250 milliseconds. However, other imprecisions also make this impossible, both spatially (how far is 3 inches when the user can scale the window or run on a different monitor) and temporally (how can we be sure that in that 250 milliseconds the system will not have to swap or process a higher priority job). While multi-media systems researchers, such as [DNN+93], are beginning to study with such real-time issues, the differential approach, like most interactive graphics systems, is not concerned with real-time performance. Coupling with real time is left for future work in Section 10.3, but a simple approach would take a step as fast as possible and then wait for the real time clock to catch up.

6.2.2 Breakdown of Interactivity

The differential approach relies on being able to provide a large number of steps per second. This is needed both to provide the illusion of continuous motion to the user, and to insure events are processed without too much lag since they are deferred to the end of a step. As the introduction to this thesis states, modern computers continue to provide increasing amounts of graphical and numerical processing capabilities. However, even though the capabilities of computers may continue to grow exponentially, the types of

problems that users are interested in tackling may similarly grow without bound. As the task grows, the amount of computation required to support differential interaction also grows. At some point, processing each step requires too much time and the quality of interaction suffers. This is called *interactive breakdown.*

It is difficult to determine precisely the rate at which interactive breakdown occurs. Even when the motion appears jerky, it can sometimes be acceptable, depending on the user and the problem. For some applications, high frame rate is crucial, for example when the motion is needed to help the user comprehend the behavior of objects. Similarly, event delay lag is most bothersome when the user is generating many events in rapid succession or that are time critical, such as selecting a moving object. As described in Section 6.2, event processing lag can be reduced at the expense of throughput.

As the complexity of the problem goes up, so does the time required to do each step. The immense scale of the objects that people design – a modern jet airplane may have *millions* of components[Hei93] – makes keeping interactive rates impractical for some problems. However, the limitations of the human cognitive and perceptual abilities make it unlikely that a user would want to operate a model of that scale interactively. The prototype implementations show that models of reasonable complexity can be handled on the current generation of computers. Performance of the prototypes is discussed in Appendix B.

6.3 Basic Controllers

Controllers are objects which attach to connectors and specify values for their derivatives. The examples of Section 6.1 briefly introduced some types of controllers. Here, we examine these controllers in more detail, introduce a more complete set of controllers, and discuss how they compute the derivative values.

Since a controller's sole function is to provide a derivative value, deciding what values to specify is the critical issue in designing a controller. This is complicated by the fact that interface elements are traditionally not defined by derivatives, but rather by positions. Another issue in picking velocities is that simulation time is not coupled to real time in a meaningful way, as discussed in Section 6.2.1. This makes it impossible to specify velocities in terms of apparent velocities, although this would be attractive to insure that the user can follow the motion. However, the more relevant speed limit is typically given by the ODE solver, as discussed in Section 3.3.

The simplest controller provides a constant derivative value. These `Constant` controllers are rarely used because attributes typically do not move at fixed velocities regardless of other factors, and determining specific velocity constants is difficult because time and space do not precisely correspond to real-world quantities. In the cases where they are used, `Constant` controllers are typically given values which

are found empirically to produce a desired rate in certain situations. The more common controllers do not inherently specify a time, but rather a target and provide the controller with flexibility in choosing the velocity that achieves it.

More useful controllers examine the value of the connector they are attached to in order to select a derivative value. The two basic varieties of such controllers are `GoTowards` and `Follow`. Each of these picks derivative values to make its attached connector achieve a target value. The difference is that a `GoTowards` has a fixed target, while `Follow` has a moving target such as a motion path or an input device. An important distinction is that a `Follow` may be able to use information about the motion of its target in order to track better.

A `GoTowards` controller picks its control velocity to get its value to the desired target. The value of the velocity must be related to the distance of the control from the target. There are several ways to choose the value. One is to pick a velocity that gets the control to its target in a set amount of time. The amount of time must be long enough to encompass a sufficient number of solver steps, and not too fast such that ODE solver inaccuracies cause the control to overshoot its target.

An alternate method of choosing the velocity for an `GoTowards` controller is to make it a scaled factor of the displacement. This creates spring-like attraction. When the control is far away it moves quickly, facilitating coarse positioning. As it gets close, it moves more slowly for precise positioning. One drawback of this technique is that it is hard to achieve target exactly: when the control is very close, its velocity is very low. One solution used in the prototypes is to switch between spring attraction and the constant rate scheme described above. When a control gets sufficiently close to its target, its velocity is chosen so that it would achieve the goal exactly in a step. This works well because for these small displacements, the approximations that the ODE solver makes may be sufficiently accurate.

An important attribute of `GoTowards` controllers is that they implicitly incorporate feedback. That is, they are continually adjusting the velocities to correct for any errors. If numerical drift or some other problem causes a control to make a turn for the worse, the error will be corrected in subsequent steps. Because of this, constraints will almost always use a `GoTowards` controller rather than a `Constant` controller with constant derivative 0.

An approximation to a `Follow` controller can be created with a `GoTowards` controller whose target is periodically updated. However, `Follow` controllers not only build in this continual update, but provide an opportunity to incorporate knowledge of the target's trajectory to improve tracking. A `Follow` controller effectively drives its value towards where its target will be, rather than where it is. In order to do this, the time derivative information of the target must be known. In practice, we rarely have good enough predictive models of input devices for this to work; however, it is useful for tracking key-framed motion paths, as described by [WW90].

6.4 Switching Controllers

The controllers we have discussed so far continuously perform the same simple task while plugged in. In this section we consider some more complex controllers that switch themselves on and off based on the values of the connector they are attached to and some measures of how hard they are working. In effect, these complex controllers simply plug and unplug more basic `GoTowards` controllers as needed. These behaviors are encapsulated as new controller types because: they are widely useful, so a simple encapsulation is handy; they can continuously watch the values, potentially switching more often than just at step boundaries; and they will look at internal solver quantities, the Lagrange multipliers, that we might prefer not to expose to system components outside of the solver.

By being clever about when controls are switched on and off based on the value of the connector they are attached to, a variety of interesting behaviors can be created. One behavior that this allows creating is an inequality constraint or boundary on a value. Switching of equality constraints is a standard technique for implementing inequality constraints, and is referred to as the *active set* method by the numerical analysis literature as it keeps a subset of all the constraints active at any given time. For the differential approach, we use similar techniques to create other behaviors as well. This section begins by describing three different variations of active set techniques, and then discuss some of the challenges in implementing them correctly.

6.4.1 Bounding

Placing boundaries or inequalities on aspect values is often useful, for example when a range of values is illegal, or only an approximate range is known. A `Bound` controller is created by enabling a `GoTowards` controller whenever the value has moved past the boundary. The `GoTowards` controller is used to move the value back within the boundary. When the output is within the legal region, the boundary constraint does nothing. The other major complication is that the boundary constraint should never pull the value further into the violated region, as depicted in Figure 6.5. That is, if another controller is trying to pull the value out of the illegal region, the boundary controller should not fight it. This case must be handled within the differential solver, as discussed in Section 6.4.4.

Inequality constraints operate by first being violated and using a `GoTowards` to fix themselves. If a value is moving towards a boundary, the `GoTowards` does not enable until after it has violated the boundary. One solution to this problem is that if such a violation is detected, back time up until the value is exactly at the boundary. This procedure is often done for physical simulation of collisions, as discussed in [Bar92b]. Such backing up could be used with the differential approach, and is discussed more fully in Section 6.4.5.

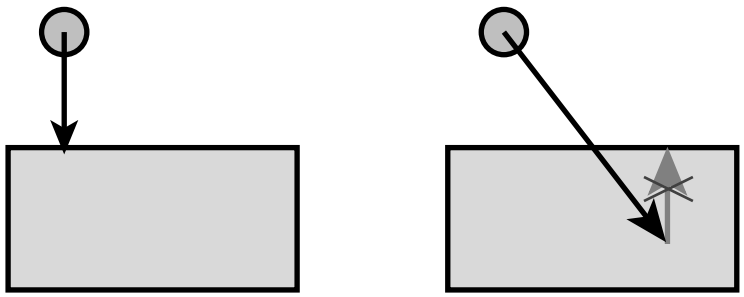

Figure 6.5: The point is bound to remain inside the rectangle. Left: when the point is in an illegal position a `GoTowards` controller pulls it back to the boundary of the legal region. Right: if some other controller will pull the point back into the legal region, a `GoTowards` controller that attempts to pull the point to the boundary of the legal region would pull against the other controller, and is therefore disabled.

6.4.2 Snapping

One of the fundamental issues in direct manipulation is precision: how can exact values be specified when they are required. This problem arises in many situations, for example when attempting to establish a precise geometric relationship in a drawing, when picking a small object, or when attempting to hold an object steadily in place with a noisy input device. One solution that has been employed to provide precision in direct manipulation is known gravity fields or snapping.

Gravity fields were first introduced in Sketchpad[Sut63]. Interesting points have a gravity field that draws the cursor in when it is close. The cursor follows the pointing device, however, when it is close to an interesting location it "snaps" precisely there. Many varieties of snapping have been used to help in direct manipulation. The most common are grids, which snap the cursor to points on a grid. Another important variety is object gravity, in which the cursor is drawn towards the graphical objects.

Just as we rely on tools such as straight-edges, rulers, and compasses to aid a pencil in drawing a precise drawing, gravity or snapping serves as a tool for creating precise graphical manipulations. Rather than having to manipulate objects to exact positions, the user can simply get them close, and the gravity takes the job of precisely positioning the cursor. This allows drawings to be created with more precision than the input device has, or to have precision even if the input device is noisy. Interface issues in snapping, such as providing feedback so the user can be confident that the correct relationship is being established and selecting among many close-together snap targets, can be handled, as shown in the *Briar* drawing program of Section 9.1.

One obvious way to incorporate snapping with differential manipulation is to use the snap target as the goal for dragging. This approach was used in the *Briar* drawing program, described in Section 9.1. One complication is that since the dragged object

does not follow the cursor exactly, the object may not establish the relationship that the cursor does. This can be combatted with feedback to inform the user when the dragged point establishes the relationship that the cursor has, and by making the snapping persist long enough to allow the dragged point to catch up with the cursor.

It is also possible to use the differential approach to implement snapping. A controller is connected to drive the output to a precise value when the output gets close. If another controller attempts to pull the connected attribute from its target, the snapping controller is disabled. An analogy for this is pushing a marble around on a grooved table. When the marble is close to a groove it falls into the groove, and will roll around inside of it until pushed hard enough that it escapes from the slot.

A `Snap` controller is implemented using a variant of the active set technique used for `Bound` controllers. By default, the controller is inactive. If the value of its attached connector gets within a specified distance of the snapping target, the controller is activated and switches on a `GoTowards` controller to drive the connector to the target value. If the magnitude of the Lagrange multiplier for the `GoTowards` ever exceeds a limit value, the controller is pulling too hard against other controllers and must be deactivated.

The key intuition behind `Snap` controllers is that the Lagrange multiplier is a measure of how hard a control is pulling. Since the "pull" of a controller is actually determined by the product of the Lagrange multiplier and the gradient of the control function, the magnitude of this vector is actually used to determine the amount of effort the controller is applying. One difficulty with `Snap` controllers is that the parameters that determine their behavior, most specifically the limit magnitude, must be determined empirically.

A `Snap` controller can be attached to any object output, not just positions. For instance, a `Snap` controller on a line segment's orientation could make a segment that was easy to place into a precisely horizontal configuration. The differential implementation provides more flexibility than traditional methods for implementing snapping. We therefore call the differential snapping *generalized snapping*. In Section 8.4.1 we discuss some applications of generalized snapping.

6.4.3 Click Stops

Another way to combat the precision problem with direct manipulation is to require a value to have one of a discrete set of values, for example to create a grid for dragging. This is easy to do with the traditional implementations of direct manipulation, which often use integer representations anyway. However, the differential approach operates on continuous values. One way to implement an interaction that limits a value to a discrete set is *clicking*.

Clicking is a variant of snapping that effectively constrains a connector output to have a discrete value, either from a finite set, or to be an integer or multiple. `Click`

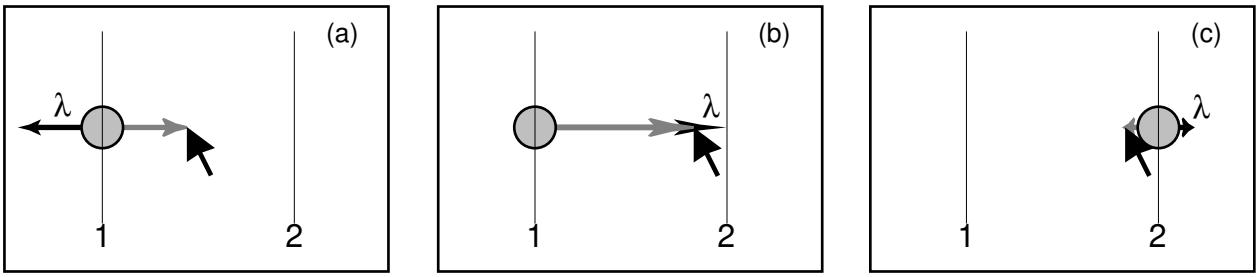

Figure 6.6: Clicking forces a value to be in a discrete set. The value, in this case the horizontal position of the circle, is constrained to have the value 1 or 2. Grey arrows represent the pull of the mouse control, black arrows represent the pull of the `Click` controller. Initially (a) it has value 1, so a `GoTowards` controller pulls to maintain this value as the mouse attempts to drag it with a pull labeled $\boldsymbol{\lambda}$. At some point, the mouse pulls hard enough so that the pull vector's magnitude exceeds a limit value causing the controller to advance to the next click stop (b), causing the point to attain the next value in the set (c).

controllers work by constraining the output to have a value in the set. If the constraint is pulling too hard, the value is changed to another element in the set, as illustrated in Figure 6.6.

6.4.4 Implementing Active Set Methods

The `Bound`, `Snap`, and `Click` controllers all operate by switching simpler `GoTowards` controllers on and off. They are all variants of the active set methods commonly used for realizing inequality constraints. Active set algorithms are detailed in standard textbooks such as [GMW81] and [Fle87]. The difficult part of such algorithms is determining which constraints to enable and disable, a combinatorial problem. The simple method discussed here tries a few guesses at the correct set, and then gives up, selecting a solution that fails to satisfy all the requirements. The solver will be likely to provide an acceptable interactive behavior. We will, therefore, begin by examining the active set problem in the context of the differential approach, and then described a simple algorithm. While the simple technique does guarantee the correct answers to the problems, it has the following advantages:

- The methods always fail in ways that can be corrected later.
- The methods are easy to implement as extensions to the existing differential optimization.
- The methods do not access any of the matrices, except by calling the differential optimization. Therefore, as with the differential optimization techniques, we can use any linear system solver.

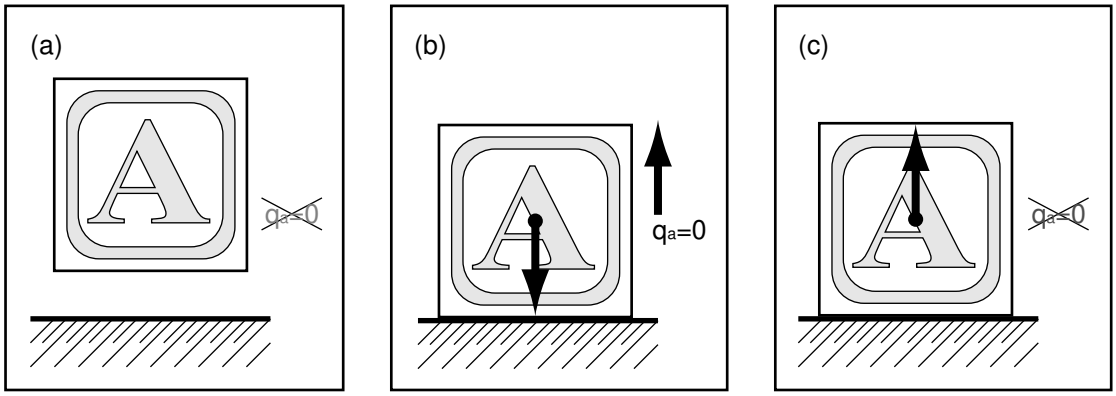

Figure 6.7: An inequality constraint $q_a > 0$ keeps the block above the floor. The inequality is implemented by switching an equality constraint $q_a = 0$ on and off. When the block is above the floor (a), the equality constraint is inactive. If the block is on or under the floor (b), the constraint pushes the block towards being on the floor, counteracting any controls that are pulling the block downwards. However, if other controls attempts to pull the block above the floor (c), the control used for the inequality would pull downward, causing sticking. Therefore, it is deactivated.

Recently, an inequality solver specifically designed for interactive systems was developed by David Baraff [Bar94]. The algorithm was designed for implementing physical simulations of collisions, a problem very similar to implementing graphical manipulation with the differential approach. Baraff's algorithm could be used in computing the differential optimization. Application of this algorithm to implementing the differential approach is left as a topic for future study (Section 10.3). However, without significant work, such algorithms do not have many of the advantages of the iterative solvers discussed in Section 4.3.2, such as the ability to exploit sparsity.

Active Set Methods in the Differential Approach

To introduce the basic idea of an active set method, consider the simple example of Figure 6.7. The example has a single state variable, q_a that measures the block's height above the floor. An inequality constraint is used to keep the block above the floor, $q_a > 0$. An active set method implements this inequality by switching the equality constraint $q_a = 0$, or in differential terms q_a `GoTowards` 0, on and off as needed.

If the block is above the floor, the inequality constraint does nothing. It is *inactive.* The more interesting case is when the block either sits on the floor, or has fallen below the floor. In this case, the `GoTowards` controller is activated to either push the block towards sitting exactly on the floor, or to prevent the block from sinking underneath

the floor. When the constraint works to keep the block above the floor, its Lagrange multiplier will be positive, that is, the constraint is pushing upwards.

Consider a case where the block is sitting on the floor, with the constraint activated. Suppose another control, such as connection to the mouse, attempts to pull the block off the floor. The `GoTowards` controller will attempt to keep the block on the floor, with a negative Lagrange multiplier pulling downwards. In essence, the inequality constraint will be sticky. The solution to this is to disallow negative Lagrange multipliers. When one is found, the constraint must be deactivated.

When there are multiple inequality constraints coupling objects, deciding which Lagrange multipliers to deactivate becomes more complicated. Consider the example of Figure 6.8. Two keep the two blocks stacked, one inequality constraint keeps block B above the ground, while the other keeps block A above block B. The former constraint is implemented by switching the equality $q_b = 0$ on and off, the latter by switching $q_b - q_a = h$. For each solution of the Lagrange multipliers, the solver must determine which of the two constraints should be activated. If a control is used to pull block A upwards and both constraints are active, the equality constraints would maintain the stacked configuration, causing the blocks to stick in place by pulling downward on them. Since the inqualities can only push upwards, we might deactivate the ones pulling downwards (e.g. both of them). However, deactivating all of the constraints pulling in illegal directions is unnecessary, and wrong. If there is a soft control pulling B downwards, that constraint should stay active.

The process of determining the active set is iterative: the solver must determine which ways the constraints are pulling, deactivate any constraints pulling in illegal directions, reactivate any constraints that should not have been disabled, and repeat. Determining the correct active set can be difficult, and might require many attempts.

If the correct active set is not found, there are two types of errors possible for any particular constraint. Either a constraint that should be active is deactivated, or a constraint that should be inactive is activated. This latter case is extremely problematic. Consider what would happen in the example of Figure 6.7: the user would not be able to lift the block off the floor. If the algorithm made this error in one step, it would make the same error in subsequent steps since the configuration does not change[1].

Deactivating too many constraints is less of a concern. Consider the example of Figure 6.8. If both constraints are deactivated, the lower block will not be prevented from moving downward for the step. However, in the next step, the algorithm could correct for this error. It is unlikely to make an error activating the constraint because the block will be separated from the upper block, which is what caused the confusion.

If we resign ourselves to not always being able to find the correct active set, we must at least make sure to always err on the side of deactivating too many constraints. Such an algorithm is simple to devise. An example algorithm is presented in the next

[1] Assuming the algorithm is deterministic and it bases decisions solely on the configuration, which the differential methods do.

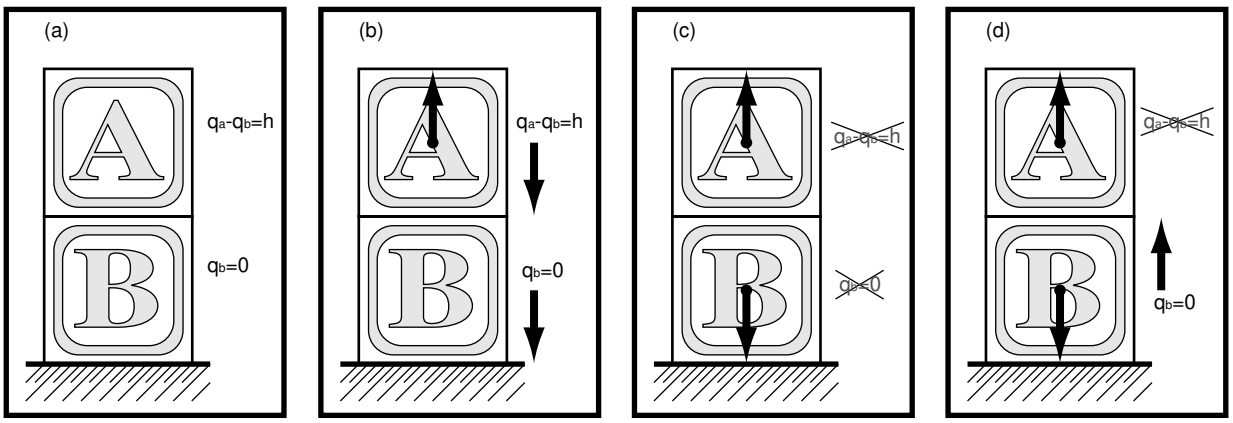

Figure 6.8: Multiple blocks are kept stacked on the floor by inequality constraints. (a) One constraint keeps block B above the floor, the other keeps block A above block B. (b) If a control pulls upwards on block A, the active constraints impose equalities that cause the blocks to stick in place. (c) Removing all inequalities that are pulling downward in (b) would cause the constraint on block B to be erroneously removed. If another control were pulling down on it, no constraint would keep B above the table. (d) The correct solution reenables constraint B.

section.

It is important to notice that excess deactivations are only a problem when there are many interacting inequality constraints. There are interesting applications, such as collision simulation, where this occurs. In such applications, these errors might be problematic. However, there are other potentially more significant sources of errors in such applications, such as improperly handling the piecewise continuous nature of the ODEs, as discussed later in this section. In Section 8.4.2, we will look at using the differential approach for collisions, and consider the problems.

An Active Set Algorithm

The basic outline of a simple active set technique is as follows:

1. Select an initial active set by activated any inequality constraints that may be active. Candidate constraints are those whose values either violate their boundaries or are within a tolerance of the boundaries.

2. Select some deactivated constraints that are potentially active and reactivate them.

3. Solve for the Lagrange multipliers.

4. Deactivate any constraints with negative Lagrange multipliers.

5. If the active set was changed, return to step 2.

The simplest method omits step 2. This leads to a method that meets the requirement that it will always err on the side of having too few active constraints. It also is guaranteed to terminate because on each iteration the size of the active set must decrease, since there is no step inside the iteration that adds constraints. It has the disadvantage that it often removes too many constraints.

A constraint is a candidate for reactivation in step 2 if the derivative of the function is negative. A constraint $f_i(\mathbf{q}) > k$ is a candidate for reactivation if $\nabla f_i \cdot \dot{\mathbf{q}} < 0$. Implementing reactivation can be complicated. The method must not to cycle, that is to infinitely loop among several active sets. A method that I have used to prevent looping is to permit a given constraint to be reactivated at most once. While this falls far short of a reliable inequality solver, it gets correct solutions on many problems on which the simpler method fails, for example the problem of Figure 6.8.

6.4.5 Piecewise Continuous ODE Solving

Solving the ODE is difficult because the derivatives are continually changing. In Section 3.3, we were concerned about them changing due to the functions being non-linear. However, the derivatives might also change discontinuously when the set of controls change. Such ODEs are called *piecewise continuous* as they consist of continuous pieces between breaks. Note that the discontinuities are in the derivatives, not in the values. Methods for solving piecewise continuous ODEs are discussed in an appendix of [Bar92c].

An ODE solving technique such as Runge-Kutta attempts to fit a continuous function (a polynomial) to the ODE over a step. If the ODE is discontinuous, this may be problematic. Even if multiple smaller steps are taken to fit the function with piecewise polynomial segments, methods that take fixed step sizes are not likely to have their discontinuities match the discontinuities in the derivatives.

Many of the discontinuities are avoided in the differential approach, since events that change the set of controls are deferred until the ends of steps. The ODEs are therefore step-piecewise continuous, or continuous within the steps. This means that techniques such as Runge-Kutta are acceptable, because they will not see the discontinuities. Techniques that require continuity between steps, such as Predictor Corrector methods, may have more problems.

Active set methods may change the set of controls during a step, causing a discontinuity in the derivatives. This is potentially problematic. However, the alternative would involve finding the precise time that the controls switch and adapting the step sizes so that the step boundaries occur exactly at this instant.

The most severe symptom of not finding discontinuities is that the system can respond to changes only after they happen. For example, an inequality constraint activates after it is violated. Therefore, there will be a brief instant when the constraint is violated. Subsequent steps will move to a it valid state. The ill-effects of an ODE solver misfitting a continuous polynomial will similarly be repaired by subsequent steps. If such error is unacceptable, cleanup steps can be used.

Piecewise ODE solving by backing up can be used with the differential approach. I have instead used the approach of letting later steps correct for any errors caused by violations. There are many reasons to prefer such an approach:

- The general root finding problem of determining when to back up to is difficult.
- The ill-effects of adaptive step sizes, as described in Section 3.3.1 are applicable - possibly even more so as adjusting the step size requires a potentially time consuming root finding operation.
- When there are many changes close together in time, a backing-up system can take only tiny steps between them. In a cleanup afterwards approach, many can be cleaned up simultaneously.
- With the cleanup approach, objects not affected by the discontinuity continue to move as they otherwise would. If the step size were adapted, all of the objects would slow down whenever any discontinuities occurred.

For certain applications, such as collision simulation, having instants of violated inequalities will be unacceptable, and piecewise ODE methods will be required.

But the ancient debate on emergence, whether indeed wholes may have properties not intrinsic to the parts, is besides the point. The fact is, that parts have properties that are characteristic of them only as they are parts of wholes; the properties come into existence in the interaction that makes the whole.

— Richard Levins and Richard Lewontin
The Dialectical Biologist, p 273

Chapter 7

A Graphics Toolkit

To this point, we have introduced the machinery of the differential approach, the methods for its realization, and the structure of a general purpose implementation. In this chapter, we consider how the approach can be encapsulated to support the construction of graphical editing applications. Graphical applications typically share a wide variety of functionality, so their construction is often facilitated by the creation of toolkits that encapsulate the common needs. This chapter discusses such a toolkit built on top of the differential approach. *Bramble* is an object-oriented graphics toolkit, built on the infrastructure of the previous chapters.

Bramble has much in common with other object-oriented graphics toolkits designed to support direct manipulation graphical editors, such as Garnet [MGD+90], Inventor [SC92], GROOP [KW93], and Alice [PT94]. Bramble's primary distinction is that it is designed on top of the differential approach. The major consequences of this are:

- The differential approach is used to create almost all graphical manipulations.
- Graphical objects have connectors that compute their various attributes, and often provide these standardized connectors in lieu of specific interaction code.
- Snap-Together Mathematics is used to connect objects together and to connect interactive controls to objects.
- Bramble provides support for application features such as geometric constraints that are easy to create with the differential approach, but difficult to implement in standard toolkits.
- Bramble's control flow is dictated by the differential approach. The model of an ODE solver with events interleaved between steps, as described in Section 6.2,

provides a centralized main loop that calls application code when needed. The differential approach allows continuous motion direct manipulation to fit nicely into such a scheme. callback style architecture as mechanisms exist that permit direct manipulation to fit in such a scheme.

Bramble not only shows how the machinery of the differential approach can be applied to the construction of graphical applications, but also how the approach's machinery can be encapsulated and hidden from the applications programmer. My goals in constructing Bramble were:

- To make it possible to quickly prototype a number of applications to illustrate and explore the differential approach. The emphasis is on speed of construction and extensibility, rather than on industrial strength tools. Much of this goal was pragmatic: I needed to construct enough examples to support the conjectures of this thesis in a reasonable amount of time.

- To support a variety of applications. It was important that Bramble could support both 2D and 3D applications. While the focus is on graphical editors and modelers, Bramble also supports other applications such as object viewers and visualization tools.

- To support the basic services of graphical applications typically provided by toolkits.

- To facilitate building applications that provide the features that the differential approach supports, such as geometric constraints.

- To facilitate experimenting with interaction techniques and evaluating them in context within applications.

- To show how the architectural features of the differential approach could impact applications architecture. In particular, the approach can enable increased modularity, since Snap-Together Math provides a common connection mechanism between parts, and separation of manipulation and representation aids encapsulation.

- To show that the machinery of the approach can be sufficiently encapsulated. The application programmer does not need to see the mathematics in order to make use of the approach's features. Bramble is designed to let developers program with the familiar abstractions of graphical applications. They see the abstractions of the approach only when defining new interaction techniques. Bramble programmers never need see the inner workings of the solver.

7.1 The Bramble Application Model

The flow of control in Bramble is dictated by the differential approach, as described in Section 6.2. Time flows forward as the ODE solver steps forward, continuously evolving the state of the objects. By continuously viewing the world as it evolves, we can see objects move according to their controls. At discrete instants, such as when an event occurs, changes to the objects in the world can be made. However, such events are instantaneous impulses: all changes to the configurations of objects must happen via the passage of time by the ODE solver.

The differential approach's model of time is significant in toolkit design for two related reasons:

1. It provides a uniform "main loop" for all applications.

2. It provides a way to incorporate simultaneous, continuous actions such as dragging into an event driven architecture. Events can asynchronously begin and end continuous motion actions, concurrency is handled by the solver: there is no need for multi-threading or other time sharing mechanisms to achieve concurrent, asynchronous actions.

These two elements make it practical for Bramble to use event callbacks as its sole application control mechanism. A Bramble application does not contain a "main" loop, but rather, after the application concludes its initial setup, it simply call a function defining a standard loop that runs the ODE solver, keeps the views of the world up to date, and calls appropriate fragments of application code when needed. Code to be called is specified in hooks, variables that can be bound to fragments of code that are accessed at defined times. Hooks are discussed in more detail in Section 7.6.

The overall structure of Bramble is shown in the schematic of Figure 7.1. Bramble is built on top of the Silicon Graphics GL graphics library [Sil91] and the Snap-Together Math toolkit, discussed in Chapter 5. Bramble itself contains no solver code and does not even provide access to the solver's internal structures. Bramble also includes the Whisper embedded interpreter, described in Appendix A. The symbiosis of Bramble and Whisper will be discussed in Section 7.1.1.

A Bramble application consists of two parts:

- Code implementing any custom classes.

- A program that is run to execute the application. This program typically sets up the application, doing tasks like opening windows, creating initial objects and widgets, and defining hooks. After initialization, the program starts the differential process' time flowing. Once time is started, application code is only executed when a hook is called.

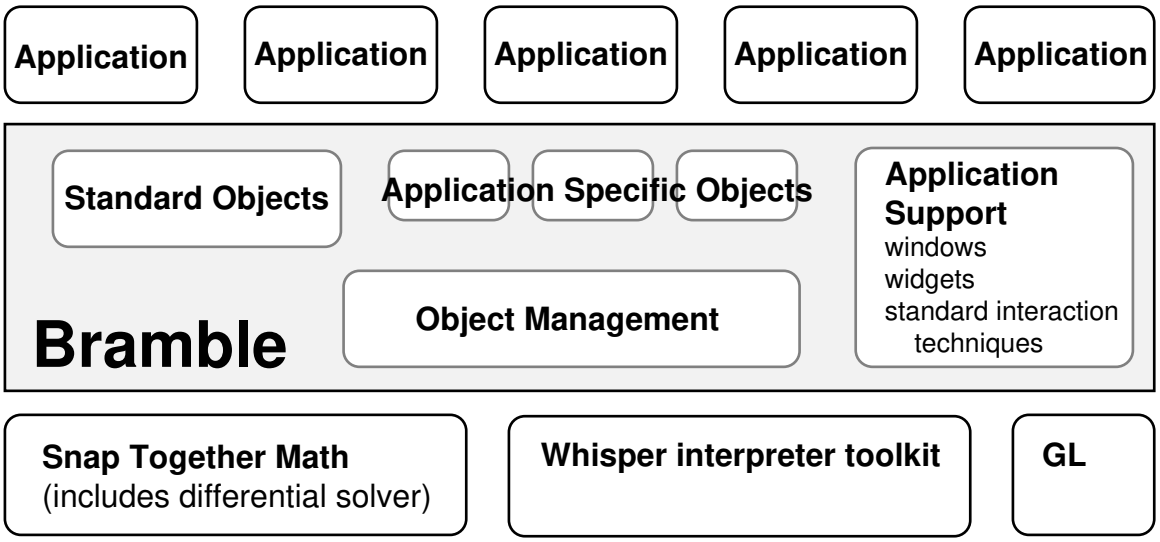

Figure 7.1: The pieces of the Bramble toolkit. Bramble is built on top of Snap-Together Math and the Whisper interpreter. It contains a variety of pre-defined object classes, support for managing sets of graphical objects, and other miscellaneous pieces needed by graphical applications. A Bramble application typically consists of a Whisper driver program and (optionally) C++ definitions of new object types.

The major pieces of Bramble are akin to other similar toolkits. Each is affected by the use of the differential approach:

Object Management – Bramble's mechanisms for managing sets of objects, discussed in Section 7.3, provide access to the features relevant to the differential approach, such as keeping track of state variables and connectors. All objects, even windows and widgets, can have state and connectors.

Standard Object Types – Bramble's standard object types, discussed in Section 7.5 provide a wide range of connectors to support graphical manipulation.

Hooks – Bramble provides a set of hooks that aim to be sufficient to specify a range of interaction techniques. These are chosen to provide a convenient set of opportunities to alter the controllers and objects of the differential approach. Bramble's hooks are discussed in Section 7.6.

Windows and Widgets – Bramble's windows can contain views of the world that are automatically kept consistent, and many of the supported widgets, such as sliders, are designed to manipulate their targets using the differential approach. These features are described in Section 7.7.

Standard Interaction Techniques – Bramble contains many basic graphical manipulation techniques in its library, all of which are defined with the differential approach. Picking and snapping services are tuned towards the selection of connectors for manipulation.

This chapter will discuss these various pieces of Bramble, following a discussion of an important part of Bramble's structure, and a simple example of how Bramble is used.

7.1.1 Whisper and Bramble

The Whisper embedded interpreter, described in Appendix A, is an important part of Bramble. An embedded interpreter is a useful feature for graphical applications as it provides support for features such as saving and loading user data files and user extensions. However, Whisper plays an even more significant role in Bramble. Internally, the Whisper interpreter's implementation is used by Bramble for support, and externally, much of the application programming in Bramble is done in Whisper.

As discussed in Section 2.3, using an embedded interpreter in a graphics toolkit is a common technique. However, the differential approach removes some of the drawbacks. The speed-critical computations occur as part of the mathematics; all standard pieces can be written in the compiled host language. The centralized main loop provided by the ODE solver also interacts nicely with the embedded interpreter. Most application behavior is defined by hooks. Bramble's hook mechanism, discussed in Section 7.6, permits pointers to C++ functions, text fragments of Whisper code, and Whisper closures to be dynamically assigned.

Like other similar languages, Whisper has a runtime support system for memory and object management. Bramble also uses these facilities for its needs. The advantages are twofold: first, it provides Bramble with a dynamic, memory managed object system; and second, it means that Bramble's objects are easily accessible from the interpreter.

The majority of Bramble application programming is done in Whisper. Bramble is designed such that applications are constructed by extending the generic default application. Constructing applications in the extension language is, therefore, sensible. Application programming in Whisper has many advantages:

- The interpretive nature of Whisper leads to faster turnaround than the statically compiled C++ environment. [1]
- The interactive loop of the interpreter provides a useful debugging environment, whereas the complexity of the entire C++ system makes using the standard C++ debuggers awkward.

[1] This is more a statement about the C++ programming environment at the present time than about Bramble.

- Whisper's abstraction mechanisms, notably first-class functions and dynamic objects, are useful in the design of interactive systems.
- Many of the details of the system are hidden from a Whisper programmer. For example, window management and refresh, and the solver internals are hidden. In fact, there are no mechanisms provided in Whisper to access internal solver data structures. The language does not even have vector or matrix constructs!

There is nothing that *must* be done in Whisper. However, almost all application programming *can* be done in whisper.

Bramble is actually wired into the Whisper interpreter as an extension. The C++ "main" of a Bramble application is a Whisper read–eval–print loop, typically that can select files to read from the command line. For my work, there is a single Whisper/Bramble executable and each application differs only by the Whisper program used to run the application. For many applications, however, a different version of the interpreter, with custom sets of extensions wired in, would be more desirable.

The interpreter-based organization of Bramble emphasizes that it meets its goals of insulating the application from the mathematics. The differential optimization is completely encapsulated inside the `ConstEngine` object as described in Section 5.4.5. Nowhere in the Bramble toolkit source code or applications is there mention of the internals of the solver process.

7.2 A Simple Example

In order to introduce the basic concepts of Bramble, a simple example will be presented. This example is designed to be similar to that used to introduce the Inventor toolkit. In the Inventor book [Wer94], a program called "hello cone" is given as a first introduction to Inventor. Here is the same program, one that displays a cone in a window, written with Bramble:

```
(1) (set my-view (make-view "Hello Cone"))
(2) (set my-cam (make-la-camera))
(3) (view-cam my-view my-cam)

(4) (set c (make-cone))
(5) (c (set material shiny-red-material))

(6) (go)
```

This simple example brings out the basic notions of Bramble. A program places graphical objects in the world, and does not worry about lower level details such as how they are drawn or how the windows are managed. The first line creates a window that

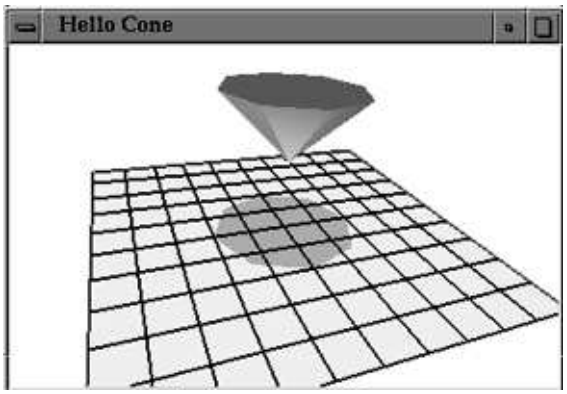

Figure 7.2: The simple "Hello Cone" program in Bramble.

views Bramble's world. Line 2 creates a default "lookat" camera, and line 3 specifies that the view created on line 1 should look at the world through the lens of this camera.

Lines 4 and 5 of the program create the cone object, and changes its color to red. The color is changed by setting the property of the cone object that specifies the surface properties to be a predefined material that is shiny red. This unusual syntax for accessing object fields is described in Appendix A. Notice that when the cone is created, it is by default created in the world and is therefore visible to viewers of the world.

The "action" of the program takes place in the last line. The `go` routine begins the interactive loop which is in the toolkit proper. In differential terms, time is started. Steps are continually taken, and in between any events are handled. The interactive loop runs continuously until either the program is killed, or a flag is set.

The simple 3D example receives a lot of a 3D interface from Bramble's defaults, as shown in Figure 7.2. By default, Bramble creates a number of interface elements such as the ground plane, some lights, and the shadows dropped vertically onto the ground as if the sun were at high noon. These defaults can easily be overridden. In a sense, the Bramble program simply modifies the default application and runs it.

The most obvious difference between this program and the Inventor equivalent is that it is written in Whisper, not C++. This difference can be emphasized by adding the line

```
(add-key dev-esckey k-any k-down
    (lambda (v) (set bramble-going nil)))
```

which attaches a procedure that sets a flag to the escape key. This flag signals the `go` routine to stop the interactive loop and return. Since there is no code after the call to `go`, Bramble will return to an interactive Whisper prompt. This allows a user to interactively alter the program. The differential interactive loop can be restarted with another call to the `go` routine.

To get elements of a standard 3D interface, some standard key definitions can be read by replacing lines 1 through 3 with

```
(read std-world.wh)
(set v (make-standard-view "Hello Cone"))
```

at the top of the file. The Bramble standard 3D interface, described in Section 7.8 binds the mouse buttons to routines that permit objects to be grabbed and dragged and the viewpoint to be controlled.

With the 3D interface, we would notice that when we grab the cone in the simple program, it does not move. The primitive cone from the library is a rigid body that has no degrees of freedom. To make it movable, we place the cone in a transformation group with the lines

```
(set g (make-qs-group))
(add-to-group g c)
```

somewhere after the cone is created, but before `go.` The `make-qs-group` function creates an empty group that can rotate and scale (qs stands for quaternion and scale).

We might want to have the cone begin pointing upwards. Since this will require a non-differential change (i.e. we want it to instantaneously appear in the right orientation), we cannot use the differential machinery. This means we must actually directly access and modify the state variable of the object. This is done by

```
(set-qvar (get-q g) 'sy -1)
```

that sets the `sy` (scale in y) variable of the group to -1. We might then want to prevent the object from scaling as it is manipulated, which can be accomplished by

```
(freeze-scale g)
```

which is a special operation for groups.

The cone can now be grabbed and dragged using the standard Bramble mousepole manipulation technique, as described in Section 7.8. We will now add some other functionality that uses Bramble to operate the differential machinery. Suppose that we want to have the top of the cone fly towards the left. This is accomplished by taking a connector that computes the position of the top of the cone and attaching a controller to it. Since the cone object has a predefined "top" connector, this operation is as easy as

```
(controller (signal (c top) 'x) '= 5)
```

that finds the x coordinate output of the top connector of the cone and creates a con-

troller that drives it towards the value 5. We might want to instead bind this operation to a key so it happens only when we're ready

```
(add-key dev-gkey k-none k-down
    (lambda (v) (controller (signal (c top) 'x) '= 5))).
```

Notice how this example maps exactly to the abstractions of the differential approach. We manipulate an object by attaching controllers to its connectors. The actual change itself happens as time passes after the discrete instant when the controller is created. The manipulation is done completely in terms of the connectors of the objects, without regard for what parameters inside make them actually change.

Suppose we want to make a chain of cones. Here, we add a second cone and connect it to the first.

```
(set c2 (make-cone))
(set g2 (make-q-group))
(add-to-group g2 c2)
(pt-eq (c bottom) (c2 top))
```

The important thing to notice from this example is that we were able to create the connection by talking in terms of geometric or graphical objects, not mathematics. The constraint is created to connect the bottom of one cone and the top of the other.

The lack of mathematics in the last example is a result of us being fortunate that the cones had the connectors we desired. However, adding a new connector is easy enough. Suppose we wanted to add a new "horizontal" connector to the cone that measured if the cone was horizontal (i.e. that the y coordinate of its top and bottom are equal, as Bramble's coordinate systems defines the y axis to mean height). This can be done as

```
(c (bind horiz (minus-block (signal top 'y)
                            (signal bottom 'y))))
```

This code fragment uses the Whisper/Snap-Together Mathematics interface to create a Snap-Together Mathematics output that computes a new attribute called `horiz`. The attribute is computed by subtracting the height of the bottom of the cone from the top. A function block is used to perform the subtraction. Whisper scoping rules cause the code fragment that creates the block to be executed in the cone's environment, so that `top` and `bottom` refer to the cone. The code fragment stores stores the function block as a field in the cone object's environment permitting other objects to access it to use it as a connector. This code fragment could be written as a procedure that could be applied to other cones

```
(define add-horiz (a-cone)
 (a-cone (bind horiz (minus-block (signal top 'y)
                                  (signal bottom 'y)))))
```

These connectors could be used just as any others, for example

```
(add-horiz c2)
(controller (c2 horiz) '= 0)
```

7.3 Bramble's World

The focus of Bramble is on building object-oriented graphical applications. One of Bramble's central roles is to help maintain the set of objects that are being presented to the user. In Bramble, there is a single, implicit "world" in which all objects exist. By default, when objects are created they are placed in the world. The concept of the world is implicit in Bramble, unlike Inventor which requires explicit creation of "scene graphs." The advantage of this is simplicity, although the ability to have multiple simultaneous worlds is sacrificed.

The world provides a uniform 3D coordinate system for all objects. The world is always 3D, even for 2D applications. In such applications, objects ignore the third dimension. This has a small cost, for example in doing transformations. The benefits include avoiding redundant code and the ability to place 2D objects in 3D worlds. The underlying GL graphics toolkit takes a similar approach.

In Bramble, the programming model is not of screens that are continuously redrawn. Instead, the model is of a time continuous world into which objects are placed and manipulated. Images on the screen provide views of this world.

7.3.1 The Bramble Object System

The mechanisms that support state variables, connectors, and other object management are implemented at a general level. This permits all major object types in Bramble, including windows and widgets, to support features that are usually associated with graphical objects, such as having state and connectors.

All major types of objects in Bramble are subclasses of the type `IDObj`. Having a common base allows common functionality to be shared among all types of objects. For example, each object has the ability to be named, and is assigned a unique ID number that serves as a soft pointer (hence the name `IDObj`). Each `IDObj` has a field that allows its major subtype to be determined dynamically. An object manager keeps track of all `IDObj`s created. The major subclasses of `IDObj` are:

`Drawable` – a graphical object in world space. This includes not only the user created scene objects, but lights, cameras, hierarchy elements, and world space interface elements such as 3D widgets.

`IDrawable` – a special object that exists in screen space, rather than in world space. Effectively, these objects are attached to lenses of cameras.

`View` – an object representing a view of the world on the screen. It is basically a pairing of a camera and a window, with added responsibilities for determining drawing parameters.

`DistinguishedPoint` – an important type of connector that represents a specific point on some object.

`Imagepoint` – a special type of connector that represents the position on the screen of a point.

`SubWin` – a subwindow of a screen window. Subclasses include widgets like buttons and sliders.

`Frame` – the "physical" GL window on the screen. May contain several `SubWins`, one of which may be a `View`.

`Material` – a set of surface properties.

`Demon` – a special object that performs its hooks when triggering events occur.

One of the key ideas behind Bramble's object management is that each IDObj contains a Whisper environment. In Whisper, environments serve as a dynamic object system (see Appendix A). Using Whisper environments as an object system for C++ has several advantages:

- It provides easy access to the object from Whisper.
- By defining methods as values in the Whisper object, they can be dynamically altered.
- Fields and methods can be added and removed dynamically as needed.
- Objects can inquire about which fields and methods other objects contain.
- Whisper environments, to a certain extent, are automatically memory-managed.

The dynamic nature of objects is important: fields and methods may be added in response to user actions. It also allows experimentation with application behavior: objects can be interactively modified while the application is running.

Any `IDObj` can have state variables and connectors. The base class handles management of these differential features. It also permits objects to create differential controllers or employ function blocks. These are automatically removed when the object is deleted. Object deletion is also handled by the base class using lists of hooks. This permits an object to define work that must be done when the object is removed. Dynamic definition of deleters, very much in contrast to the static C++ notion of a deleter method, is a very useful feature.

Memory management can be a difficult task in an interactive system. Bramble uses a variety of manual techniques to automatically manage memory without using a garbage collector. Bramble contains mechanisms to deal with the inverse garbage collection problem. An inverse garbage collector activates when an object must be deleted, for example in response to a user command, and insures other objects that refer to it are appropriately altered or deleted. A garbage collector does not provide this behavior. In fact, in a system that was simply garbage collected, the object would not be deleted until all references to it were removed. Interconnection of objects is pervasive in and central to the differential approach, so handling inverse garbage collection is important in Bramble. The mechanism used to handle this problem is to have each object maintain lists of other objects that should be notified or deleted when the object is deleted.

7.3.2 Graphical Objects

The graphical objects in Bramble's world are derived from the class `Drawable`. This class requires only a few methods. Subclasses of `Drawable` define a draw method that needs only to operate in the object's local coordinate frame. Objects may also provide other functionality. For example, most objects will define at least a few connectors so they can be manipulated, and will allocate a state vector to store their configuration. Other examples of optional `Drawable` functionality include ray intersection, bounding box computation and generation of external representations. Almost all methods apply in local coordinates. Hierarchy mechanisms perform required conversions automatically.

Bramble keeps all of the instances of `Drawable` on a list. The list of "the stuff" is Bramble's notion of a scene. Only one list is kept by the program. Selective drawing can effectively provide multiple scenes. Although the list is a flat representation, hierarchies are supported by the grouping mechanism, described in Section 7.5.3.

The graphical objects are not unlike their counterparts in other object-oriented graphics toolkits. One difference in Bramble is that each object class does not have to provide methods for interaction. An object need only provide connectors that the more general manipulation techniques can connect to. "Object-centric" styles of interaction are possible with Bramble. For example, an application can have each object define methods which are called by a dispatcher that receives events. Intermediate styles, for example

having an object use hooks to alter general manipulation techniques, are also possible.

Another distinction of Bramble's graphical objects is that they must also manage state and connectors in ways that permit the differential approach to manipulate the object. Bramble's object system provides a uniform place for objects to keep a state vector. Once a graphical object registers that it has a state vector, Bramble automatically insures that the variables are managed correctly.

Connectors are handled in a less uniform manner. In Bramble, a connector is simply a Snap-Together Mathematics `Port` that an object stores in a way that other objects can gain access to it. Standard mechanisms exist for certain important types of connectors, as discussed in Section 7.4.

7.4 Connectors in Bramble

In Bramble, a connector is simply a Snap-Together Mathematics `Port` that an object exposes in a way that other interested objects can gain access to it. The most common way to do this is for the object to place the `Port` in a field of its kept environment. Objects are free to create any types of connectors they want. Objects often provide special purpose connectors tuned to compute their specific types of attributes. Many specific examples are given in Section 8.1.

The most important type of connector in Bramble is the *distinguished point.* A `DistinguishedPoint` represents a particular point on a graphical object. The primary output of a `DistinguishedPoint` object is its position in space, but instances usually provide other attributes, such as surface normals and tangents, when they are known.

Distinguished points are first class Bramble objects unto themselves, that also happen to be `Port` objects as well. By being a real `IDObj` a distinguished point can:

- be accessed by standard object naming and reference techniques;
- store state, permitting the point to be moved around on the object;
- define hooks to specify its behavior;
- have other connectors associated with it, for example, to compute the position of its shadow on the floor or wall.

To standardize access, each object keeps a list of its distinguished points. Similarly, a global list of all distinguished points is kept to aid in such tasks as picking and snapping. Registry in this global list is automatically handled by the `DistinguishedPoint` creation process, and permits points to be found by name or by location.

The position and orientation of a distinguished point are not simply a function of the graphical object the point is on, but also the transformations applied to the object.

However, the hierarchy mechanisms automatically handle these transformations. A point must provide only methods for computing its connectors in the object's local coordinate frame. Jacobians are also composed for points in hierarchies.

Standard subclasses of `DistinguishedPoint` include:

Fixed Points that are a fixed position in the object's local coordinate system.

Point on Point that connect directly to a set of object variables that represent a position.

Free Points that store their position in the object's local coordinate space as their own state vector. Constraints are often used to keep such points in the volume of or on the surface of an object.

Different types of objects may also define special types of `DistinguishedPoint`. For example, a parametric surface might define a type of point that stores its parameter values as state. Such a point can slide along the surface, or the parameters can be frozen to create a fixed point.

7.5 Graphical Objects

Bramble supports a wide range of graphical objects. Many standard subtypes of `Drawable` are provided, and new application-specific types can be defined.

7.5.1 Standard Object Types

Bramble predefines a variety of basic object types, along with standard connectors with which to manipulate them. For example, the 2D set includes lines, circles, rectangles, ellipses, and polygons. A general parametric curve class allows the definition of new object types by simply providing the parametric function. Connectors and a drawing function are defined automatically from this as described in Section 8.1.1.

Most of the standard 3D objects are defined as rigid bodies that must be placed inside of groups to be moved. Most standard shapes, such as cones, tori, cylinders, and spheres are provided. Polyhedra can be defined in several ways, including readers for several data file formats.

A planar mirror is provided as a 3D rigid object. In terms of its geometry, the mirror is simply a rigid square. However, the mirror's draw function calls other objects' draw function with a new transformation that draws the object on the mirror's surface to simulate reflection. Mirrors define special connectors that permit the reflections to be directly manipulated.

Geometric constraints are represented by `Drawable` objects which create associated controllers on some of their connectors. For example, a connection constraint

would take two `DistinguishedPoint` position connectors, and compute the difference as a connector. When created, it would also create a `GoTowards` controller on the distance connector to maintain the constraint. The objects that represent constraints may provide drawing methods that provide feedback to the user. Bramble includes many basic constraint objects in two and three dimensions, including point connection, distance, collinearity, normal or tangent alignment, and parallel. Inequality constraints in the basic set include constraints to keep points inside many of the basic shapes. All of these constraints operate generically on object connectors so they can be attached to many types of objects.

7.5.2 Defining New Object Types

Many applications will need special purpose object types not included in the standard set. For example, the planar mechanisms simulator of Section 9.2 defines special objects for mechanical parts such as motors, linkage rods, and sliders. Often these objects are simply slight variations on the standard toolkit objects. For example, a linkage rod is simply a line segment that is drawn in a different way and has its length constrained to be fixed at creation.

Most of the standard functionality of `Drawable` objects can be implemented in Whisper, allowing new types of objects to be dynamically defined. For example, the following code defines a new type of `Drawable` that is a simple 2D line segment.

```
(1)  (defun make-line (x1 y1 x2 y2)                    ; procedure to create a line
(2)    (let* ((obj (wh-drawable 'line))                ; create an empty object
(3)           (q   (make-stobj 4)))                    ; make a 4 place state vector
(4)     (set-qvar q 0 x1 y1 x2 y2)                     ; put initial values into state
(5)     (caste-vars q vc-sceneobj)                     ; declare type of variables
(6)     (brobj-add-vars obj q)                         ; install variables in object

(7)     (point-on-vars-2d obj 0 1)                     ; create point connectors
(8)     (point-on-vars-2d obj 2 3)                     ; one for each endpoint

(9)     (bind-in (get-env obj) length                  ; define a length connector
(10)     (dist2d-block (signal q 0) (signal q 1)       ; compute with distance block
(11)                   (signal q 2) (signal q 3)))

(12)    (bind-in (get-env obj) drawf                   ; define draw method
(13)     (lambda (draw-flags)                          ; function of draw params
(14)       (prog (move (val q 0) (val q 1))            ; use GL move/draw commands
(15)             (draw (val q 2) (val q 3)))))         ; to draw line
(16)   obj))                                           ; return created object
```

This defines a procedure that creates an instance of the new type, given initial positions for its endpoints. It first creates an "empty" object (line 2) and a state vector

(line 3) that contains space for 4 variables. The object represents the line by the positions of the endpoints, so the initial values can be placed directly into the variables (line 4). Lines 5 and 6 declare the type of the variables and install them into the object. Lines 7 and 8 create `DistinguishedPoint` connectors for the endpoints. These connectors can obtain their values directly from variables in the state vector. Lines 9 through 11 create a length connector by computing the distance between the two endpoints. Lines 12 through 15 define a draw method for the line segment. The method takes a single argument, the drawing mode, that it ignores since it uses the defaults.

To facilitate the creation of new 2D objects types, Bramble provides a special `define-shape` function, detailed in Section A.2.1. It provides a concise syntax for specifying how the shape is drawn and placing connectors on it. The syntax permits intermediate variables to be specified for convenience and internal constraints on the object to be defined.

7.5.3 Groups

Bramble supports object hierarchies with its `Group` class. A `Group` is a subclass of `Drawable` which contains a list of other objects and a transformation to apply to them. Like the hardware and graphics library it has been built on, Bramble presently supports only linear transformations. Each `Group` has a connector that computes its transformation matrix from its state variables. Different `Group` types define different functions, for example to employ a quaternion or an Euler angle representations for rotation.

The `Group` class automates the process of building graphical hierarchies. `Group` objects manage the hierarchy for their member objects. For example, the members need only draw in their local coordinates as the group ensures that the proper transformations are applied before the object draws itself, permitting the graphics hardware to handle the transformation hierarchy. Making sure that the correct results occur when position or surface orientation connectors are computed is more complicated. The `DistinguishedPoint` class handles this automatically.

Bramble's grouping mechanisms make it simple to build hierarchies. Adding an object to a group requires simply

```
(add-to-group group obj)
```

and everything is configured correctly. Similarly the ungrouping operation

```
(remove-from-group group obj)
```

does all of the required hierarchy management. One complication with ungrouping is that when the transformation is removed, the child object might jump in space. To prevent this, each object keeps a matrix that contains any previously applied transfor-

mation. This matrix is called `leftStuff` since it represents what was to "the left" of the object when it was drawn.[2] When an object is removed from a group, the group's transformation is premultiplied into the object's `leftStuff` matrix. This way the object does not jump when ungrouped. The `leftStuff` matrix is inserted into the matrix stack as each object is drawn.

Each different type of `Group` needs only to define its function that computes its matrix from its state variables. Some auxiliary information, denoting which variables, if any, correspond to rotation, translation, scaling, or center may also be provided. Bramble's standard set of `Group` types include rigid body transformations (translate and rotate) with both quaternions and Euler angles, rotate/translate/scale transformations, and a slider transformation that restricts objects to translate along a vertical line from the normal of a given point.

7.5.4 Cameras and Lights

A camera in Bramble is not just a 3D element. It is used for all transformations between the world coordinate system and screen coordinates. Like a `Group`, a Bramble `Camera's` primary distinction is a function that maps from its state variables to a transformation matrix. In the case of a `Camera,` however, this transformation is from world space to screen space. The inverse of this matrix is used to draw the camera, for example when another camera is looking at it.

The computer graphics literature is filled with many camera models. However, Bramble does not need to support a huge variety because the differential approach makes them unnecessary. As best exemplified by Blinn's work on spacecraft fly-bys [Bli88b], alternate formulations of a camera model are used to provide parameters which serve as convenient controls for the user. With the differential approach, a single camera model can be used with a variety of interactive controls. By mixing and matching controls, the user can specify camera positions as conveniently as with special purpose formulations. The through-the-lens camera controls of Section 8.1.4 provide a building block from which many camera manipulations can be created, including those of Blinn's paper as well as the more common LOOKFROM/LOOKAT model.

Bramble provides a small number of standard camera transforms. Orthographic transformations are provided to produce 2D images, as well as front, top, or side views of 3D scenes. The most common models in computer graphics define camera position by a rotation about the eye point. Models differ in how they represent the rigid body configuration of the camera. Bramble provides camera representations based on quaternions and Euler Angles. Usually, the quaternion representation is used because of its attractive mathematical properties, but an Euler angle representation is also available, mainly for expository purposes.

[2] I use the postmultiply or function application notation, where the point to be transformed is written to the right of the list of matrices.

A common representation of the orientation of a camera in computer graphics is to store a "look at" point that is to appear in the center of the image. A camera parameterized by LOOKFROM/LOOKAT is provided by Bramble. However, the functionality is typically obtained by employing a Quaternion camera and a through-the-lens control, because the representation has better mathematical properties. The primary use of the `LookAtCam` is when the configuration of the camera is going to be statically configured in a program, rather than manipulated. In such a case, the programmer will type the numbers that configure the camera, and therefore will require an easy to type representation.

Like cameras, lights are represented by graphical objects within the scene. A `Light` object is a special type of `Drawable` that has an extra method which initializes the graphics hardware to use the light. Lights provide information for optional shadow generation as a special connector which denotes the "bulb position" in homogeneous coordinates to allow for distant light sources.

Bramble supports several types of light sources, corresponding with the capabilities of the GL graphics library and some of the renderers used. The Bramble light classes are: point lights, distant lights, directional spot lights, and ambient light. As graphical objects in the scene, point and spot light sources can be manipulated and transformed as other graphical objects. This allows placing lights in "fixtures," like the Luxo lamp of Section 1.1. Special connectors for manipulating lights will be described in Section 8.1.5.

7.6 Hooks

Bramble's interaction model most closely resembles a dispatcher or notifier model [FvDFH90] where an application registers callback procedures with a centralized dispatcher that calls the appropriate procedures at the appropriate times. Bramble uses this model for most application behavior, permitting an application to specify functionality and then delegate the flow of control to the main ODE solver loop.

The callback mechanism in Bramble is through Whisper *hooks*. Like their LISP namesakes, Whisper hooks are variables that can be given procedural values that are called at appropriate times. In Bramble, the hook mechanism allows either a pointer to a C++ function or a whisper closure. Often a variable stores a list of hooks to be called, rather than just a single hook.

By defining behavior with hooks, rather than hard-coded routines, the behavior of an application can be dynamically altered. A goal in the design of Bramble was to provide a sufficient set of hooks such that all post-setup application behavior can be defined with hooks. This is more practical with the differential approach because time continuous activities are taken care of by the passage of time, so manipulation actions are discrete events that can be tied to hooks. Only in extremely rare cases do hook

functions not act instantaneously. Because the main loop only cedes control for short instants, processes such as screen update, window maintenance, and object animation can be maintained.

Many of Bramble's hooks are similar to those provided by systems not built with the differential approach. Having the right set of hooks to attach and detach controllers at the appropriate moments is essential to creating graphical manipulation. In this section, we summarize the most important of Bramble's hooks, and provide simple examples of how they can be used with the differential approach. These mechanisms will be used in Chapter 8 to define specific interaction techniques.

7.6.1 Events

An event in Bramble corresponds to a GL input device event, such as a key press or mouse click. Separate events are generated for both button up and down, allowing each to be handled differently. Bramble keeps a dispatch table that assigns a hook to each event. The dispatcher permits defining modifier selectors, allowing different sets of modifier keys to cause events to be interpreted differently. Event hooks are called with the current view as an argument.

The following code implements a 2D dragging behavior. The code fragment attaches a hook that handles left mouse-button down events. The `add-key` function takes an event name, a set of modifier keys, an event type, and a function to call when this event occurs. When it gets the currently selected point from the snap server (Section 7.7.3) and creates a constraint that equates this point with the position of the mouse, which is provided by the view as a connector.

```
(1) (add-key k-leftmouse k-none k-down            ; left-mouse down definition
(2)   (lambda (view)                              ; attach a procedure, called with view
(3)     (if (snapdp)                              ; if there is a selected point
(4)       (let ((c (pt-eq-2d (snapdp)             ; create a constraint attaching
(5)                          (view mouse-port))));    selected point to mouse
(6)         (add-key k-leftmouse k-any k-up       ; redefine left-mouse button up
(7)           (lambda (v) (delete c)))))))        ;    remove the attachment constraint
```

When the left mouse button is pressed, the procedure uses the `snapdp` function to get the currently snapped-to point (line 3). If there is a selected point, a constraint is created that attaches the point to the position of the mouse (line 4 and 5). The procedure also redefines the button up event to delete the constraint (line 6 and 7). Notice how the use of Whisper's lexical scoping rules keeps the internal data of this operation, the constraint, localized and how the code for the dragging action is only executed at the beginning and end of the operation.

Bramble automatically handles GL window events, such as raises, focus changes, and window removal.

7.6.2 Object Hooks

Event hooks are defined on a per-application basis. By default, an event handler does not provide different behavior based on the object being referenced. However, it is often useful to provide behavior on a per-object basis. Some toolkits, such as Inventor [SC92], dispatch events directly to objects to facilitate the diversity of object behavior.

To support diversity of object behavior, Bramble permits objects to specify hooks that are called when certain operations are performed on them. Most of these, such as the save hook, the draw hook, or the deleter hook are primarily useful for dynamically creating a new class of graphical object, as shown in Section 7.5.2. Certain applications permit objects to define specific event hooks and perform the dispatching themselves. For example, the Showoff application, described in Section 9.7, permits each object to define a method to be called when the right mouse button is clicked on it. The application defines the right mouse event handler to determine which object should be notified of the event, and to call the appropriate hook.

One set of standard event handling hooks that is particularly useful for defining interactive behaviors with the differential approach is the grab/ungrab pair. Unlike the example code of the previous section, the standard built-in drag handlers permit objects and specific points to specify hooks that are called before a dragging operation is begun and after it has ended. This is often used to apply constraints that should act during the dragging. Some examples of how this is used are provided in Section 8.3.6.

Object hooks can be defined on a per instance basis, permitting individual instances to each have their own behavior. Because hooks can be defined dynamically, the behavior of an object can be altered while the program runs.

7.6.3 Demons

A `Demon` is a special class of object that causes hooks to be called when certain conditions occur. The common condition for a `Demon` is that a signal is within a threshold of a desired value. Each `Demon` can define three main hooks:

do which is called when the condition the demon is looking for occurs. The demon removes itself after calling this hook.

fails which is called if the demon is removed before its condition is satisfied.

done which is called when the demon is removed.

`Demons` have an optional timeout which causes them to be removed after a specified number of steps, whether or not their conditions have been achieved.

`Demons` are useful for creating a variety of behaviors. One use of a `Demon` is to create a non-persistent constraint by removing a `GoTowards` controller after it achieves its goal. For example, to make example of Section 7.2 non-persistent so that

the cone can be freely manipulated after its top reaches the left edge of the table, we could

```
(set cl (controller (signal (c top) 'x) '= 5))
((demon) (bind done (lambda (x) (delete cl)))
         (bind tolerance .2)
         (bind lifetime 50)
         (bind possessed cl))
```

In this example, after creating the controller, a demon is created that "possesses" it. When the controller gets within tolerance of its target, or when the demon's specified lifetime of 50 steps times out, the demon's done hook is called, deleting the controller. Demons are also often used to remove constraints that are unable to be satisfied, permitting the application to give up.

7.6.4 Other Hooks

Much of the behavior of an application is defined by hooks. Most of these hooks are used for tasks not particular to the differential approach. For example, each `View` defines a hook that draws the background of the image when its screen is cleared. Any of these hooks *could* contain code that alters the set of controllers. However, a small set seems useful for defining interaction techniques. The set includes:

sub-step-hook is called before each call to the differential solver (e.g. each substep of the ODE solver). This hook could be used to create active set methods, although hooks do not have access to solver internals to access the Lagrange multipliers.

step-hook is called after each step. One common use of the step-hook is for periodically altering a `GoTowards` controller to achieve the effect of a `Follow` controller.

add-obj-hook is called each time a new graphical object is added to the system. This can be used to permit automatic registering of new connectors.

redraw-hook is called each time a view is redrawn. This is called once per view per step as all views are updated each step in Bramble. This is often used to provide feedback to the user by drawing an overlay.

7.7 Other Application Components

Most of Bramble is concerned with providing support for the graphical objects which the user will actually place into models. However, there are other supporting objects, such as windows and buttons, which applications require and Bramble supports. Many

other toolkits provide similar support for these things; relevant aspects of how they are handled in Bramble will be mentioned briefly. The emphasis in the development of these objects is in providing tools which will allow these parts of applications to be built quickly, so that more effort can be concentrated on the development of graphical manipulation techniques with the differential approach.

7.7.1 Windows, Views, and Widgets

Bramble contains an object type called a `Frame` which represents a window system window object. A frame can contain many subwindows. One of these subwindows may be a `View` where Bramble draws a depiction of the current state of the graphical objects. Each `View` has an associated camera and attributes which determine how the image is rendered. There can be many views at once, but each must have its own camera and frame.

`Frames` and `Views` are first class objects in Bramble. In fact, `Views` always have connectors that provide the position of the mouse relative to the coordinate frame of the window. This simplifies attaching other connectors to the mouse.

A `Frame` can contain other subwindows besides a `View`. These are used to provide buttons and other widgets around the edges of the `View`. All behavior of the widgets is defined by hooks, including the widget's appearance. These subwindow hooks include methods for drawing, handling mouse clicks, and a "tick" function that is called periodically. Like the graphical objects, the predefined subwindow widgets are designed to automatically update themselves to maintain a consistent view of the values that they access. The provided set includes:

Buttons that watch the value of a Whisper variable, and update automatically.

Radio Buttons that watch a variable and allow its value to be selected from a set.

Differential Sliders that can attach to connectors and create controllers in response to mouse clicks. These sliders permit not only dragging a connector's value, but also nailing or bounding it, as discussed in Section 8.2.1.

Color Sliders that create a set of differential sliders for RGB values along with feedback for what the color is.

Text Elements that allow static display of information.

A special type of widget is provided to help watch the status of an object's state vector. The `VarWatcher` is a separate window that contains sliders for each variable in a particular state vector. The special sliders depict the gather state of the variables as well as permitting their values to be displayed and controlled. `VarWarchers` are particularly useful for debugging. A programmer can monitor an object's state with a single function call.

7.7.2 External Representations

Bramble provides rudimentary features for reading and writing representations of models outside of Bramble. For example, save and load is handled using Whisper, the saved representations are Whisper programs that recreate the model. Applications augment objects to write Whisper code to recreate themselves when reloaded. Objects have hooks that are called during the save process, allowing applications to specify this behavior.

Other external file representations are handled by Bramble. Scene descriptions for renderers, including Renderman [Ups89] and Rayshade [Kol91], can be generated, as well as PostScript pictures. In each case, objects define a hook that is called whenever the object needs to be written to a file. By defining this behavior as a hook, applications can alter the way that objects are written on either a per class, or even a per instance basis. Bramble also has support for writing bitmapped images of views to disk.

7.7.3 Picking and Snapping

Picking is a fundamental operation in any graphical editor. Bramble contains methods for easily using the standard Iris GL pick by redraw mechanism. Picking by ray casting is also supported for object classes that provide a ray intersection routine.

The most commonly used picking mechanism is cursor snapping. Cursor snapping continuously locates the cursor near important points in the view. It is used to aid in object selection and precision cursor positioning. Cursor snapping was first introduced in Sketchpad [Sut63], and has since been further extended to better support precision manipulation (as in Snap-Dragging [Bie89]), infer constraints (as in Briar [GW94]), or provide feedback to the user as to the range of available options [Hud90].

In Bramble, cursor snapping is handled separately from the event process. The model for cursor snapping is also continuous, but in practice it is updated only at the end of each step. Event actions can inquire about the state of snapping at any time. A single Whisper call returns the object currently snapped to. The snap server actually keeps a list of targets that are near the mouse, to permit hysteresis and cycling between objects that are close to one another. At present, the Bramble snap server snaps only to existing `DistinguishedPoint` connectors. It has been designed to be extended to support the range of snapping seen in Snap-Dragging [Bie90, BS86].

Facilities are provided to control the scope of snapping, permitting techniques such as Semantic Snapping [Hud90]. By using either a hook or status bits in each potential target, the snap server is able to cull objects based on criteria other than being close to the mouse. One common use of this is to limit snap-targets to legal choices for the current operation. Bramble automatically provides the user with feedback of the snapping state. Applications can provide more than just the cursor feedback, for example highlighting objects that are snapped to.

Constraint inferencing from snapping is supported by Bramble. By using snap-server accesses in object creation and manipulation event handler hooks, augmented snapping [GW94] can be implemented easily. Augmented snapping, discussed in Section 9.1, automatically generates constraints from snapping operations in order to make the positioning operation of the snap persistent. For example, the dragging handler of Section 7.6.1, might be extended to

```
(1)  (add-key k-leftmouse k-none k-down            ; left-mouse down definition
(2)    (lambda (view)                              ; attach a procedure, called with view
(3)      (if (snapdp)                              ; if there is a selected point
(4)        (let* ((p (snapdp))                     ; store selected point
(5)               (c (pt-eq-2d p (view mouse-port)))) ; attach point to mouse
(6)          (add-key k-leftmouse k-any k-up       ; redefine left-mouse button up
(7)            (lambda (v)                         ; now also infers constraint
(8)             (prog (delete c)                   ;     remove the attachment constraint
(9)                   (if (snapdp)                 ;   if there's a point to connect to
(10)                    (pt-eq-2d p (snapdp))))))))))) ; infer a constraint
```

The code has two differences from the earlier example. First, the point that is being dragged is stored in line 4. Secondly, when the mouse button is released, line 9 checks to see if the cursor is snapped to a point. If it is, a constraint is created in line 10 that connects it to the point that was dragged. In Section A.2, a code fragment demonstrates the addition of augmented snapping to the creation of a line segment by rubber banding.

7.8 The Bramble Standard 3D Interface

Bramble provides a standard set of objects and connectors to support basic 3D interaction. The goal is to provide a fast and easy interface for 3D experimentation. The basic interface, shown in Figure 7.3, has a particular style derived from systems built in the CMU Animation Lab over the past few years. Applications can use or ignore these interface elements.

An important part of Bramble's 3D interface is a floor called the *groundplane* and an optional back wall. This reference object defines the coordinate system and gives the user a reference frame called a stage[HZR+92]. To further aid the user's perception of 3D objects, Bramble can draw shadows on these reference objects. These plane shadows can be easily generated with the available hardware using techniques described in [Bli88a]. Bramble's shadows can either be simple drop shadows that are directly below the objects as if the light was the sun at high noon, or shadows computed from the positions of the light sources.

A `Follow` controller couples the motion of a connector to the motion of an input device. Any connector which represents a position in 3–space could be connected to a 3D input device. Bramble's standard 3D input device is a virtual device that uses the

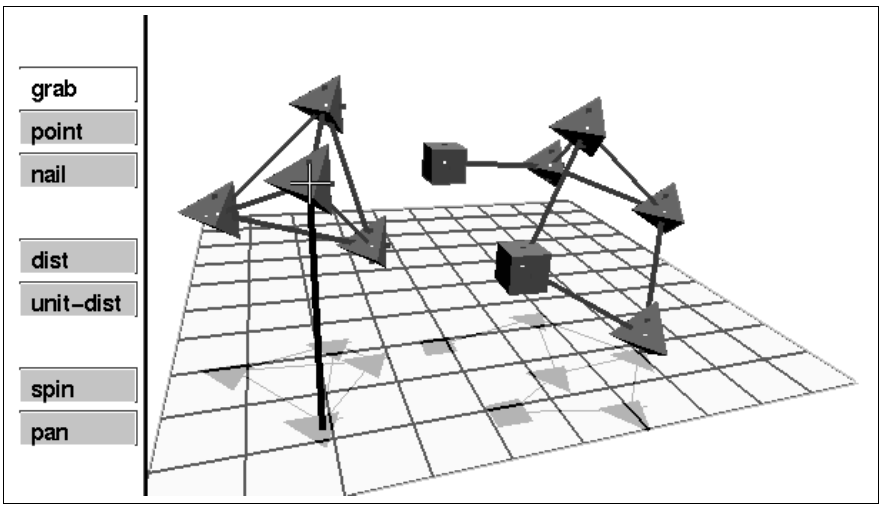

Figure 7.3: The *Point Tinkertoys* application demonstrates Bramble's standard 3D interface. The user manipulates the point objects using the mousepole. A groundplane and shadows are provided to help depth perception. This application is discussed in Section 9.6.

mouse along with a special graphical object called the *mousepole*.[3] The mousepole is a vertical line which extends from the floor to the mouse position. The line is used to provide feedback of the 3D cursor location. As the mouse is moved, the pole tracks it, with the top point moving parallel to the groundplane. When a mouse button that has been designated as the "elevator" button is held down, the pole top moves in a vertical plane instead of a horizontal one. The tip of a mouse pole can be tracked by a controller.

A standard 3D interface package can be loaded from any Bramble application. It provides a standard set of key bindings and object manipulation techniques including mousepole manipulation of scene objects, virtual trackball-like manipulation of the viewpoint by grabbing the corners of the groundplane, and viewpoint panning by grabbing the center of the groundplane.

In addition to its standard 3D interface, Bramble provides support for developing other types of 3D interactions. Objects provide a variety of connectors which serve as building blocks for creating interaction techniques. Many of these will be described in Chapter 8.

[3] Although the mousepole is unpublished to date, I credit it to Andy Witkin.

The material object of observation, the bicycle or the rotisserie, can't be right or wrong. Molecules are molecules. They don't have any ethical codes to follow except those people give them. The test of the machine is the satisfaction it gives you.

— Robert Pirsig
Zen and the Art of Motorcycle Maintenance, p 146

Chapter 8

Interaction Techniques

The differential approach allows a range of interaction techniques to be created by defining connectors which compute interesting attributes of objects, attaching controllers to these connectors at the proper times to cause the object which they depend on to move, and by using combinations of these controllers to create more complicated behaviors. This chapter discusses these topics in that order.

Throughout this chapter, the differential approach will be applied to specific interaction problems. In many cases, we will simply recreate previous interaction techniques on top of the new abstractions, often in a way that allows them to be extended, generalized, or at least implemented with fewer of the typical hassles. Several of the interaction techniques, however, would be difficult or impossible to create without the approach.

8.1 Attributes to Control

An important feature of the differential approach is the flexibility it provides in the types of attributes that can be controlled. In this section, we provide examples of attributes which potentially make interesting connectors to control. All of these examples are available in Bramble's standard library.

One consideration is that some attributes work better than others, either from a mathematical or user interface perspective. While developing methods for determining the effectiveness of an attribute as a control is left for future work in Section 10.3, here are some intuitions developed from experience:

1. Simpler functions work better.

2. One constraint should remove one degree of freedom. It it much better to use multiple equations than to combine multiple constraints into a single equation.

For example, it is better to have $x = 0$ and $y = 0$ than $x^2 + y^2 = 0$.

3. The derivatives should not vanish when the constraint is met. This often relates to 2. In the example, the derivatives of $x^2 + y^2$ are 0 when $x = 0$ and $y = 0$.

4. Normalizations throw away information, and therefore should be avoided.

5. Controls with physical analogies are easier to explain to the user and to understand the mathematical behavior.

6. Controls that are positional (e.g. compute a position in space) are easier to connect to input devices and to understand.

Each of these issues will be discussed in the specific examples to which they apply.

8.1.1 2D Object Controls

The most basic attributes are the positions of points on 2D objects. These can be directly controlled by attaching their values to the mouse's position. Since the system must be able to compute points in order to draw the object, the functions required for manipulation are known. Often, points are placed at interesting points such as the center of an object, in addition to places distributed around the object.

The `define-shape` function in Bramble, detailed in Section A.2.1, provides a mechanism for defining 2D objects. It demonstrates how the same information used to draw an object is used to control it.

For parametric curves, the simplicity of the differential approach is especially apparent. A parametric curve is defined by a function

$$\{x, y\} = \mathbf{f}(u, \mathbf{q}). \tag{8.1}$$

This function computes the position of a point of the curve, given a value for the free parameter and the parameters defining the configuration. The ability to compute this function is necessary to draw the curve. Just as we draw the curve by specifying u values for specific points, we can create connectors to manipulate the curve by specifying u values. Using the parametric function for a connector requires the derivatives of the function that can be obtained easily and automatically. Even if the function is provided only as a black box that can be evaluated, the derivatives can be estimated numerically using finite differences.

The use of connectors from the parametric function provides an automatic and uniform methods for providing interfaces. All that is required to define an object type is the parametric function. Connectors can be placed along the length of the object, permitting the user to grab the object at various points. Each type of object is manipulated in exactly the same way: the user can grab any point on it and pull it. All parameters of the

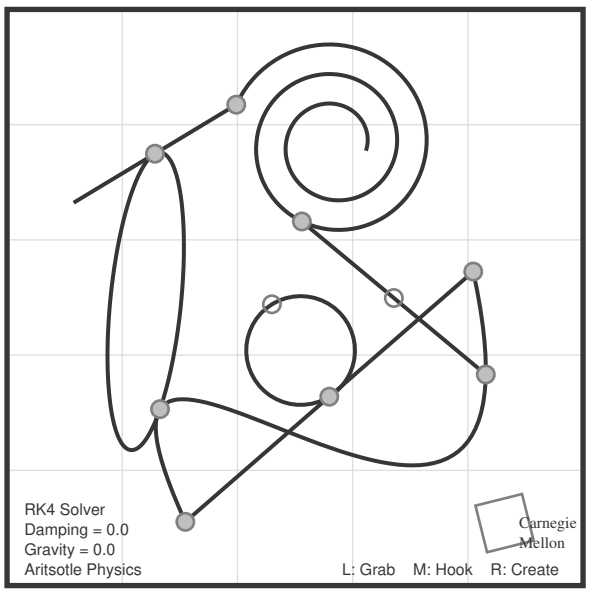

Figure 8.1: A simple curve editor showing a variety of parametric curve types. New types can be defined simply by providing the parametric function used to draw the objects. All objects are manipulated uniformly: pulling a point on the objects causes all of the object's parameters to be adjusted accordingly.

object are affected, not just their translations. The mass matrix metric of Section 3.4.1 provides a uniform metric for curves as well, as first demonstrated in the FF system by Witkin [Wit89b]. A later implementation that I wrote is shown in Figure 8.1, and was presented in [GW91a]. These systems both permitted the definition of new types of objects at compile time by specifying a parametric function to a symbolic mathematics package that automatically generated the code for the object. A later version, described in Section 9.4, permits new functions to be defined dynamically.

For a specific object, the uniform interface may not be better than a hand-crafted one. However, it may not always be practical or possible to devise good interfaces for all object types. For a parametric curve in the differential approach, interfaces do not need to be defined for each object type. Given the parametric function used to draw the object, the code to compute the connectors can be automatically generated. In fact, to add a new curve type to a drawing editor, all that needs to be provided by the programmer is the parametric function and a little auxiliary information such as how finely to sample the curve and initial values for the parameters. Everything else can be generated automatically. A prototype system, described in Section 9.3, even adds an icon creating instances of the object to the program's interface.

A connector for a particular u value provides a point on the curve that can control

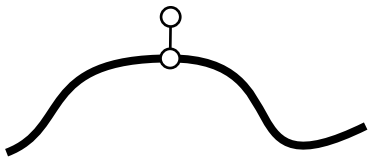

Figure 8.2: A crowbar point is computed by an offset of the normal to a point on the curve. The crowbar provides a handle to control the normal of the point that is positional, and therefore can be directly coupled to the pointing device.

the parameters. If the connector stores u as a modifiable state variable, the point can slide along the curve as well as to move the curve. We call such a point that can slide around on its parent objects a *bead.*

Normals and tangents can also be computed for points on 2D curves. The functions to compute them can be determined automatically from the parametric function used to draw the curve by symbolic differentiation. One way to present the position of the normal or tangent to the user is by showing a "crowbar" point that is a point offset from an original point on the curve by the tangent or normal, as shown in Figure 8.2. This attribute provides a connector that can be grabbed just as a point on the curve can. It is computed by simply adding the tangent and position attributes together.

8.1.2 2D Geometric Relationships

A wide variety of attributes express geometric relationships among objects. Very often, these relationships can be expressed as functions of a number of points, permitting modularity: any point provided by any type of object can be plugged in. There is no need for special types of controls for each object variety. Attributes that compute geometric relationships are typically used as constraints with their values held constant, rather than being directly controlled by the user.

The most basic 2D geometric relationship is attachment or coincidence of two points. An attribute for attachment is computed by subtracting the positions of the two points. The distance between two 2D points is also a useful attribute. However, using a single distance constraint driven to a driven value does not function well as an attachment constraint. This case violates two of the rules for attributes: it uses a single constraint to remove two degrees of freedom; and the derivatives vanish when the constraint is met. An attachment constraint that uses a subtraction for each coordinate functions much better.

Collinearity of three points is a useful constraint. In 2D, it is straightforward to create by computing an attribute that is the area (determinant) of the triangle formed

by the points, or the distance of one point to the line formed by the other two. Driving either of these two attributes to zero creates the collinearity.

The simplest implementation of a point-on-curve constraint uses an implicit formula for the curve. While this works for a small class of objects such as circles, an alternative approach can be used for the general case of parametric curves. A "bead" point on the curve is created by creating a new point connector allowing its free parameter to be a variable. This bead is attached to the point that is to lie on the object. The point is therefore attached to the object, but free to move along its length.

Relative orientations of objects can be controlled by attributes involving dot products. For example, driving the dot product of two vectors to zero makes them perpendicular. Such a control might also simply drive the length of one (or both) of the vectors to 0, also achieving the desired dot product value. To avoid this, the vectors are typically normalized before the dot product. Inverse trigonometric functions typically have singularities in their derivatives making angle computations using them difficult to use in defining connectors, so it is difficult to place constraints on absolute angles.

The constraint that two line segments, or four points, are parallel can be expressed as the normal of one segment dotted with the other segment's direction vector must be zero.

An attribute that computes the depth of a point into a half-plane is computed by taking the dot product of the normal to the half-plane with the difference vector of the point and a point on the dividing line. This signed distance is useful for creating point inside polygon constraints, which are easily created for convex polygons by a conjunction of point half-plane constraints, or to make point-outside-polygon constraints, as will be described in Section 8.4.2.

8.1.3 3D Objects and Constraints

As in 2D, point positions are the main connectors for manipulating 3D objects. Points in 3D can define not only their positions, but also their normals and two tangent vectors to define a coordinate frame.

The position of a point is not only a function of the object it is part of, but also, the transformation hierarchy "above" the object. Surface geometry must be similarly transformed. The same transformation applied to points is applied to tangent vectors, and the inverse transpose is applied to the normal vector. Rather than compute the derivatives of the inverse transform, `DistinguishedPoint` connectors transform their tangent vectors and then take the cross product. Types of `DistinguishedPoint` do not compute normal vectors themselves.

The basic 2D constraints also transfer to 3D. Nailing a point, attaching points, and controlling distances are equally simple. A collinearity constraint is more problematic. The constraint removes two degrees of freedom. Point to line distance with driven to a zero value has the same problems as a 2D distance constraint driven to zero. Point

to line distance is a useful constraint for expressing that a point is on a cylinder with a non-zero radius. A better way to constrain a given point to lie on a line is by creating a connector with a free parameter inside it that specifies a position on the line, and constraining this connector to be coincident with the given point. The "bead" connector can be computed by $u\mathbf{p_1} + (1-u)\mathbf{p_2}$. Such an approach uses two constraints to remove the two degrees of freedom (actually, it uses three constraints, but adds an extra degree of freedom sliding along the line, so the net result is two constraints).

Aligning two coordinate frames requires 6 constraints: 3 to co-locate the origins, and 3 to equate the orientations. A related constraint aligns the positions and the normal vectors of two coordinate frames. This uses five individual constraints: 3 for position, and two that compute the dot product of the normal vector of one point with both tangent vectors of the other. The constraints express a contact-like relationship that allows the objects to slide along one another as if in contact. Such a constraint is best represented by attributes that compute the dot products of one point's normal vector with the other point's tangent vectors.

Expressing that the two planes are parallel, as in the last paragraph, can be done in two ways: by maximizing the dot product of the normals, or by driving the dot products of the normal of one point and the tangent vectors of the second to zero. The former has the advantage that its sign can be used to insure that the normals face the same direction. However, it is a single constraint that removes two degrees of freedom. Therefore, it does not work well to maintain that the planes remain parallel. It can be used to establish the correct relative orientations initially, and the other formulation can be subsequently used when the planes are close to parallel. Several of the systems I have constructed use this strategy: a demon (see Section 7.6.3) switches between a normal dot product controller used to get the orientations close initially, and the normal/tangent dot products used to maintain the relative orientations.

Various point-on-object constraints can be represented by using the implicit formulation of the object. Constraining a point to lie on a plane, cone, cylinder or a sphere can be easily achieved using an implicit representation. Surface beads permit point-on-surface relations for parametric surfaces.

8.1.4 Camera Controls

The problem of specifying a viewing transformation or virtual camera configuration is a central problem for 3D graphics. Previous work on the problem of camera control is discussed in Section 2.4.2. My work on using the differential approach for the problem of camera control was first presented in [GW92], and a video accompanying the paper [Gle92b] demonstrates the interaction.

Computer graphics viewing models are defined by linear transformations in homogeneous coordinates. That is, a viewing transformation is defined by a 4×4 matrix,

and the position that a point appears in the image is given by

$$\mathbf{p} = \mathbf{h}(\mathbf{V}\mathbf{x}), \tag{8.2}$$

where $\mathbf{x}$ is the world-space point that projects to $\mathbf{p}$, $\mathbf{V}$ is a homogeneous matrix representing the combined projection and viewing transformations, and $\mathbf{h}$ is a function that converts homogeneous coordinates into 2-D image coordinates, defined by

$$\mathbf{h}(\mathbf{x}) = \left[\frac{x_1}{x_4}, \frac{x_2}{x_4}\right], \tag{8.3}$$

where the x_i's are components of homogeneous point $\mathbf{x}$. The matrix $\mathbf{V}$ is some function of the camera model parameters.

A perspective camera transformation can be thought of as rigid body camera object that exists in the same world space as the objects it views, just as a camera in the real world. If there are other views in which the camera object is visible, the camera can be manipulated as any other object in the scene. The transformation of the camera into world space can be found by inverting the viewing transform without the perspective projection. Because matrix inversion is difficult to differentiate, simply creating a connector for the inverse transformation given the camera transformation is impractical. However, the function for the inverse transformation can be provided easily for many camera models. Rather than inverting the entire camera matrix that is the composition of several simpler pieces, the pieces are inverted and composed to create the inverse camera transform. Often the pieces are primitive transformations that are easy to invert. Providing the inverse transform for cameras permits connectors on a camera for attributes such as the tip of the lens or the top of the camera. Such points are not only useful for dragging, but also for expressing constraints such as the camera is always relatively right side up.

Rather than controlling a camera by its position in the world, it is often useful to control it by manipulating what is seen in its image. We call such controls *through-the-lens* controls. The most basic through-the-lens control is an attribute that computes the position where a point in the world projects onto the film plane of the camera (the image seen through the camera). Equation 8.2 is used to compute an attribute that serves as the control. These controls are independent of the variety of camera model. Any viewing transformation can be controlled, although Equation 8.2 may be replaced by some other function if a non-standard camera model is used.

It is important to realize that the image position of a point depends on both the camera and the point's position in the world. A through-the-lens control can be used to affect either, or both, depending on which object's parameters are free to change. Through-the-lens controls are a particularly important type of attribute because they provide a way to couple the 3D world to the space of 2D input devices, permitting manipulation of 3D objects and viewpoints by pointing to screen positions.

A single through-the-lens control does not specify enough information to usefully control a 3D object or camera transformation by itself. With the many degrees of freedom involved, the 2 controls of a through-the-lens point are underdetermined, and good default behaviors are difficult to specify with the optimization objectives. Typically, other constraints are used in coordination with through-the-lens controls to provide desired behaviors. Many examples will be given in Section 8.2.

The same constraint relationships applied to points on 2D objects can also be applied to through-the-lens controls. For example, an attribute might measure the distance between two image points. Placing constraints to keep through-the-lens point controls within in a region, for example on the screen, is often useful.

Through-the-lens controls are the most basic element of *appearance-based manipulation.* Such an approach permits the user to control a 3D world by specifying what is seen in a picture of it. It is useful because images are 2D and therefore map nicely to common input devices, and also because often an image is the goal of a graphical application.

8.1.5 Manipulating Lights and Materials

The configuration of lighting and material properties can also be controlled by a variety of attributes. Like cameras, lights are typically objects in the scene and can be manipulated as such. This is especially useful for point and spot light sources. With both types of lights, connectors for the position of the bulb is the most obvious control. For spotlights, points on the aperture, both in the center and around the rim, are useful not only for direct grabbing, but also for use in the shadow manipulation techniques of the next section.

Light intensity and material surface properties are parameters that can be represented in the state vector. This permits their values to be constrained.

One appearance-based method for controlling lights and material properties is to directly manipulate the color of points in the image, for example by attaching a point's color to a set of sliders. Techniques for manipulating point colors are presented for changing material surface colors in [HH90] and for changing light colors in [SDS+93]. Using the differential approach to control point colors permits these techniques, but also can be used to alter other parameters that affect apparent colors such as surface orientations and light positions.

The equation that computes the color that a particular point appears must be known to the system so it can draw it. A shading model computes the color from the surface geometry, lighting, and surface properties. The color that a point appears is the result of a physical process which has been modeled to varying degrees of accuracy [Hal89]. Modeling and rendering systems most often use simpler models. The parameters to these models serve as the controls which are used to specify surface properties. Therefore, a user interested in the colors in an image must understand this model, even though

it is merely a historical artifact of computer graphics research.

The most commonly used shading model [FvDFH90, Sil91] divides lighting into three components, diffuse, specular, and ambient parts. A surface is specified by three colors which define how the surface reflects each of these components. The color of a point is

$$\mathbf{c} = \mathbf{c}_a * \mathbf{i}_a + \mathbf{c}_d * \mathbf{i}_d + \mathbf{c}_s * \mathbf{i}_s, \tag{8.4}$$

where the asterisk denotes component-wise multiplication, $\mathbf{c}_a$, $\mathbf{c}_d$, and $\mathbf{c}_s$ are the surface color properties, and $\mathbf{i}_a$, $\mathbf{i}_d$, and $\mathbf{i}_s$ are the intensities of each type of light at the point. Each is typically represented as a 3-vector containing RGB color values.

If we know how much light is available at a point $(\mathbf{i}_a, \mathbf{i}_d, \mathbf{i}_s)$, we can manipulate the color of the point in order to control the surface properties, $(\mathbf{c}_a, \mathbf{c}_d, \mathbf{c}_s)$. This is most interesting when we control the colors of multiple points which share the same parameters. In the special case of fixed lighting and geometry, all of the apparent colors are linear functions of the surface properties. This is exploited by the system described in [SDS+93].

The amount of light which strikes a point can be computed by summing the contributions of each light source. The amount of ambient light, $\mathbf{i}_a$, is merely a scene-wide constant. For other varieties of light, the contribution of each light source is summed. The amount of diffuse illumination given by

$$\mathbf{i}_d = \sum_{i \in lights} \mathbf{i}_i (\hat{\mathbf{n}} \cdot \hat{\mathbf{l}}_i), \tag{8.5}$$

and the specular illumination by

$$\mathbf{i}_s = \sum_{i \in lights} \mathbf{i}_i (\hat{\mathbf{n}} \cdot \hat{\mathbf{h}})^{shi}, \tag{8.6}$$

where $\mathbf{i}_i$ is the intensity of light source i, $\hat{\mathbf{n}}$ is the point's unit normal vector, $\hat{\mathbf{l}}_i$ is the normalized vector from the point to the light, and where $\hat{\mathbf{h}}$ is the normalized vector that bisects the vectors from the point to the eye and light, and shi is a parameter which controls the highlights size.

Controlling the colors of points is useful in controlling the lighting. Manipulating the color of a point can alter the intensities of the lights that contribute. This provides a way of achieving desired color effects when multiple colored lights are used, as in stage lighting. Colors can even be used to control the geometry of lighting. Altering the color can cause a face to turn to towards or away from a light, or cause a light to move so it provides the proper amount of illumination.

Using the lighting equation as a control has a great deal of potential, but, it is problematic. Even simple lighting models are complicated expressions of many parameters. More complicated lighting models may not even be expressible as closed form differentiable expressions.

Even in the simple shading model used by the Iris hardware and in Bramble, many terms of the equation do not function well as controls. In particular, the specular component of lighting is particularly problematic. The amount of specularity at a point is given by Equation 8.6. Most obviously, it involves an exponentiation which gives it very non-linear behavior with rapidly changing derivatives. Also, it involves normalizing a quantity twice in computing the normalized half angle vector. Although this makes it difficult to use color control to position specular highlights, the techniques of the next section can handle such tasks by treating specular highlights as the reflection of the light source.

8.1.6 Shadows and Reflections

We treat reflections and shadows in a similar manner: we compute where the image of a particular point appears on a particular surface. Shadows are the simpler case. A point is the shadow of another if the two points and the light source are collinear. This is typically implemented by creating a bead on the ray from the light source through the shadowing point. The bead can either be attached to some point that is to be shadowed or constrained to lie on the surface of the shadowed object. An additional constraint could be added that insures that the occluding object is between the shadow and the light source.

The explicit use of the ray between the light and the shadow also serves as a mechanism to display the shadowing. With available graphics hardware, general inter-object shadows cannot be drawn efficiently, so drawing the line serves as a feedback device, as shown in Figure 8.3. Manipulating the shadows can control the position of the light, the shadowing object, or both.

When the shadow is cast onto a planar surface, its position can be computed directly. The general projection matrix can be used to compute where the shadow ray hits a plane. This is a simple computation, given by

$$\mathbf{p} = \mathbf{h}(\mathbf{S}\mathbf{x}), \tag{8.7}$$

where $\mathbf{h}$ is given by Equation 8.3, the function which converts from homogeneous coordinates, $\mathbf{x}$ is the position go the point to be projected, and $\mathbf{S}$ is the projection matrix. In [Bli88a], the matrices are derived for projection from point and distant light sources onto a ground plane. More generally, the matrix can be computed for a light source point $\mathbf{l}$ in homogeneous coordinates onto a plane $(\mathcal{P}_a x + \mathcal{P}_b y + \mathcal{P}_c z + \mathcal{P}_d = 0)$ by[1]

$$\mathbf{S}(\mathbf{l}, \mathcal{P}) = \begin{bmatrix} \mathcal{P}_b \mathrm{l}_y + \mathcal{P}_c \mathrm{l}_z + \mathcal{P}_d \mathrm{l}_w & -\mathcal{P}_b \mathrm{l}_x & -\mathcal{P}_c \mathrm{l}_x & -\mathcal{P}_d \mathrm{l}_x \\ -\mathcal{P}_a \mathrm{l}_y & \mathcal{P}_a \mathrm{l}_x + \mathcal{P}_c \mathrm{l}_z + \mathcal{P}_d \mathrm{l}_w & -\mathcal{P}_c \mathrm{l}_y & -\mathcal{P}_d \mathrm{l}_y \\ -\mathcal{P}_a \mathrm{l}_z & -\mathcal{P}_b \mathrm{l}_z & \mathcal{P}_a \mathrm{l}_x + \mathcal{P}_b \mathrm{l}_y + \mathcal{P}_d \mathrm{l}_w & -\mathcal{P}_d \mathrm{l}_z \\ -\mathcal{P}_a \mathrm{l}_w & -\mathcal{P}_b \mathrm{l}_w & -\mathcal{P}_c \mathrm{l}_w & \mathcal{P}_a \mathrm{l}_x + \mathcal{P}_b \mathrm{l}_y + \mathcal{P}_c \mathrm{l}_z \end{bmatrix}. \tag{8.8}$$

[1] Thanks to Pat Hanrahan for lending us this very useful matrix.

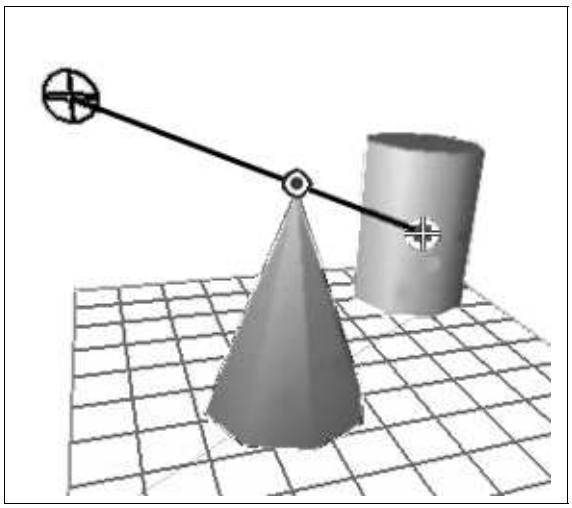

Figure 8.3: Lights and objects are manipulated by controlling a shadow. Because inter-object shadows cannot be drawn with the available graphics hardware, a line is used to connect the light, the shadowing point, and its image.

The shadow matrix is useful for drawing shadows using the standard rendering pipeline, as described by Blinn [Bli88a]. We merely concatenate the shadow matrix and the viewing transform, and redraw the scene in a dimmed grey color. Simply drawing a dark color on the ground plane fails when the shadows go off the rectangular stage surface and seem to float in space. I have used two methods to give the illusion of dimming. One is to draw the shadows using a dither pattern and the sky color, which is chosen to be darker than the ground color. An alternate approach is to choose the ground and sky colors such that turning a particular bit off in the ground color dims it, but does not affect the sky color. Shadows are then drawn using a write mask so that only this one bit is affected.

Positions of shadow beads can be computed using equation 8.7 and directly constrained and controlled. A system can permit the user to grab shadow points and to drag them. This permits not only the interaction techniques of [HZR+92], but manipulation of lights as well.

The spot caused by a spotlight onto the reference planes is also useful. Drawing this spot in a bright color, using techniques like those used for shadows, gives an indication of where the spotlight is aimed. This ring is an important part in some lighting effects. A spotlight can be manipulated by controlling the projections of aperture points on the ground. This is used in the Luxo lamp example of the introduction.

The same projection techniques apply to mirrors as well as to shadows. We use the same matrix, except that we place the projection point at the virtual eyepoint rather at the light source. The virtual eyepoint is the place where a viewer would have to stand to see the reflected image through the surface if the surface were transparent [Ups89],as

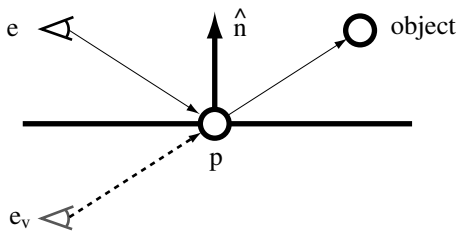

Figure 8.4: An observer with eye point **e** sees the reflection of an object at point **p** on a shiny surface with unit normal vector $\hat{\mathbf{n}}$. The virtual eyepoint $\mathbf{e_v}$ is where the eye would be located to see the image of the object at **p** if the surface were transparent.

shown in Figure 8.4. The position of this point is computed by

$$\mathbf{e}_v = \mathbf{e} + 2((\mathbf{e} - \mathbf{p}) \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}, \tag{8.9}$$

Where $\mathbf{e}$ is the position of the eyepoint, $\mathbf{p}$ is the position of the surface point on the mirror, and $\hat{\mathbf{n}}$ is the surface normal at the point. Although there is potentially a different virtual eyepoint for each surface point, all points on a planar surface share the same virtual eyepoint. Explicitly computing $\mathbf{e}_v$ is preferable to other reflection formulations as it provides a geometric position as an intermediate result that can be examined (guideline 6).

Computing the virtual eyepoint allows the projection matrix of Equation 8.8 to be used to compute the transformations that place reflections on a planar mirror. This allows reflections to be drawn using the rendering pipeline. To provide proper occlusions in a z-buffer, we draw the objects slightly above the surface of the mirror, with a height equal to the inverse of the distance from the mirror so that closer objects occlude ones further from the mirror. Also, because all the objects in the reflections are flat, their normals are invalid, so lighting calculations cannot be done. The positions of points in planar mirrors can serve as controls to manipulate the mirror, the reflected point, and even the camera.

The analogy of techniques for manipulating shadows and reflections extends to nonplanar surfaces as well. Although there is no way to efficiently draw these reflections with available graphics hardware, a line connecting object, image, and virtual eyepoint serves for feedback. Similarly, enforcing collinearity between these three points permits using the manipulation of one to control the others. An example of this is shown in Figure 8.5.

Reflection techniques can be used to position specular highlights on surfaces. The light source is placed such that it is seen in the reflection on a surface point. By sliding the point as a bead around on the surface, the highlight can be positioned. This can be used to position the light source or to alter the surface geometry.

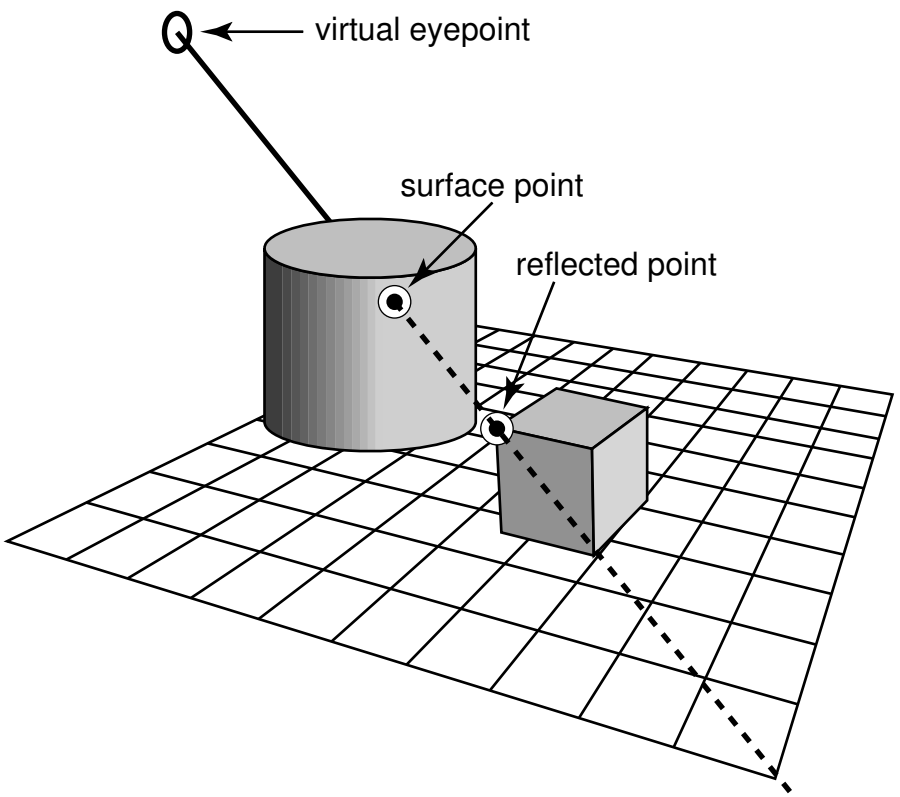

Figure 8.5: The cube is controlled by manipulating the position of its reflection in the cylinder. Since the reflection in the curved surface cannot be drawn, the line connecting the point of the cube, its reflection, and the virtual eyepoint is drawn.

8.1.7 Free Form Curves and Surfaces

Free-form curves and surfaces are traditionally a difficult problem for interface design. Typical approaches to creating such objects require devising representations that are sufficiently expressive, are convenient for the user to manipulate directly, and have good mathematical properties. This typically leads to interfaces like the control points of B-Splines, that sacrifice usability. Some of the difficulty stems from the fact that the representations also serve as the parameterizations of the objects. Because of this, it would seem that the differential approach would be a natural solution.

For many types of curve and surface representations, the differential approach can be applied. For many types of parametric surfaces such as B-splines and NURBS, the positions of points are simple functions of the parameters that can be computed as connectors and manipulated differentially. Several of the approaches in the literature are variants of this, using different schemes for computing parameters changes. For example, Welch, Gleicher, and Witkin [WGW91], Welch and Witkin [WW92], Fowler [Fow92], and Hsu et al.[HHK92] all use various constrained optimization techniques to map manipulation of points on surfaces to the underlying parameters. This permits the objects to be manipulated by controlling points on them, rather than control points. In some cases, users may prefer to use control points to manipulate curves or surfaces. Since the positions of the control points can be computed from the curve or surface, control points can be provided even if another representation is used. This allows, for example, to use Bezier control points to manipulate a B-Spline curve.

Unfortunately, the differential approach of this thesis is insufficient for adequately addressing the issues in manipulating free form curves and surfaces for many reasons:

- free form objects have too many degrees of freedom;
- free form objects require global control for effective manipulation. Such control can be provided only with attributes that compute properties of the entire surface, or large regions of it;
- free form objects often need fine detail local control, requiring adaptive subdivision;
- many of the constraints that are used to sculpt free form objects' are specialized cases that can be handled more effectively for the large numbers of degrees of freedom.

In short, the methods of this thesis do apply to the manipulation of curves and surfaces, as we showed in [WGW91], but, by themselves, the methods do not address many of the difficult issues and more specialized methods apply. Controlling free form curves and surfaces is a very important problem, and is therefore well studied. Some interesting optimization-based approaches are explored by Celniker and Gossard [CG91] and by Welch and Witkin [WW92] and [WW94].

8.2 Strategies for Interaction

The previous section described a large number of attributes that can be computed and provided as connectors on objects. With the differential approach, we can control the objects by attaching controllers to the connectors. In this section, we consider some strategies for determining what controllers to attach to which connectors at what time.

8.2.1 Presenting the Abstractions to the User

The most obvious way to employ the differential approach is to provide the abstractions to the user as directly as possible. At an extreme, the user could be presented with a schematic representation, like the diagrams of Section 1.3.3. While this graph editing approach has been attempted in systems such as the SPAR Modeling Testbed [FW88], Condor [Kas92] and ThingLab [Bor86], I do not believe it is in the spirit of direct manipulation, and prefer to hide such representations from the user.

The direct application of the differential approach gives the user a palette of connectors to which controllers can be applied. Connectors that represent positions can be controlled by attaching them to the mouse. Non-geometric values can be controlled with sliders or similar widgets. To specify multiple controls, the user might either nail connectors to their current values, or place goal points for connectors to seek.

Presenting the abstractions directly to the user has a number of benefits. It permits the user to select controls that are applicable to their problem, and to mix and match controls as needed. It maximizes flexibility over what the user can do. Such a direct approach is a useful strategy for testing out new types of controls, as shown in the scene composition program of Section 9.7. By utilizing efficient mechanisms for selection, a large array of attribute types can be provided for control.

The advantages of this direct application are also its most serious problem. The question of how to present the controls to the user can be difficult, especially as the number of types that are available grows. The interface must help the user understand the potential choices and to find the controls that are applicable to their task. While the flexibility of a range of controls is useful, it also means that a user might need to spend time deciding which control to use and how to employ it. A system must show the user what is being controlled and explain the behavior of the objects. The issues of scalability also arise as the user adds more controls, creating more complicated behaviors and worse performance.

One variety of direct application of the differential approach is the "live world." A live world is an environment where just about any point can be grabbed and manipulated. In such an application, objects, shadows, reflections, lights, or just about any other entity can be grabbed and dragged. This provides a uniform interface for a wide variety of tasks. However, it still does not solve the problems. A user must be made aware of what can be dragged and what cannot. Some mechanism for specifying what

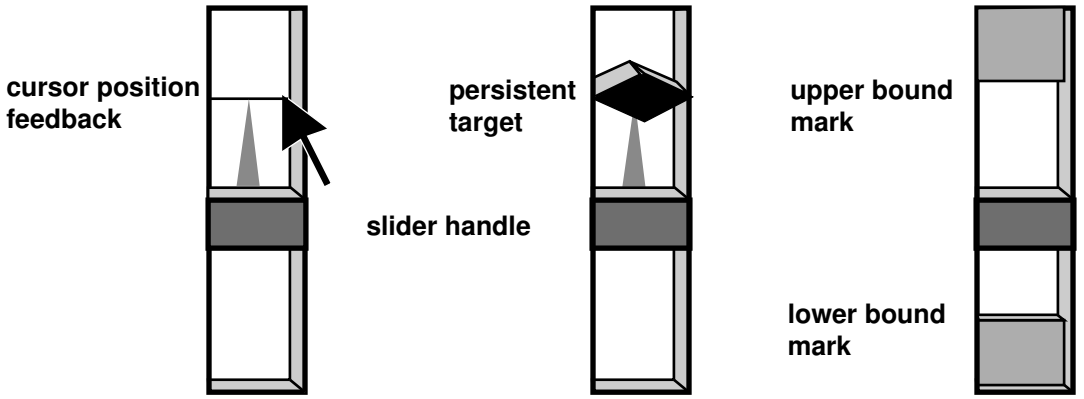

Figure 8.6: A differential slider. Left: feedback shows not only the value specified by specifying dragging, but also the actual value of the attached connector. Middle: the `GoTowards` controller used to drive the connector toward the specified value can be retained after dragging is completed to nail the value in place. The persistent value is displayed by the diamond. Right: users can place boundary constraints with the sliders. Upper and lower bounds are displayed as greyed regions.

might change when a point is grabbed is important for controlling things like shadows, where many things might be affected by a single control. Some mechanism for permitting the user to specify the desired behavior in under-constrained cases, for example by permitting grabbable points or objects to be nailed in place, needs to be provided.

Differential Sliders

To present controllers and connectors the user directly there needs to be a mechanism for controlling any connector output, even ones whose output is simply a dimensionless scalar value. The mechanism must permit the user to specify the various controller types, and provide values for those types that require them. Bramble has a widget called *differential slider* that is a variant of a standard slider designed to control a connector with the differential approach. A basic differential slider is shown in Figure 8.6. Better designs that are more self-revealing and easier to use are certainly possible.

A standard slider converts the position of the mouse into a value for its attached parameter. For the differential approach, such a slider cannot be attached to a connector since the value cannot be set directly. To create a basic differential slider, a `GoTowards` controller is connected to drive the value towards what is specified by the mouse, rather than directly specifying new values for its attached parameter when dragged. Because the connector's value may not perfectly track that of the mouse, both values are displayed to the user.

The ability to nail a slider's value in place is useful feature for a differential slider.

Such a facility can be implemented by simply retaining the `GoTowards` controller beyond the completion of the dragging operation. With the Bramble differential slider, this is done by holding a modifier key as the dragging button is released. Key presses can be used to create controllers that drive the slider to its minimum, maximum, or middle value.

A differential slider can be used to apply other controllers as well. For example, the Bramble slider can place boundaries on the value by pressing the third mouse button either above or below the present value. The slider displays the current boundaries, as shown in Figure 8.6. Snaps can also be placed on the slider's value. The slider must display where the snaps are and provide feedback for when the slider is snapped.

8.2.2 Drawing or Modelling with Constraints

Drawing or modelling with constraints can be seen as a variant of directly providing the abstractions of the differential approach to the user. In such an approach, a user declaratively specifies relationships among parts of the model. Differentially, this entails creating `GoTowards` controllers for selected attributes that compute the relationships. Often, this is coupled with dragging: the user can drag pieces of the drawing either by placing and dragging positional constraints, or dragging the model subject to the constraints.

The idea of using constraints in interactive drawing and modeling dates back to Sketchpad[Sut63], the earliest system. Since then, there has been considerable interest in the approach of declaratively specifying parts of the drawing. Some of the advantages of the constraint-based approach are:

- the user can specify what is most convenient, in any order;
- the constraints can give structure to the model, potentially embedding the semantics of the thing being modelled;
- the constraints can be used to specify exact relationships in the model precisely;
- the constraints are persistent so they maintain previously established relationships to avoid redundant work in reestablishing them after editing.

Many difficult issues have limited the success of constraint-based systems for drawing. Not only must a system be able to solve constraint satisfaction problems, but it must make it easy for users to specify, debug, and edit constrained models. Constraints change the nature of interaction in a graphical application. Without them, actions only affect the objects to which they refer. For example, dragging an object in a traditional drawing program moves only the object. With constraints, this locality is lost: altering one object may cause other objects to be affected. This global nature of constraint operations is at the core of many of the difficult issues in employing constraints.

Without user specified constraints, graphical objects have fixed behaviors. For instance, an ellipse in a drawing program behaves like an ellipse. The system designer can design a good, usable behavior which the user can learn and apply to all ellipses. When user specified constraints among objects are introduced, the situation changes. To begin with, the behaviors can become more complicated because of interactions among objects. Each combination of objects and constraints will have its own behavior. These behaviors are specified by the user in terms of the constraints; the user is effectively programming.

As in more traditional programming, complexity in the constrained behavior of a graphical model becomes a problem when it has bugs, e.g. when the behavior is not what is desired or expected. The most obvious form of bug is when the constraints force the model into a configuration that is not what the user desires, or the constraints prevent the user from achieving a desired configuration. Another class of constraint bug stems from bad constraints where solutions cannot be found, either because of conflicting specifications or solver failures.

Because constraint errors occur, interactive graphical applications which provide constraints to users must deal gracefully with bad situations, such as conflicting or redundant constraints. Underdetermined models must also be handled, as it is impractical to expect the user to specify all possible degrees of freedom. Because of the potential for errors, it is crucial to aid the user in understanding the complex behaviors of constrained models.

The differential approach helps with some of the issues in creating constraint-based applications. Continuous motion facilitates understanding the behavior since users are able to employ their perceptual skills to help understand change. The methods for implementing the differential approach handle under and over determined cases. The dynamic implementation of the differential approach can serve as a backbone to providing a constraint-based system where users can experiment with their models in order to comprehend and debug them.

The Briar drawing program, described in Section 9.1, builds on the differential approach to provide a strategy that addresses other issues in constraint-based tools as well. By using constraints only to maintain existing relationships during direct manipulation, Briar is able to avoid problems with conflicting or unsolvable constraints and unpredictable selection of constraint solutions. A key to making systems like Briar possible is the ability to enforce constraints during continuous-motion, direct-manipulation dragging, as provided by the differential approach.

Although many modern constraint-based systems, such as Briar, show promise in addressing the issues of the approach, it is not certain that the issues can be sufficiently resolved to become the standard and dominant tools. In particular, issues of scalability may be the ultimate Achilles heel for constraint-based drawing and modelling.

8.2.3 Building Interaction Techniques

In the previous sections, we discussed how the abstractions of the differential approach can be presented directly to the user. In this section, we consider leaving them in the hands of the interaction technique designer. The idea is to use combinations of controls in ways that define desirable behaviors for traditional direct manipulation style interactions.

The typical way the differential approach is employed is to use a control to provide interactive dragging, but to define other constraints so the correct behavior occurs. Rather than having the user define the constraints, the interface designer can choose them. Objects or handles are predefined and can be given carefully designed behaviors.

For the designer of direct interaction techniques, the differential approach provides a new set of abstractions. The approach can lead to techniques that could not have been implemented using traditional techniques. Other times, the abstractions are applied to more conventional interactions. In fact, most of the examples given later are simply recreations of existing interaction techniques. The approach may be most practical for prototyping interaction techniques: the technique can be designed using the tools of the differential approach, permitting it to be quickly defined and evaluated. Then the mathematics can be re-derived to compile the interaction technique into a more traditional implementation.

Because controls are used in predefined combinations, many of the most serious drawbacks of the differential approach and constraint-based systems are avoided:

- the number of controls is a small constant that does not grow as the model becomes larger;
- the combination of controls can be checked to insure that they are well behaved before being handed to users;
- the behavior of objects is explicitly designed, rather than just coming about as a byproduct of placing constraints. This can leave the specification of interactive behaviors in the hands of the "trained professionals."

Many of the same arguments are given for the design of the Siri constraint-based programming language [Hor93]. Siri uses constraint techniques within objects to help define their behavior, but uses more conventional methods for inter-object communication.

There are two main ways that the differential approach is used for defining interactive behaviors. One is to define types of objects that behave as desired. The other is to define handles that have particular functionality. This latter, more common category typically involves creating a number of constraints in addition to the mouse attachment during a dragging operation. Many examples are given later in this chapter.

Figure 8.7: A 3D image overlaid on top of a corresponding real image. Our goal is to place the virtual plant on the real table. In each image, there is a table (denoted by the white rectangle for the graphics image). Initially, the images do not correspond because the virtual camera is not in the same place as the camera used to create the real image.

To define the behavior of a object that will be manipulated using the traditional direct manipulation interface of dragging the position of a single point at a time, the differential approach offers advantages. First, constraints can be used internal to the object to define relationships that must be maintained as the object is dragged. Secondly, by providing a way to implement the dragging, it spares the object designer the effort of mapping from position values to parameter values. It also permits a uniform mechanism to be used to drag all points on all objects.

8.2.4 A Concrete Example: Aligning a Camera to an Image

We now consider a very specific interaction technique as an example, and discuss how it can fit in with the strategies of this section. The problem is to align a synthetic image of a 3D scene with a real image by configuring the virtual camera creating the synthetic image. We assume that both the real scene and the synthetic scene each have a table of identical size in them. An example is shown in Figure 8.7.

With through-the-lens controls and the differential approach, the correspondence problem can be solved easily. Through-the-lens controls are used to drag the corners of the virtual table to their corresponding positions in the image, causing the camera to be moved. Each control is locked as it is placed. Manipulating the positions of the four corners of the table causes the virtual camera to be placed in a position where the two

Figure 8.8: The synthetic and real images are registered by successively dragging the corners of the table to their corresponding positions in the image.

images correspond. This process is shown in Figure 8.8.

The registration task is an excellent example of the benefits of the differential approach. It solves an important and useful task that would be extremely difficult with traditional interaction techniques. In order to define the interaction technique without the differential approach, the mathematics to determine the camera configuration from the corners of the rectangle would have to be derived. While such a derivation is possible, it would be extremely difficult to do in a robust manner as it is an overdetermined problem. With the differential approach, it is a straightforward combination of four 2D point controls. The method over-determines the solution, specifying eight controls for a camera which has only seven degrees of freedom. However, the methods of Chapter 3 can handle these over-determined cases.

The table registration required the use of four through-the-lens point controls. Those controls could have been specified either by the user, or by an interaction technique designer building a mechanism for the particular task of image registration. If the abstractions of the differential approach were directly provided to the user, the registration task would require the user to freeze the position of the table in the world, enable the camera to be controlled, select through-the-lens position controls for each corner of the table, and then drag these controls. In a more constraint-based interface style, there might be a command for dragging points with through the lens controls, which the user could successively apply to each corner of the table.

Because image registration is an important task, an interaction technique designer might want design an interface for it. For example, the designer might create the virtual table as a special object that when its corners are grabbed, the system knows to apply a through-the-lens control on the point to manipulate the camera, and to leave these controls locked in place even after they've been manipulated. The user would see a command to create the alignment table, and would directly manipulate table corners, but the abstractions of the differential approach would be hidden, serving simply as a tool for the interaction technique designer.

8.3 Sources of Constraints

The basic idea behind each of the strategies of the previous section is to provide constraints so that manipulation operations like dragging have the desired effects. The major difference in the strategies is how these constraints were specified, whether they were provided by the interaction technique designer or by the user. In this section, we consider what constraints can be useful for manipulation, and how they are used in conjunction with dragging to create interaction techniques. This section is organized by "sources" of constraints, the kinds of things that specify constraints on manipulation.

The general problem in manipulation will be to handle the underdetermined nature of dragging. Our input devices will undoubtedly specify far fewer degrees of freedom than the model contains, or even than typical objects contain. For example we might control an articulated figure with a 6 degree of freedom tracker or control a 3D position with a mouse.

The large number of degrees of freedom problem is especially important in 3D manipulation. Unlike in the real world, the objects we manipulate with a 3D user interface can float freely in space, and therefore have extra degrees of freedom. The problem of 3D manipulation is to somehow control these degrees of freedom with the limited input devices we have. One approach to this problem is to develop better input devices. For the work in this thesis, I have considered only the mouse. However, the differential approach is input-device-independent, and even with better input devices, the constraint-based approach applies – in the real world we still use constraints for manipulation even with our dextrous hands.

When we encounter an object with many degrees of freedom in the real world we employ a variety of tactics to manipulate it. We use manipulators in parallel, for example by coordinating our hands or fingers, or we create constraints, either by using an extra hand or finger or by using interactions between objects. For example, we might turn 3D problems into a 2D one by placing the objects on a flat surface, or for more complex manipulations we might build a jig to limit the objects' behaviors to make it easier to achieve the desired manipulations.

However, we normally design our objects so that they don't have as many degrees of freedom. The behaviors of most objects we manipulate are constrained by their relationships with other objects or their structure constrains them to move only in the correct fashion. For example, operating a door requires only one degree of freedom (its hinge) or two (its knob and hinge) and car steering wheels and levers rotate only in useful ways.

These real world tactics for manipulation all rely on constraints on object motion to simplify the manipulation problem. The sources of these constraints vary: they come from the mechanical structure of the object, interactions among objects, conventions on the uses of objects, or from other hands or fingers. In this section, we will use the same approach to address the problem of manipulation in interfaces. The general idea

for creating an interaction technique will be to provide a sufficient set of constraints so that control of a single point defines a desired behavior. For defining 3D interaction techniques using the mouse, we will attempt to define a sufficient set of constraints so that the two degrees of freedom specified by a through-the-lens control are sufficient to control the 3D object.

8.3.1 Intrinsically Constrained Problems

For some manipulation tasks, the object being manipulated is intrinsically constrained sufficiently. For example, even in a 3D world, dragging a point along a fixed plane, such as the floor, is a 2D problem. So, if a point is by definition in a fixed plane, it is sufficiently constrained so that a 2D input device can be used to control it. An example of such a point would be shadows on the floor. This is part of the reason for the interest in shadow manipulation.

8.3.2 Artificial Constraints

A common way to create interactions is to create synthetic constraints based on some user command. For example, based on the state of a modifier key or a command mode, the system might constrain object motions to rotate about a particular axis or translate in a particular direction.

An example 3D interaction technique using artificial constraints is the mousepole, introduced in Section 7.8. The mousepole allows the user to position a point in 3D using the mouse by constraining the point to lie in the plane parallel to the ground. The ray cast from the mouse position defines a unique point on this plane. Depressing a button switches the mousepole to operate in a plane perpendicular to the ground. We draw a vertical pole from the point to the ground in order to indicate height, hence the name mousepole.

The advantages of these artificial constraints are that they can be applied to any object and are easy to create. However, it is easy to make such interfaces complicated by including large numbers of modes, modifier keys, and commands. It can be difficult to make such interfaces self-revealing. Using a standard of artificial constraints on all objects has the advantage of uniformity, but has the problem that it does not permit objects to special behavior.

8.3.3 Object Semantic Constraints

The graphical objects that we manipulate often have structure or semantics that may dictate how they should behave when manipulated. This behavior can often be expressed as constraints, sometimes corresponding to the mechanical structure of the object, and sometimes to the intended purpose or conventions of use of the objects. For

example, the Luxo lamp of the introduction has many pieces each with many degrees of freedom. However, the mechanical structure of the lamp significantly reduces the degrees of freedom in the lamp, making it much easier to control.

Many of the objects we must manipulate have only a few degrees of freedom: useless motions are constrained away. There are many example of objects whose behavior is sufficiently constrained so that one or two controls are sufficient, for example, steering wheels, airplane yokes, doors, levers, or the handle of a slot machine. There are some implied assumptions in these manipulations, for example when the handle of a slot machine is grabbed, we assume that we are attempting to pull the lever, not move the machine.

Sometimes, the constraints on objects stem not from the objects' mechanical structure, but rather from conventions on how they are used. For example in moving a piece of furniture, we are most typically interested in sliding it along the floor or turning it, not necessarily lifting it or flipping it over. In such cases, it is conceivable to build interfaces which imbue the objects with the constraints that cause the more common behavior, and require some less direct method for other manipulations. In analogy to the real world, I can push the furniture around on my floor, but to lift it or flip it, I need to get some extra help. The desire to perform non-standard manipulations also exists for mechanically constrained objects, for example, we might want to rip the head off the Luxo lamp.

Often, the manipulation task provides sufficient information to adequately constrain objects so that simple point manipulations suffice. For example, positioning a picture on a wall or a lamp on a desk are both 2D problems. Pulling the arm of a slot machine is a one dimensional problem if the model behaves like a slot machine. I call the strategy of attempting to use the natural constraints of the task to create interaction techniques *manipulation from structure.* Such constraints are called "context specific constraints" by [Hou92]. These constraints that arise from the structure of the model are advantageous because they only restrict the user from performing operations which are typically undesirable, they are often already inherent in the parameterization of the model, and they are easily understood by the user.

There are many issues in employing manipulation from structure in realistic systems. For one, it requires that the system have some knowledge of what the objects are, and how they should behave — a collection of polygons might look like a painting which should remain hung on a wall to the user, but the system must not only know how paintings are to behave, but also how to identify them. The structure of the model must be created in a manner that has the proper semantics. When manipulating such models, it can be difficult to know if the model is sufficiently constrained such that 2D manipulations will provide enough control. While the differential approach does not directly address these problems, it does make it far easier to explore manipulation from structure by providing a vocabulary with which the semantic constraints can be expressed and by mapping controls to whatever parameters are used to represent the

models.

8.3.4 Handles

Handles are particular points associated with objects that can be grabbed. Handles are often given specific meanings as to the manipulation that they perform. For example, in a typical Macintosh direct manipulation interface, a graphical object has particular handles that cause it to be scaled.

Handle behavior can be defined by associating each handle with a set of constraints that are applied to the object as the object is manipulated. For example, to create the Macintosh-style scale handles, the position of the opposite corner of the object must be pinned down while the handle is dragged. Bramble's grab and ungrab hooks were specifically designed to help define handle behaviors with constraints.

8.3.5 Widgets

Although building the constraints required for manipulation into each graphical object has many attractions, it does have some serious drawbacks. Most obvious is that behavior must be built into each object, and that each object must have sufficiently well-designed behavior so that it is manipulable and that the constraints are apparent to the user.

An alternative to giving every object a behavior is to define special objects which have behaviors and to manipulate the other objects by attaching them to these *widgets*[2]. These widget objects can be specially designed to have desirable and self-revealing behaviors. A widget can be thought of as a tool for providing a specific type of manipulation. Widgets have become ubiquitous in graphical interfaces. Most graphical user interfaces now have users specifying values with widgets such as sliders, dials, and buttons.

With the differential approach, connections are bidirectional. If a widget displays a value, it can also serve to control the value. The functions that map from the screen position of the input device to the values stored by the widget do not need to be provided, only the forward direction functions required for drawing. This is exemplified in the fuel gauge widget example shown in Figure 8.9. The complete code for this example is presented in Section A.2.1, but a simplified version is defined using the Bramble `define-shape` function:

[2]The definition of widget here is slightly different than that of [CSH+92]

Figure 8.9: A fuel gauge, define with Bramble's `define-shape` function.

```
(define-shape gas-gauge (val) (0)          ; object with 1 variable
  ((t (+ .2 (* 2.7 val)))                  ; convert percentage to angle
   (x (- 0 (* .3 (cos t))))                ; x position (left is 0)
   (y (* .3 (sin t))))                     ; y position
  (drawf                                   ; define draw method
   (prog (color gl-black) (linewidth 3)    ; set color and line width
         (arc 0 0 .3 0 1800)               ; draw the shape of the gauge
         (move -.3 0) (draw .3 0)          ;     and the bottom
         (move 0 0) (draw x y) ))          ; draw the needle
  (> val 0)                                ; limit the needle to valid
  (< val 1)                                ;     range
  (handle x y) ))                          ; put  a handle on the needle
```

The values that are computed for the position of the handle tip, used for drawing, also serve as a handle. No code for mapping from the mouse position to the angular value was required.

Bramble provides a small set of widgets that can be used in windows outside of the views. However, widgets are often created as graphical objects that exist in the world with the user objects. Because Bramble's world is 3D, the widgets actually exist in 3D space, even though they are usually flat onto the $z = 0$ plane. However, since the widgets are 3D objects they can be transformed into other places. For example, we might place gauges on a model of an instrument panel as shown in Figure 8.10.

The behavior of a widget can be defined using the same techniques as used for other objects. For example, to create a gauge with a dial, the arrow could be constrained to have its endpoint at the center of the dial, and its orientation equal to the value of the widget.

An example of some experimental widgets in Bramble are the aircraft gauges. The set of gauges include an altimeter, a heading indicator, and an artificial horizon, shown in Figure 8.10. In each case, the gauges draw themselves based on the values of their parameters, and provide `DistinguishedPoint` connectors on their moving parts. For example, either hand of the altimeter or many points on the heading indicator's compass ring can be grabbed.

This set of airplane gauges is potentially interesting because it provides a 2D display, and therefore control, of the orientation of a 3D object. However, I have not found

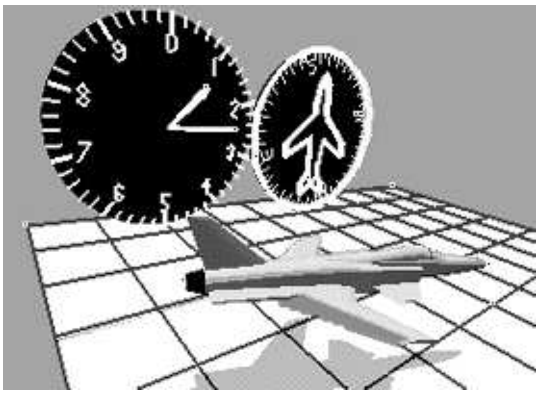

Figure 8.10: Gauges serve as both displays and controls of the plane's configuration. Although they are 2D objects, the gauges can be placed anywhere in the 3D scene.

the airplane gauges to be a useful interaction technique for controlling 3D objects.

8.3.6 3D Widgets

Traditional 2D widgets can be applied to 3D interfaces. To avoid the drawbacks of such an approach, [CSH+92] introduces the notion of 3D widgets as tools in the object's space.

The differential approach offers a mechanism for defining the behavior of 3D widgets, which is a central difficulty in their design[ZHR+93]. With our approach, we define the behavior of a 3D widget with a set of constraints that sufficiently restrict its behavior so that a through-the-lens control can be used. These constraints are applied while the widget is operated.

Figure 8.11 shows some example 3D widgets modelled after the ones presented in [CSH+92] and [SC92]. The "jack" widget on the left is used to translate an object along an axis. Pulling a tip of the jack causes the widget and its attached object to move in the direction shown by the arrow. When a handle is grabbed by pressing the mouse button, the object is constrained so it can only translate along the axis, and a through-the-lens control is applied to the handle. When the mouse is released, the constraints are removed. Similar techniques are used to create widgets for rotation about a point or axis, or to scale either uniformly or along an axis. Better graphic design can make the widgets more self-revealing.

Widgets to control the viewpoint, such as a virtual sphere, can also be created with the differential approach. The exact virtual sphere technique of [CMS88] is harder to create using the building blocks provided because the sphere that the user grabs is in screen space, rather than in object space. This has a number of disadvantages as it loses the kinesthetic coupling between dragging and the visible behavior. The arcball method of [Sho92] makes this problem worse by introducing a scaling factor to simplify the

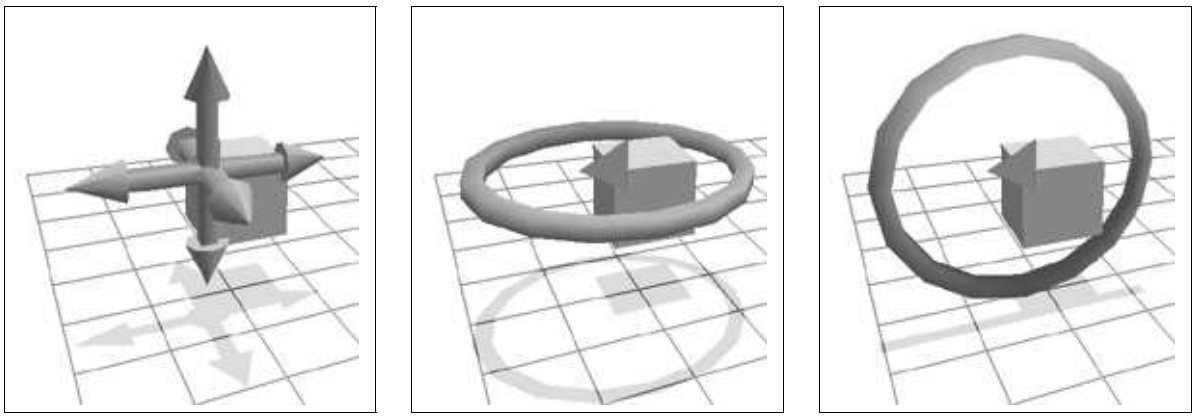

Figure 8.11: 3D Widgets for translating an object, rotating an object about a vertical axis, and rotating about a horizontal axis. The object is made semi-transparent to avoid obscuring the widget. Notice that since the widget is an actual scene object, it casts a shadow on the floor.

derivation of the mathematics.

A virtual sphere variant that we prefer is the virtual trackball. The virtual trackball can be thought of as a glass sphere surrounding an object in the world. To rotate an object, the sphere is grabbed with a through the lens control and is constrained so it can rotate only around its center. To make the manipulation of the object more "direct," the sphere is often omitted. Points on the object are grabbed through the lens, the object's center of rotation is nailed in place, and the object is made rigid so it revolves around the center. One addition to the virtual trackball is the addition of an elevator key that freezes the rotation but frees the uniform scaling of the object. This causes the virtual trackball to work like a spherical coordinate mousepole.

To use the virtual trackball to control the viewpoint, we consider the objects in the world to be encased in the sphere that is dragged. In practice, when the through the lens control is applied, the distance between the camera and the center of the world is constrained, as is the apparent position of the center on the screen. This virtual trackball behavior is part of the standard Bramble 3D interface of Section 7.8. Whenever a corner of the groundplane is selected, the trackball behavior is used. A dollying (sometimes incorrectly referred to as panning) behavior, created by allowing the camera to translate but not rotate, occurs when the center of the groundplane is dragged.

There are several advantages to developing a 3D widget with the differential approach and through-the-lens controls. First, it permits defining the widget's behavior without deriving any of the mathematics for converting from the input device's motion to parameter changes. Secondly, it defines the widget in a manner that is independent of the underlying representation of the object. Finally, the descriptions of the widget in terms of constraints provides a concise, executable specification of the widget's behavior.

8.3.7 Discussion

The differential approach lets us define and implement a variety of strategies for manipulating objects. The strategies are not mutually exclusive, for example we might attach a widget to a part of a constrained object (particularly if it is not sufficiently constrained) or use modifier keys to alter the behavior of a widget. As we experiment with different strategies, we find that they all have advantages and drawbacks. Here, we discuss some considerations.

User provided vs. designer provided: Making the user responsible for providing constraints may give the user more flexibility, but might also make them expend more effort as they specify behavior in addition to geometry. It also exposes them to the range of problems inherent in specifying behavior. Interface designers are (hopefully) better at devising good interactive behaviors, but cannot tailor designs to specific operations. If the objects to be manipulated will be used often, or replicated many times, it becomes worthwhile to spend effort in building behavior into them. This might be done by the user, the interface designer, or by someone else who is simply an object designer.

Smart objects vs. smart tools: Do we put lots of behavior into the objects, so that they can be manipulated with simpler tools, or do we develop better tools so that our objects need less behavior? An extreme case of the former would be an interface where every object had sufficiently well defined behavior so that any point could simply be grabbed with a though-the-lens control. An extreme case of the latter would be to have uniformly simple objects, for example ones that are being subject to the standard translate, rotate, scale transformations, and provide a set of widgets or commands which operate on them.

Context-sensitive vs. context-free: Is object behavior uniform, or do different objects (or even parts of objects) behave differently? Tailoring behaviors to objects has advantages, as discussed by [Hou92], but uniformity does too.

Bounded scope vs. unlimited connection: Permitting arbitrary relationships among objects can simplify manipulation by restricting unwanted behaviors, but when many objects interact, their coordinated behaviors become complicated as longer chains of causality are possible. Strategies which manipulate single (or a small number of) objects, such as widgets, have the advantage of keeping their simple behavior as the environment scales, but may grow tedious to use as the user must consider an increasing number of objects and interactions.

8.4 Employing Switching

In Section 6.4, controllers that operated by switching simpler controllers on and off were presented. In this section, we examine how these switching controller might be used.

8.4.1 Generalized Snapping

In Section 6.4.2, a controller for snapping to values was developed to aid in providing accurate direct manipulation. The `Snap` controller causes a connector to be driven to a particular value when it approaches that value. Because the controller can be attached to any connector, it provides a general method of snapping and its use is, therefore, referred to as generalized snapping.

Generalized snapping can be used to recreate the typical cursor snapping by representing the cursor as a differentially controlled object and creating `Snap` controllers that drive it towards desired snap targets.

Other behaviors can be created by placing `Snap` controllers on connectors other than those that are being directly manipulated. Because controller handling happens asynchronously in parallel with manipulation, precise relationships can be established away from where the cursor is. For example, if users often desire line segments to be accurately vertical, that is when something appears vertical it should be vertical, `Snap` controllers for the values $\pi/4$ and $3\pi/4$ can be placed on the angle connectors of each line segment. If a line segment is brought near being vertical as it is dragged, the configuration of the line segment would "snap" to a configuration where it was vertical. While this small jump may violate the continuous motion, it is usually small and predictable enough that it is permissible, especially when proper feedback is provided to express what snapping operation has occurred.

With generalized snapping, the snapping can occur away from the dragging action. For example, if the line segment of the preceding paragraph was attached to some linkage mechanism, the user might be controlling the line segment by manipulating some other part of the mechanism, as shown in Figure 8.12. This makes it even more crucial to provide proper feedback to the user to denote when something is snapped. Because the focus of attention is possibly away from the snap, for example at the dragging site, auditory cues might be useful as well.

The generalized snapping technique has many drawbacks that need to be resolved. First, it does not revert objects to their original configuration when unsnapped. Secondly, it needs to be connected to mechanisms that handle multiple snap targets. To do this properly might require the ability to determine whether two constraints conflict. Third, because the differential approach only drives connectors towards particular values, there is no guarantee that the snap controller accurately reaches its target. Finally, the snap controller has a number of potentially sensitive parameters that must be fine-

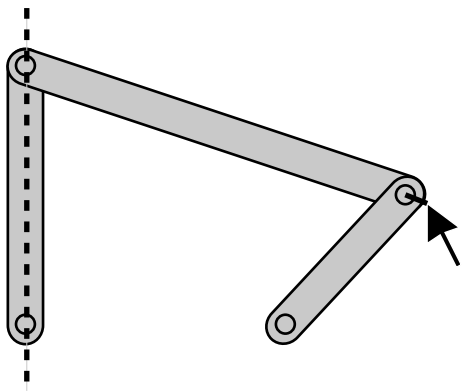

Figure 8.12: Generalized snapping can occur away from the dragging action. As the user drags the linkage mechanism, the left line segment gets snapped to the vertical position. Because such snapping can take place away from the focus of attention, it is important to provide proper feedback.

tuned to provide the correct "feel."

8.4.2 Collisions

Non-interpenetration is a useful constraint on objects. Such a constraint causes objects to collide and contact one another when they touch, rather than to simply pass through. A collision is the initial impact between two objects, which may remain in contact with one another afterwards.

With the differential approach, it is possible to create non-interpenetration constraints using inequalities. While the methods are not as sophisticated as the special purpose collision simulation methods reviewed in Section 2.2.3, they are simple to build with the abstractions of the differential approach, and are sufficiently effective to be interesting.

A particularly simple case of collision avoidance is used in box-and-arrow diagram creation, an application discussed in Section 9.3. The particular constraint we would like to enforce is that two axis-aligned rectangles do not overlap. This can be expressed as a disjunction: if at least one of the following inequalities hold, then the rectangles do not overlap:

$$\begin{aligned} x_1 + s_x &< x_2 - s_x \\ x_1 - s_x &> x_2 + s_x \\ y_1 + s_y &< y_2 - s_y \end{aligned} \tag{8.10}$$

$$y_1 - s_y \quad > \quad y_2 + s_y,$$

where x and y are the coordinates of the center of each rectangle, and s is the size. Each clause represents one of the possible ways for the rectangles to be separated, either 1 is above 2, below 2, left of 2, or right of 2. Alternatively, the disjunction can be expressed as a maximum operation of the four terms.

The differential approach permits conjunctions of constraints to be handled easily, however disjunctions are more difficult. Enforcing the disjunction is not a problem when it is true, as each term can be checked until one that is true is found. However, when none of the expressions of the disjunction is true, a choice must be made. As with inequalities, when the constraint is violated it will be activated in order to "pull" it back to the admissible region. In the case of the disjunction, at least one of the expressions must be chosen and used as a constraint. Even though all of the inequalities are violated, all cannot be enabled because they would conflict.

The strategy for handling disjunctions of inequalities will be to select one of the inequalities to enable when the disjunction is violated. Because we are not backing up the instant where the disjunction began to fail (see Section 6.4.5), we must resort to enabling a violated inequality and permitting it to pull the configuration back to a legal configuration. The difficulty is selecting the inequality, which must be done by a heuristic. Although we would like to pull things out the same way they went in, the lack of history makes it impossible to do exactly.

A heuristic used for the simple rectangle overlap problem is to pick the inequality that is least violated. This technique is used for object non-overlap in the box and arrow diagram editor program of Section 9.3. The heuristic fails in two cases depicted in Figure 8.13. First, if the overlap is near a corner, an object may be pushed out to the side, rather than the way that it came in. Second, if the ODE solver step moves the object too far through the second object, it might be closer for the object to push out the opposite side that it came in from.

The basic rectangle non-overlap can be extended to the general case of rigid (e.g. rotatable) convex polygons. The basic element of a non-overlap constraint for two such objects is a point outside of polygon constraint. This is created by a disjunction, there must be at least one edge of the polygon that the point's distance inside the half-plane defined by the edge is positive. A point outside polygon constraint can be created with a similar heuristic as the rectangle non-overlap, with similar selection problems.

Two convex polygons do not overlap if and only if all the points on each are outside of the other. This makes it easy to break the polygon collision problem into point outside of polygon problems. What a polygon collider must do is find points that are inside the other polygon, called contact points, and enable constraints to push them outside. It will always be sufficient to choose at most two contact points, as this will establish a dividing edge between the polygons. After being selected, each of the contact points chooses an edge to be pulled out through. The heuristic must make sure that the points both exit through the same edge, even if they are on different objects.

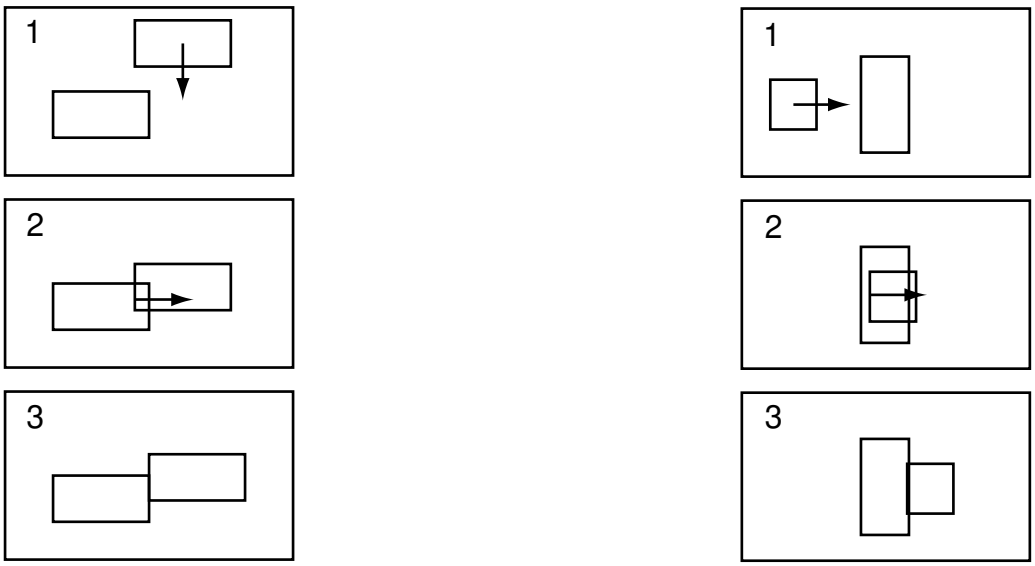

Figure 8.13: A disjunction of inequalities constrains two axis-aligned rectangles from overlapping. Because the selection of a direction to pull the block out is made without knowledge of the direction it came in, two types of errors may be caused. If the overlap is in a corner, the distance out the side may be less than the distance back out the top, as shown in the sequence on the left. Also, if the object moves too quickly it may pass through the other object, as shown in the sequence on the right.

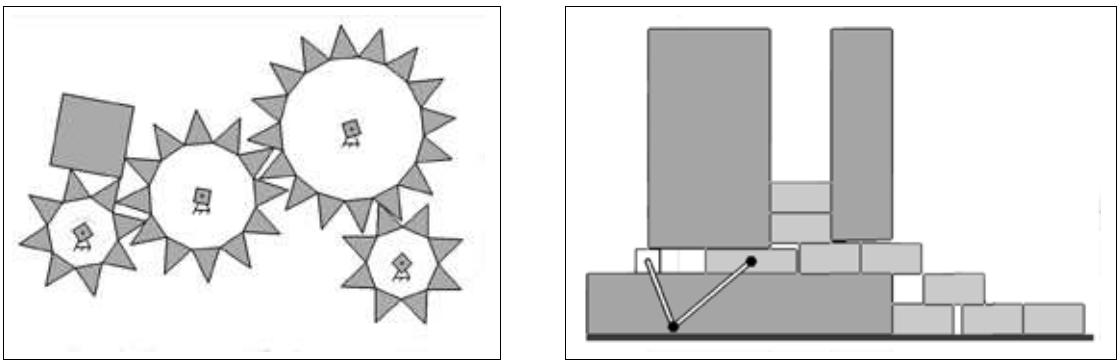

Figure 8.14: Mechanisms that can simulated using the simple collisions methods of the differential approach. The left shows a set of sawtooth "gears" jammed with a square block, and the right shows a block feeder.

While this simple method for collisions is not perfect, it is extremely simple to implement, and achieves good enough performance to allow some interesting mechanisms to be created with it. Two examples are shown in Figure 8.14. Since these methods were developed, techniques to permit robust simulation of collisions interactively have been developed by David Baraff [Bar94].

The philosopher's have, in many ways, tried to interpret the World. The point, however, is to change it.

— Karl Marx
11th thesis on Feurbach

Chapter 9

Example Applications

This chapter describes some sample applications built with the differential approach and the tools developed in the previous chapters. The purpose of discussing these applications is threefold:

- to show how the interaction techniques developed with the differential approach can be fit in the context of a realistic application.
- to show that the differential approach is viable.
- to show some of the range of the tools, particularly the Bramble toolkit, giving evidence of how the differential approach can be used to make such tools more general.

All of the example applications discussed, except for the Briar drawing program, were constructed using the Bramble toolkit.

9.1 A Drawing Program

Although constraint-based techniques have been used since Sketchpad [Sut63], the earliest drawing program, they have not been generally successful. Many difficult issues have limited the success of constraint-based systems for drawing. Not only must a system be able to solve constraint satisfaction problems, but it must make it easy for users to specify, debug, and edit constrained models. The *Briar*[1] drawing program attempts to address all of these issues. The issues of constraint-based drawing that Briar addresses apply more generally to interfaces built with the differential approach. Briar is discussed in detail in [GW94] and [Gle92a], and illustrated in Figure 9.1.

A major goal in the development of Briar was to build a system with constraints that provided users with the fluent interface that they have come to expect from direct

[1] It is called Briar because, like the plant it is named for, things stick together inside of it.

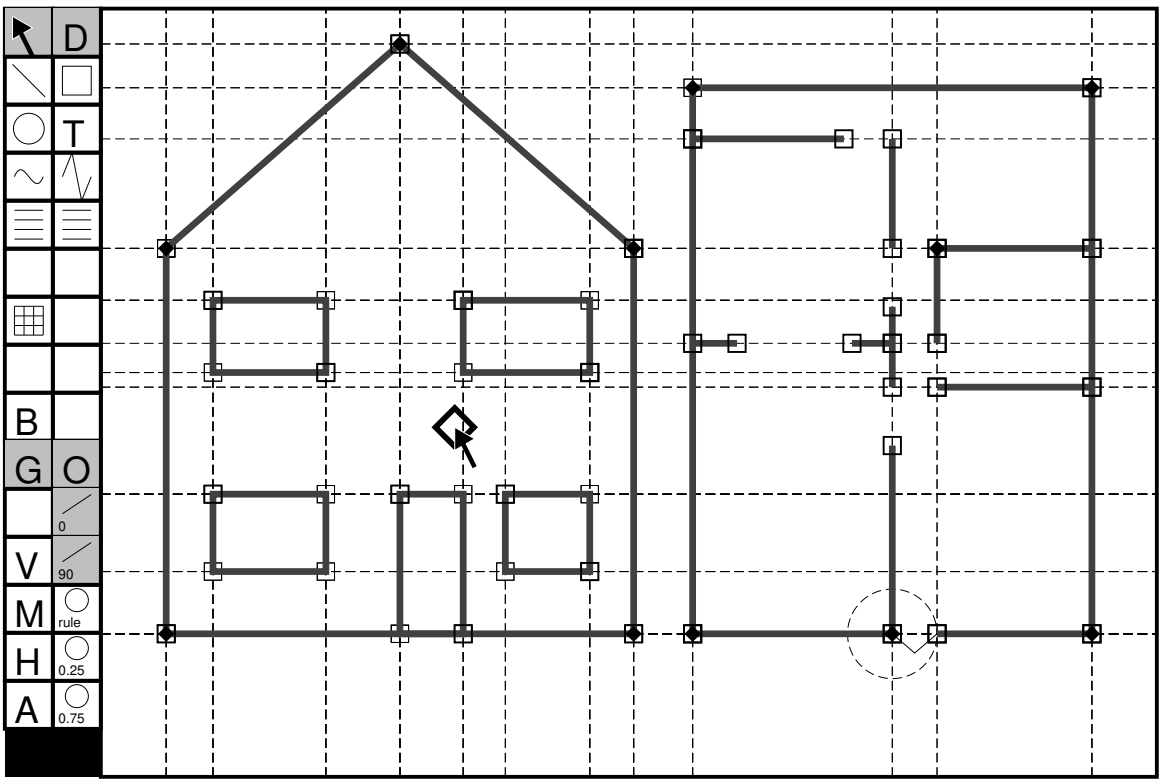

Figure 9.1: Briar editing a constrained drawing.

manipulation drawing programs. The techniques in Briar aim to add at least some of the advantages of constraints without detracting from what has made direct manipulation drawing programs so successful. In short, they aim to make Briar a direct manipulation drawing program augmented with constraints, not a drawing program with a primarily constraint-based interface. The interface feels similar to other direct manipulation drawing programs which provide snapping except that once snapped, things can stick together.

The basic idea behind Briar is to enhance an existing, successful, direct-manipulation drawing technique with constraints. Briar separates the task of initially establishing relationships in drawings from that of maintaining them during subsequent editing. It uses snap-dragging, a successful non-constraint-based technique introduced in Gargoyle [BS86], for initially establishing relationships in drawings. By augmenting snap-dragging, Briar obtains the constraint specification with little or no additional effort from the user. The methods of the differential approach allow the constrained drawings to be be manipulated directly; as the user drags an object, constraint techniques adjust other objects to maintain relationships.

Combining snap-dragging with constraint techniques significantly changes the nature of the constraints. Unlike most previous constraint-based approaches that rely on solving methods to initially satisfy the relationships, constraint methods in Briar are used only to maintain relationships during dragging. This permits using the differential approach, as objects move only during continuous-motion dragging. The primary benefits of using constraints only to maintain constraints during dragging parallel those that motivated the differential approach in Section 1.2: we can avoid solving non-linear equations from arbitrary starting points; we need not select configurations in under-constrained cases, and we can aid the user in the problem of understanding state transitions by always providing continuous motion.

In Section 8.2.2 discusses many of the challenges in creating constraint-based drawing editors. The differential approach addresses some of these. Briar includes techniques designed to address many others including how to specify constraints, how to edit them, and how to display them to the user. These issues arise in a wide range of applications. This section describes Briar in detail to illustrate some possible solutions to these problems.

9.1.1 Augmented Drawing Tools

Briar's approach only uses constraint techniques after relationships are already established. To establish the relationships initially, we use techniques such as grids, gravity, and snap-dragging [BS86] that have been employed by non-constraint based drawing programs. To avoid giving the user the extra work of specifying both the constraints and an initial solution, Briar provides *augmented snap-dragging*. Augmented snap-dragging is a variant of snap-dragging that has been extended to specify persistent con-

straints as well as positions. The basic idea is that the cursor placement operations of snap-dragging contain information about why an object was positioned where it was, and therefore they can also provide a constraint specification in addition to positional information. The technique can also aid traditional constraint-based drawing programs which require good starting points to prevent long jumps that are hard for solvers to make and users to understand.

Ours is not the first attempt to spare users from additional effort required to explicitly specify constraints. Previous systems have attempted to infer relationships after drawing operations by looking at the resulting drawing, as in automatic beautification [PW85], sequences of drawings, as in Chimera's snapshot mechanism [KF93], or at a trace of user actions, as in Metamouse [MKW89]. Because this information typically does not specify the relationships unambiguously, these systems relied on heuristics or asked the user to resolve the ambiguity, as in Peridot [MB86] and Druid [SKN90]. Our approach provides positioning methods which unambiguously specify constraints, eliminating the need for inferences. We simply augment drawing tools to specify constraints as well.

In Briar, augmented snap-dragging is the only method for specifying constraints. It provides a uniform method for creating a variety of constraints, such as controlling distances, positions and orientations. Other systems which infer constraints from snapping either have a limited vocabulary of constraints, such as the Manhattan gridding rules of the interface builder of [HY91], or use other methods to specify the complete set of constraints, as in Intellidraw [Ald92] and DesignView [Com92]. Rockit [KLW92] also infers gridding constraints from drawing actions, but does not avoid ambiguity, actually averaging multiple possibilities. Chimera [Kur93] has both snap-dragging and constraints, but does not integrate the two.

Here is the basic idea behind snap-dragging. When drawing, it is difficult to position a pencil precisely without using some form of aid. Similarly, it is difficult to draw precisely with a mouse or other pointing device unaided. Computer software can provide tools for precise placement by drawing from a software-positioned cursor[2] rather than using the pointing device location. The software cursor's location is influenced by the position of the pointing device, but determined by a function which helps the user position elements precisely.

The uniform grid is the most common function for mapping pointer location to cursor position. It displaces the cursor to points on an equally spaced rectangular grid. "Gravity" is another cursor positioning function. When the pointer is brought sufficiently close to an interesting element in the scene, the cursor snaps to it. The idea of gravity has existed for a long time, having been demonstrated as early as Sketchpad [Sut63]. snap-dragging [Bie89, BS86] enhances the usefulness of gravity. The cursor snaps not only to the edges of objects, but also to interesting points in the scene such

[2]This differs from the original snap-dragging terminology [Bie89] where the position of the hardware pointing device is known as the cursor and the software cursor is known as the caret.

as intersections and vertices of objects. The ability to snap to intersections enables the use of traditional drafting compass-and-straight edge constructions.

Since cursor placement operations contain information about why an object was positioned where it was, they can also provide a constraint specification. Suppose the user, while dragging an object, moves the pointer near another object so that the cursor, and the point being dragged, snap to the second object. This may have been an accident, but the user might have been trying to achieve this relationship. We provide the user with the option of making the relationship persistent, so if it was intentional it can be preserved during subsequent editing. We call the extension of snapping to specify a relationship in addition to a position *augmented snapping.*

When a new relationship is established by snapping, the system acknowledges it by displaying a symbol indicating the constraints that the snapping operation implies. The user can accept the new constraint, by pressing a key to make it persistent, or ignore it. If this automatic constraint generation process works well, the user will want to accept most constraints so the option of making this the default should be provided.[3] In such a mode, it must be easy for the user to reject an action as an accident. In Briar, a key is used to toggle new relationships between the accepted state, in which they are made into persistent constraints, and the ignore state, in which the symbols are removed before the next drawing operation begins.

Snap-Dragging provides two basic operations for positioning points in two dimensions: snapping the cursor to a point, such as a vertex, and snapping the cursor to an object's edge or curve. These operations correspond directly to the constraints "points–coincident" and "point–on–object" respectively. The two constraint types work with each object type that Briar supports. The set of objects presently includes lines, circles, ellipses, and rectangles.

Relations other than contact are created in snap-dragging through *alignment objects:* objects that are not part of the drawing *per se,* but exist only to be snapped to. The original snap-dragging work includes several types of alignment objects, each corresponding to types of relationships which are useful in drawings. For example, the distance from a point can be specified by placing an alignment circle around that point. Other alignment objects include slope lines, angle lines, and distance lines. The usefulness of alignment objects is further enhanced by making them easy to place. In fact, heuristics can often automatically place alignment objects where needed.

The two simple snapping operations combined with alignment objects allow a user to establish a wide variety of relationships. The simple mapping from snaps to constraints extends to a variety of constraints. For instance, snapping to an intersection is a conjunction of the simpler constraints. By using the simple constraints with alignment objects, the relationships specified by snapping to these objects can be made persistent. For example distance-from-point constraints are created by snapping to an alignment circle. The Gargoyle editor [Bie89] shows how snap-dragging can be used to create

[3] In our experience, the automatic constraint generation is so good that we make it the default.

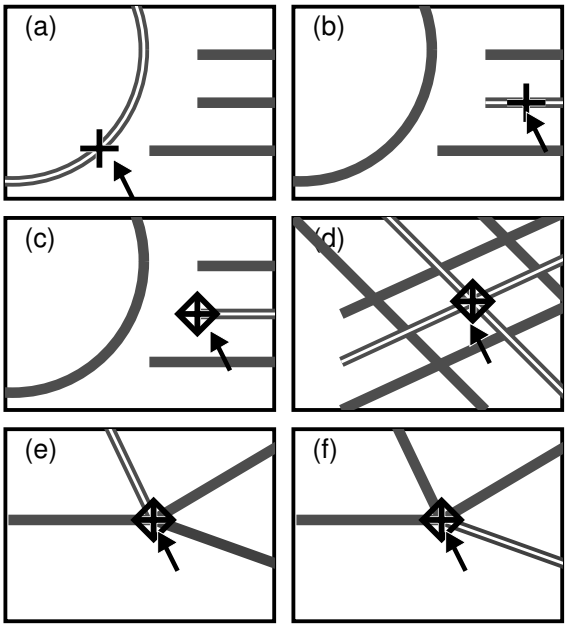

Figure 9.2: Feedback mechanisms display exactly what is snapped to. The cursor changes shape depending on whether it is snapped to a curve or edge (a,b), or a point such as an endpoint or intersection (c,d). The object snapped to is brightly lit. If several objects are close together, the one desired can be selected by cycling (e,f). Color is used for feedback when available.

most of the relationships which are needed in drawings. Augmented snap-dragging extends this to inferring a similarly complete set of constraints.

Hidden state is a fundamental problem in the interface between man and machine; therefore, feedback is an important aspect of any user interface [Nor90]. To make augmented snap-dragging work, feedback is crucial. Our feedback mechanisms (Figure 9.2) ensure that the snapping state is not hidden from the user, by highlighting the object snapped to and changing the cursor's shape to show if it is snapped, and whether it is snapped to a point or an edge. Good feedback makes snapping easier to use because the user never needs to guess what relationships the system is establishing, or which relationships can be made persistent when the snapping operation is complete. The other benefits of snapping feedback [Hud90] also apply as snapping shows the user what can be snapped to, not just what is snapped to.

There are many advantages to augmented snapping as a front-end to a constraint system. Augmented snapping is opportunistic, creating constraints where it can. Constraints are specified with little additional user effort beyond what is required to initially establish them. The constraint creation process is quite transparent so it does not interfere with the user's drawing process. Since snap-dragging is used for all drawing operations, including dragging and creating new objects, constraint creation is always available. Augmented snapping still provides the user with the snapping interface

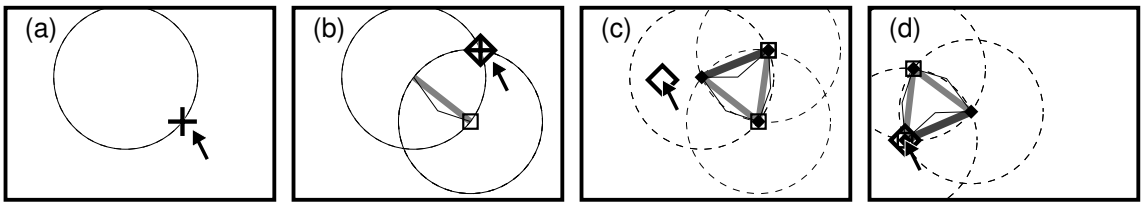

Figure 9.3: Alignment circles help create an equilateral triangle. Briar places 3/4 inch alignment circles around points of interest. Snapping to the circle created with the initial mouse point sets the length of the first line (A). Snapping to the intersection of the circles created around the ends of the newly created line segment constrains the point to be 3/4 inches from each endpoint (b). Snapping the final segment in place leaves three lines constrained to be connected with their endpoints 3/4 inches apart (c). The system can maintain these constraints as as the user drags pieces of the triangle (d).

which has proved successful in systems without constraints, so that constraints need not be used if they are not convenient for a particular drawing operation. Briar augments snap-dragging, which by itself is an extremely powerful drawing tool.

Removing Ambiguity

If multiple objects coincide when snapping, it might not be clear which object to snap to. If we are only using snapping for positioning, the ambiguity is irrelevant: only the target location is important. However, with persistent relations this distinction becomes significant. If the two coincident points are later separated, the correct attachment relationship must be maintained.

Feedback, along with snap-dragging's cycling mechanism, solves the problem of ambiguity. Feedback mechanisms clearly show the user which object is being snapped to and what relationships are being established. If these are incorrect, the user can click the cycle button and the system will snap to the next object within gravity range.

A related problem is that the user might construct a model in a manner which does not convey the desired constraints. As an example, consider the equilateral triangle construction (Figure 9.3) which is used to demonstrate snap-dragging [BS86]. In this example, alignment circles of $3/4$ inch are used to create an equilateral triangle. The user has specified a triangle with all sides equal to $3/4$ inch, not a triangle whose equal sides can be scaled as well as rotated and translated. The program only knows what the user has specified and, therefore, cannot guess another option.

Briar does not attempt to guess the user's intent. Instead, it tries to keep the automatic generation process predictable, to use feedback to remind the user exactly what the system has been told, and to provide tools which convey the desired relations directly. Since our goal is to extract constraints without extra work, it is wrong to require users to expend extra effort to use constructions which create the correct constraints.

Therefore, we must make it as easy as possible for the user to convey what is really intended.

The scalable equilateral triangle problem is solved using a mechanism from snap-dragging. Rather than specifying that the alignment circles, and therefore the resulting triangle, have size $3/4$ inch, a length measurement tool is employed. The first line segment is drawn and then measured. Circles are then created with radius equal to the length of the line segment. In Briar, this construction requires only one more mouse click than to construct the fixed sized triangle. Angle and slope measurement tools could also be provided to express other relationships.

As we find relationships that are needed in drawings but are difficult to express directly with current tools, we can develop new tools to expand the set of relationships that are easily specified. Expanding the vocabulary by adding a wider assortment of tools only makes modeling easier to a point, after which the larger number of tools becomes unmanageable. User experience with Gargoyle [Bie89] shows that the standard set of snap-dragging tools are sufficient to create a wide array of drawings.

Another complication in augmenting snap-dragging is that if the constraints already imply that two objects are related, there is no reason to snap them together. Suppose the user is dragging a point. Gravity snaps the cursor to nearby objects. If another object is connected to the point, it will always be nearby and it might be snapped to. Since it is already connected, this would create a redundant constraint, or require to the user to cycle in order to snap to another object. Filtering the set of objects that can be snapped to eliminates this nuisance.

However, recognizing that a relationship is implied by a set of geometric constraints can require difficult geometric proofs as some complicated mechanism might imply additional relationships. From our experience, it appears that handling the simple cases—not snapping to the object being dragged, checking for existing connection constraints, applying basic geometric identities, etc.—filters out the vast majority of redundant snaps. Avoiding redundant snaps is an optimization; it is not critical to the correct functioning of snapping. However, not filtering all redundant snaps means that the solving techniques must be robust in handling redundant constraints.

9.1.2 Constrained Direct Manipulation

Briar uses augmented snap-dragging to initially establish constraints and then uses the methods of the differential approach to maintain these constraints during subsequent editing.

With constrained direct manipulation, objects are dragged as with standard direct manipulation, except that relationships are maintained among them. In Briar, direct manipulation does not break persistent constraints, that is, dragging is subject to the constraints. This allows the drawing process to be incremental: each new relationship added to a drawing does not disturb previously established ones. The existence of a

direct manipulation facility means that all parts of the model do not need to be specified by constraints. If it is difficult to devise a way to describe an aspect of a drawing with constraints, direct manipulation can be used instead.

Dragging parts of drawings allows the user to experiment with the constrained model. This interactive animation is a useful tool for understanding and debugging constraints. It also opens up the possibility of *dynamic drawings:* models which are created to be dragged and played with. Such drawings contain unconstrained degrees of freedom, so they exhibit motion when their parts are dragged. The use of constraint-based approaches to animate drawings dates back to Sketchpad [Sut63] and special purpose techniques for interactively simulating mechanisms have been developed [Kra90, RK77, End90]. Differential constraint systems are also well suited for such mechanical simulation and animation tasks as well as interaction. Section 9.2 describes a special purpose application for mechanism sketching.

The facility with which we handle constraints opens the possibility of using constraints to aid in manipulation. Most constraints are used to represent structure in the drawing. However, temporary constraints that are easy to create and destroy are useful for making drawings easier to control. We call such constraints, which are meant to be short-lived, "lightweight" constraints.

Dragging is one example of a lightweight constraint. It is achieved by temporarily constraining the point being dragged to follow the mouse. Another useful lightweight constraint is the *tack.* A tack hold a particular point in place. It acts as an extra hand, making it easy to stretch or rotate an object. Nailing a point at a particular point in space is a common facility in constraint-based systems. Making it easy to place and remove the nails easily enables new uses for them. For example, tacks can perform the tasks that anchors do in traditional snap-dragging [Bie89], specifying the center of rotation and scaling.

9.1.3 Displaying And Editing Constraints

A constrained drawing has more state that must be displayed to the user than a non-constrained one does. A system must convey to the user not only the geometry of the model, but also its constraints. The user must be able to edit this structural information as well as the geometry. Previous constraint-based systems have used three types of techniques for displaying constraints to users: textual languages, diagrammatic representations, and graphical cues drawn directly on the model.

Textual languages for describing constraints, such as that employed in Juno [Nel85], have the advantage that they are editable. Unfortunately, they are distinct from the drawing and can be difficult to connect. Schematic representations of constraints, such as that presented by Borning [Bor86], are similar in that the constraint display is separate from the drawing.

Visual representations, such as in Converge [Sis91], CoDraw [Gro89], and Viking

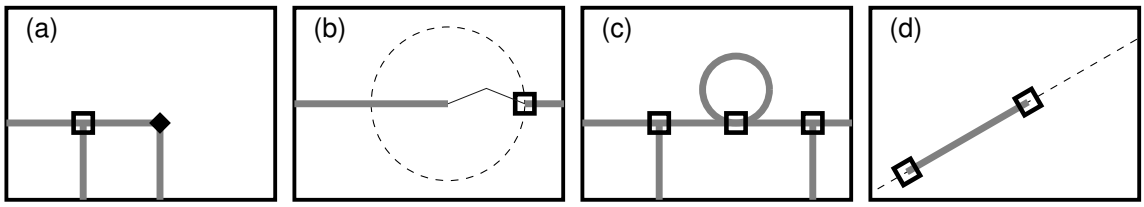

Figure 9.4: Snap-Dragging provides a basis for visual representation of constraints. The two basic constraints, point-on-object and point-to-point, are drawn as an empty square and a filled diamond (a) respectively. Other constraints are represented using the basic ones and alignment objects, for example, distance-between points (b), points-aligned (c), and orientation (d). Thin wedges as seen in (b), emphasize distance constraints. Color is used for constraint display when available.

[Pug92], superimpose constraints directly on the drawing. The connections between the relationships that will be maintained and the objects they affect are shown to the user. This is particularly dramatic when the model is moving subject to the constraints, such as when it is being dragged.

Constraint representations that are superimposed on drawings do have drawbacks. Each type of constraint must be displayable in a manner that makes clear both what the constraint is and what it effects. This can be particularly difficult in systems with a large palette of constraints. Visual representations can also be difficult to edit. The tendency of constraints to cluster is one source of this difficulty.

A Visual Representation of Constraints

Augmented snap-dragging provides a graphical method for specifying constraints. One of its features is that it provides a uniform method for describing a wide variety of relationships. It can also be used as a representation for displaying constraints, providing an equally uniform visual language for displaying constraints that is the same as that used to specify them.

Augmented snap-dragging specifies all relationships by one of two basic types of snapping operations. The graphical depictions of these two types of snaps becomes a way to depict the constraints the snaps create. Alignment objects are also part of relationships and therefore they must also be made persistent, unlike traditional snap-dragging. The snapping symbols and alignment objects provide a graphical representation for a wide variety of constraints. Some examples are shown in Figure 9.4.

One factor which complicates visual display and editing of constraints is the tendency of constraints to cluster, often being in exactly the same place. The semantics of constraints can be tuned lessen this problem. For example, rather than using binary relations to connect points to each other, linked points are placed into an equivalence class. If a set of points has been made equivalent, just the equivalence class needs to

be shown, not the potentially large number of equality relationships all piled on top of each other.

Editing Constraints

Ease of editing constraints is important; relations should be as easy to break as to make. Users may change their minds as to what relationships should be in the model, or may simply have made a mistake in specifying the constraints. Using a visual representation alleviates only part of the difficulty in deleting constraints: before being able to point at a constraint, the user must know which constraint or constraints to delete. To address this, we allow users to remove constraints by referring to the desired effects, not to the constraints themselves. In fact, Briar provides no mechanisms for users to point directly to constraints.

A direct method of removing constraints by referring to the objects they influence is to "rip" them. When an object is grabbed for dragging, holding down a modifier key causes the grabbing operation to "grab hard," removing any constraints on the point grabbed. However, this method is often undesirable as it may remove too many constraints.

Another technique for deleting constraints without pointing at them is to allow the user to temporarily disable constraint maintenance. In this mode, the user can manipulate the objects as if they did not have any constraints on them. When constraint enforcement is re-enabled, constraints which the user has broken are discarded. Feedback as to which constraints are broken and the ability to use snapping operations to reassemble things help make this a useful technique for editing constraints. A similar mechanism for destroying unwanted constraints is provided by Chimera [KF93], where the user can create a new "snapshot" of the drawing which shows the constraint broken.

9.1.4 Lessons from Briar

Briar was constructed as a testbed for ideas and techniques for constraint-based drawing. It was developed in the autumn and winter of 1990, before most of the work in this thesis. Briar was built with a predecessor to Snap-Together Mathematics. It employed a conjugate-gradient solver directly to handle under- and over-constrained cases, as described in Section 3.6, and a "Briar-style" two pass solver for soft controls, as described in Section 3.5.1.

Briar is an important demonstration of the differential approach because:

- it provides an example that shows the approach is viable;
- it shows that many of the difficult issues in using the approach can be addressed;
- the techniques that it introduced, such as augmented snapping, can be applied to other applications;

- it provided me with experience of what services constraint-based graphical editors required, which influenced the design of subsequent toolkits;
- it showed how Snap-Together Mathematics could be incorporated into an application, which led to alterations in the design of Snap-Together Mathematics;
- it inspired the Bramble toolkit because building a graphical editor without significant support to leverage off of was too much work;
- it emphasized many of the features required for the mathematical techniques, such as continuous motion and redundancy handling;
- it provided an interface that permitted creating larger, constrained models that taxed the performance of implementations, providing inspiration and test cases for exploring performance enhancements.

9.2 A Planar Mechanisms Sketcher

Creating drawings of mechanisms is a common use of a constraint-based drawing program. Not only do the constraints help accurately place objects to create the mechanism, but they also keep the mechanism together as it is manipulated. This permits the user to play with the mechanism to experiment with its behavior. Because mechanism creation was such a popular use of Briar, the first Bramble application was a special purpose drawing program designed especially for drawing and experimenting with planar linkage mechanisms. The *MechToy* application is illustrated in Figure 9.5.

Mechtoy defines a number of special purpose graphical object types. Linkage rods are line segments drawn to appear more like what is commonly used to draw mechanisms. The length of the rods are constrained like the rigid elements that they model. The point equality constraints used to connect pieces are drawn as hexagonal bolts to look more like a mechanical object. The ground points that provide fixed locations for pivot points are drawn using the standard symbol from mechanics texts.

Special adjuster objects permit altering the geometry of mechanisms. When an adjuster is dropped onto a linkage rod, the length of the rod is unfrozen. Deleting the adjuster re-freezes the rod's length.

Several MechToy object types have points that are nailed in place except when they are grabbed. For example, ground points are normally locked in place, permitting attached objects to pivot around them. However, if the user grabs the ground point, it is temporarily freed so it can be moved during dragging.

Special variants of objects permit creating a variety of mechanical contraptions. Sliders are line segments that have their endpoints nailed in place, and a floating bead that is limited to sliding along the line's length. Motors are special variants of line

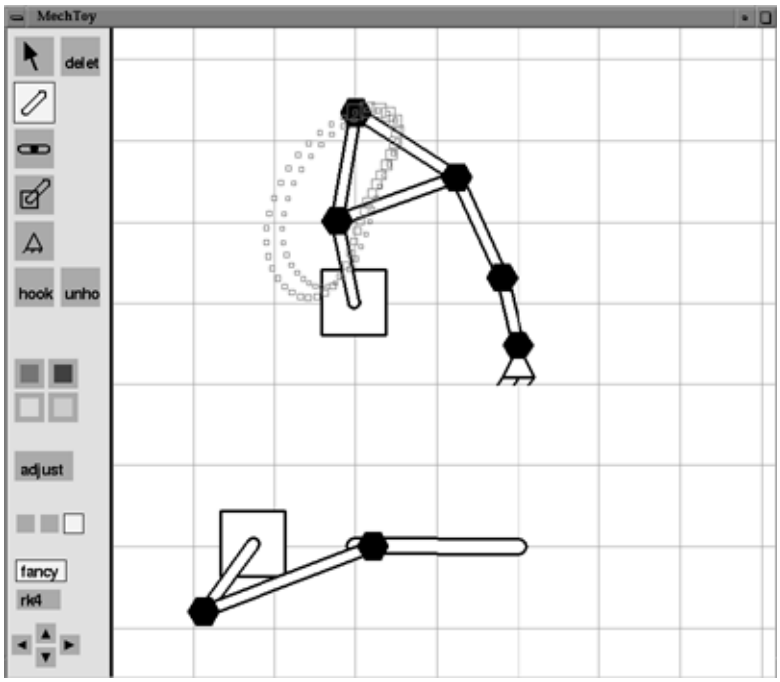

Figure 9.5: The *Mechtoy* application is specifically designed for drawing and animating planar linkage mechanisms. This picture shows a mechanism being animated, with smoke trails tracing the motion.

segments that have one endpoint nailed in place. When a button is pressed, a soft control is placed on the motor's orientation that causes it to rotate. Motors permit mechanisms to be animated.

Smoke trails are objects that can be drag-and-dropped on points in the mechanism. They remember a history of the point's position, and display it in a non-obtrusive fashion as a decaying trail of dots, as seen in Figure 9.5.

Mechtoy uses augmented snapping to infer constraints on objects as they are created. If an object is created in a position snapped to another, the two are bolted together. This permits mechanisms to be sketched very rapidly. Mechanism can easily be disassembled by deleting parts, such as the bolts between parts.

9.3 A Box-and-Arrow Diagram Editor

Another major use of a constraint-based drawing program is to create diagrams with arrows connecting the pieces. Constraints are particularly useful in such an application because they permit the structure of the drawing to be retained as the pieces are moved. This can save a considerable amount of work of reestablishing connections after each edit.

Diagrams are an important enough class of drawings that a market for specific tools for creating diagrams exists. Such tools can maintain connections without a general purpose solving mechanism because of the simplicity of the specific constraint problem. Simple dependencies suffice for keeping boxes and arrows attached. One of the many successful commercial examples is Visio[Sha93]. Visio is a particularly interesting example because it permits users to define new types of objects and connections using a spreadsheet interface.

A simple diagram editor called *Boxer* has been built with Bramble and is shown in Figure 9.6. Boxer provides a few basic object types, each providing connectors at points around their periphery. Boxer objects automatically display text inside themselves, although text-editing is not supported[4]. The objects are constrained to be axis aligned. They can be created and placed using a drag-and-drop interface from the palette provided at the top of the window. The drag-and-drop interface permits users to select icons from the palette and drag them onto the drawing area. The technique, which is also used in Visio, avoids separate modes for object creation.

Connecting lines are specified by providing two points on boxer objects. Arrows can be placed at either end. The connecting lines and arrows look at the normals of the points they are attached to in order to attach at a proper angle. The connecting arrows are drawn as Bezier segments to have a smooth appearance yet approach their destinations with the correct orientations.

[4]Text handling facilities are conspicuously absent from Bramble.

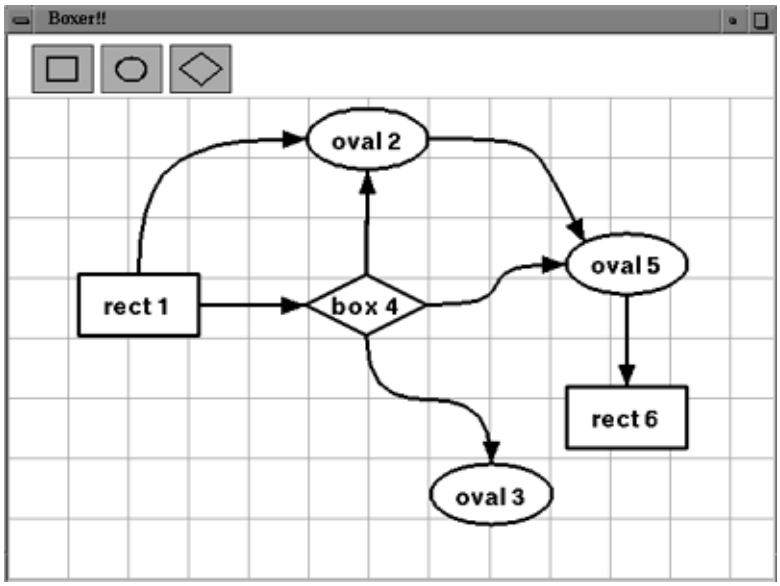

Figure 9.6: The *Boxer* diagram editor is designed for creating box and arrow diagrams.

Boxer was the first Bramble application to employ non-interpenetration constraints, as described in Section 8.4.2. Boxer objects are constrained not to overlap. As an object is dragged, it pushes other objects out of the way, just as physical objects would. The collision mechanism treats all boxer objects as rectangles. The simple collisions have been useful for a variety of purposes, including moving stacks of objects and clearing space around an object by pushing neighboring objects away.

Boxer is designed to be extended easily. It defines a Whisper function called `install-shape` that installs a new shape in the interface, adding a drag-and-drop icon to the top area of the screen. The `install-shape` function takes the output provided by `define-shape` (Section A.2.1), or the functions to create instances and draw icons can be created by hand.

9.4 A Curve Modeller

The *NewFF* program shown in Figure 9.7 is an application designed to permit experimenting with 2D parametric curves and constraints. It is named after Andy Witkin's original parametric curve manipulation program [Wit89b]. Like Witkin's original FF system, and several generations of successors described in [WGW90], [GW91b], and [GW91a], NewFF provides a variety of curve and constraint types, and offers easy addition of new types.

As discussed in Section 8.1.1, all that is required to add a new type of parametric curve is the parametric function used to draw the curve. NewFF is distinguished from

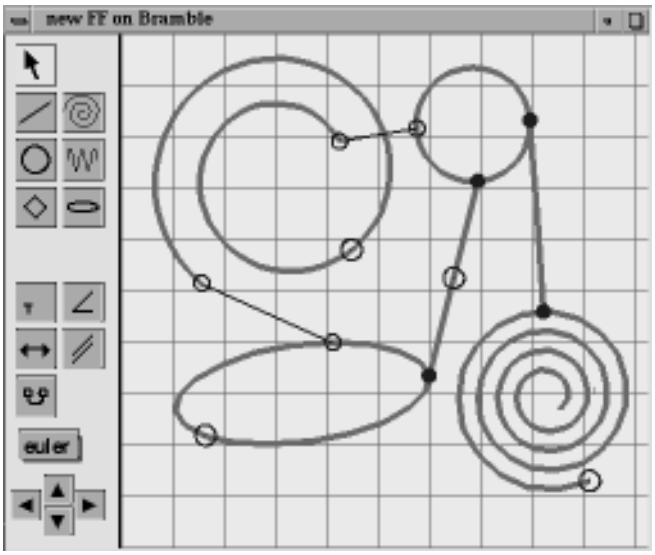

Figure 9.7: The *newFF* curve modeller allows experimenting with a variety of types of parametric curves and constraints.

its ancestors by permitting new types to be defined dynamically while the application is running. The parametric function is given as a Whisper expression, and a new curve type is defined, including an icon for creating instances. A small amount of auxiliary information is also required to specify the sampling of the curve, and initial values for the parameters. Because the dynamic addition of curve types interprets the function using Whisper, performance (especially for redraw) is poor. Therefore, new curve types are best added at compile time. A Mathematica program automatically generates C++ code for new Bramble object types from parametric functions.

The standard creation mechanism in the curve modeller simply creates an instance of the object with its default initial values for the parameters. The initial values can be defined in terms of an initial point value. When NewFF creates an object instance, it sets these parameters equal to the position of the mouse. Certain objects in NewFF have more sophisticated creation mechanisms. For example, line segments and circles are created with rubber banding and augmented snapping to infer connection constraints on the object.

The curve modeler allows a variety of constraints to be placed on objects. All constraint types attach to points, and therefore can be connected to any object. Available constraints include connection, distance, collinearity, and parallel-ness of four points.

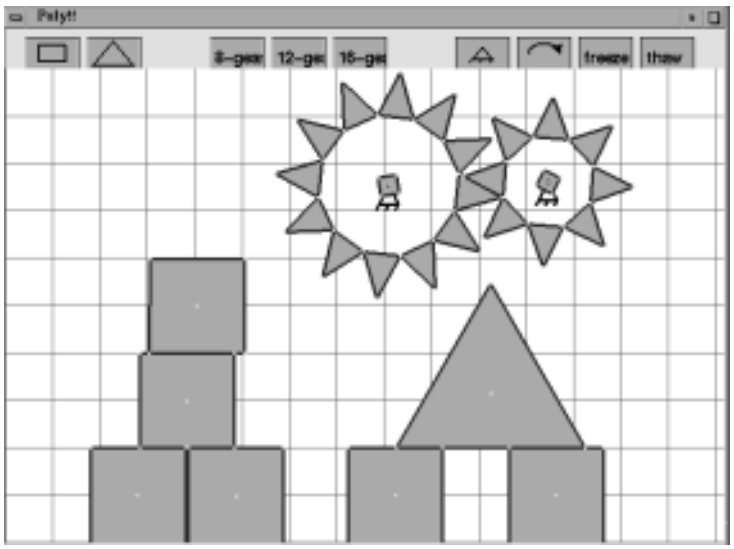

Figure 9.8: The *Poly* application for experimenting with planar collision simulation.

9.5 A Collision Simulator

A simple application to experiment with the planar convex polygon collision constraints of Section 8.4.2 has been built with Bramble and is shown in Figure 9.8. The *Poly* application permits several types of polygonal objects, such as blocks and sawtooth gears, to be created with a drag and drop interface. All objects automatically are constrained not to overlap other objects and to remain above the floor. Gravity, implemented by a soft controlling pulling downwards, can optionally be placed on objects.

A major use of Poly is to create mechanisms, so some features to facilitate this are provided. Using the drag and drop interface, the user can nail points in place, freeze objects, or place a torque source at the center of the object (e.g. turn it into a motor). All of the Bramble functionality, including the objects and constraints from MechToy, can be used in creating Poly models.

9.6 3D Construction Toys

Simulating Tinkertoys[5] was one of the original motivating applications for the creation of the constraint technology that the differential approach is built on. The idea is to create a graphical modeling environment where pieces can be connected together to build more complicated objects that have interesting behaviors. Tinkertoys, with pieces that can be plugged into one another to create mechanical connection, was an direct

[5]Tinkertoy is a registered trademark of PlaySchool, Inc.

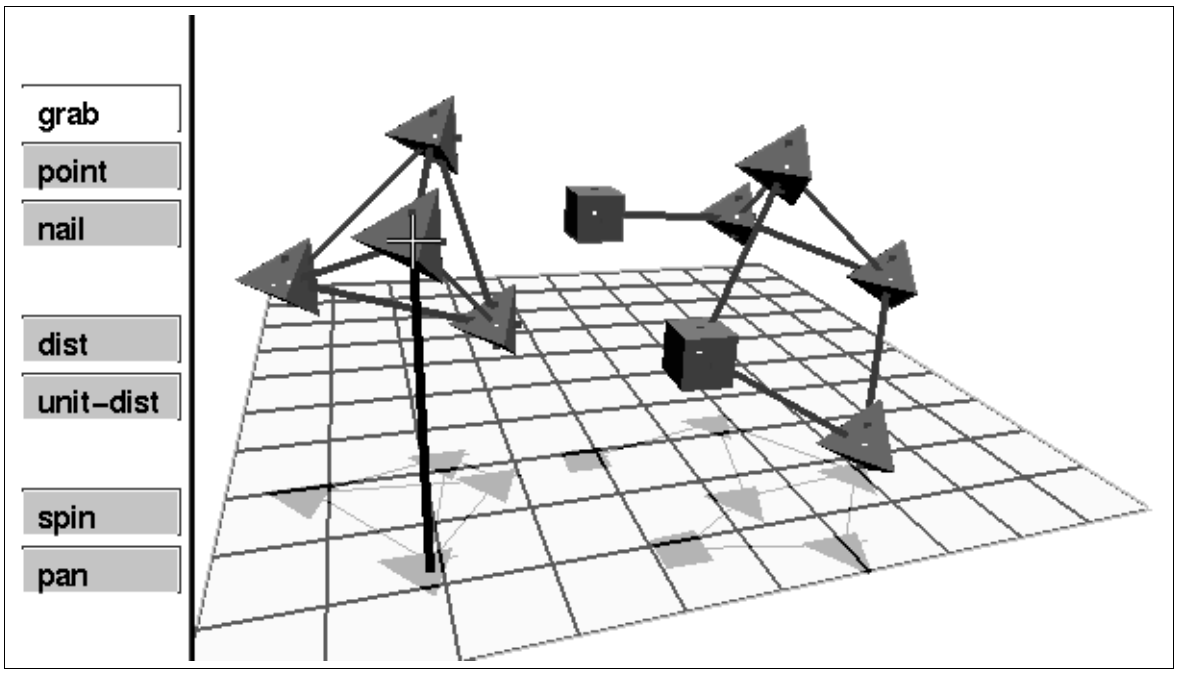

Figure 9.9: The *Ptinker* application permits the user to build simple models from point objects and constraints.

metaphor for constraint-based modeling. The freedom and flexibility needed to play with Tinkertoy models had served as a touchstone for evaluating 3D interfaces.

One of the earliest applications built with the original snap-together math implementation was a simple Tinkertoys like system which provided a simpler set of objects that could be attached with point-to-point or distance constraints. Although this application was called Tinkertoys at the time[WGW90], it models a simpler set of constructions. A version constructed with Bramble is shown in Figure 9.9. *Tinker* or Point Tinkertoys recreates the functionality of the original, allowing a user to create simple objects, connect them with distance constraints, and manipulate them with the mousepole.

Point objects in PTinker are denoted by small tetrahedra. Points that have fixed positions in space, called "Ethernails,"are denoted by cubes. Distance constraints are shown as lines between point objects. The interface permits specifying distance constraints by selecting two points to connect. When connected, the distance can be constrained either to be a unit distance or to simply maintain the distance between the points.

The *Toys* application, depicted in Figure 9.10, is a more ambitious simulation of tinkertoys. Toys includes pieces designed to mimic their counterparts in a real Tinkertoys set. The sizes and colors of the objects are reproduced in the simulation. Connecting pieces by plugging rods into holes on the yellow connector objects is modelled using

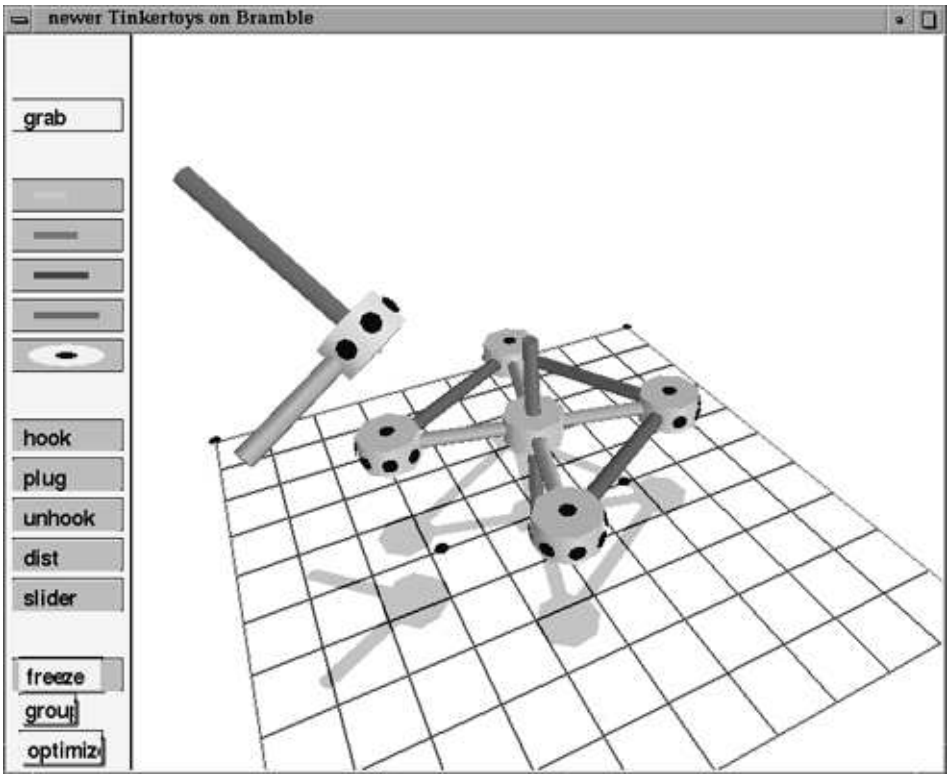

Figure 9.10: The Tinkertoys simulation permits users to create objects out of pieces modelled after the real tinkertoys.

constraints.

In order to test if the abstractions of Bramble were sufficient to create the range of behavior desired in a complicated application, no new functionality in the toolkit was added. Toys is written entirely in Whisper. The application provides a complete 3D system, with a tuned interface. This was important to show that such an application could be constructed with the tools, but also to provide a realistic testbed for interface ideas. The interface for Tinkertoy simulation provides a variety of interface challenges. The application must provide for the fluent manipulation of 3D objects, easy specification of constraints, and quick control of the view. Because the application is designed to be fun, it would be unacceptable if the interface was not fluent and expressive. Personally, I think the Toys application is a success — *I* enjoy playing with it.

9.6.1 Tinkertoy Pieces

Toys models most of the objects in a Tinkertoy set. The dimensions and colors of the real parts are reproduced. The rods and connector spools are made from standard cylinder primitives. The dots for the holes on connector spools are created by using

a Bramble function that marks a `DistinguishedPoint` with a circle in its tangent plane. The connector spools use a `DistinguishedPoint` on their surface for drawing, but each hole also has a corresponding point inside the cylinder that is used for connection. This is required to model the fact that the rod pushes into the hole.

Connections between pieces are specified by picking an end of a rod and a hole. The pieces then "self-assemble" to establish the connection. The self-assembly is phased into several stages. First, the pieces orient themselves, then they fly together, and finally, the piece pushes into the hole (rods actually do go inside the holes). The phasing was created for purely aesthetic reasons, and is implemented by demons that wait until one phase is complete before beginning the next. To make the connection operation seem more natural, Toys attempts to move only one object in order to establish the constraints. Heuristics are used to decide which object to move. The system prefers to move pieces that have not been connected yet, and prefers to freeze connector spools instead of rods. If both the rod and its target have other connections, neither is frozen.

The Tinkertoy interface permits objects to be grabbed and dragged using the mousepole, and the view to be controlled using a virtual trackball by grabbing the groundplane. Rods are plugged into the holes on the connector pieces. Once connected, the rod can rotate is the hole.

To aid in picking, Tinkertoys provides semantic snapping so that only valid objects can be picked when various operations are to be performed. For example, when plugging a rod into a hole, only unattached rod ends and empty holes are snapped to. For unplugging, only rod ends that are plugged into holes are seen by the snap server.

9.6.2 Performance of Tinkertoys

Simulating Tinkertoys provides difficult performance goals. Because of the complex behavior of the 3D models, rapid refresh rates are important. However, tinkertoy models have large numbers of constraints. Each rod into hole connection requires 5 constraints. The total number of constraints to simulate can grow large very quickly. For the Toys application I set a specific performance goal: on the machine in our lab (an SGI Indigo 2 Extreme), the program should achieve acceptable performance on objects that were as complicated as those that could be built with the small set of real Tinkertoys. The small jar of Tinkertoys contains roughly a dozen rods and a half-dozen connectors.

Meeting the performance goals for Toys pushed the limits of Bramble. The objects in Tinkertoys were all standard cylinders, quaternion transformations, and frame-alignment constraints. Because these are often used objects, they were carefully optimized. However, users of Toys quickly assembled models that were large enough that that $O(n^2)$ linear system solving dominated the performance. Therefore, a number of implicit constraint methods were created to enhance the performance of Toys. All of the methods use only standard Bramble features.

The implicit constraint methods in Toys exploit the fact that connected pieces form rigid bodies. Technically, a rod that plugged into a connector's hole has a degree of freedom to rotate around its axis. However, since the rod is symmetric, this degree of freedom is invisible to the user. Toys can, therefore, treat the two pieces as a single rigid body. Rather than having two objects and 5 constraints, the system need only simulate a single rigid object. If the other end of the rod is plugged into another connector, the degree of freedom can be manipulated by the user. Therefore, Toys can use implicit constraint techniques for at most one connection per rod. Another case where the symmetry of a rod permits implicit constraints is when a rod is connected to the ground. In such a case, the rod can be frozen.

When a connection between two parts is accomplished by an implicit constraint, the connection must work the same way that it would if normal techniques were used. All connections must appear the same to the user; they are specified and deleted in the same ways. This creates several complications. For example, when a connection is deleted, there are no constraints to delete if the constraints are implicit. Instead, disconnection must ungroup the two objects. This is implemented by having phantom connection objects that represent implicit constraints. The phantom objects specify a deleter hook that performs the ungrouping. The phantom objects are also convenient for implementing save and load.

Creating a connection with an implicit constraint also poses a challenge as the constraint must self-assemble first. This is accomplished by initially creating the connection with regular constraints. A demon waits for the connection to be made. When the two pieces are connected, the demon converts the connection to use implicit constraints. If the demon fails by timing out, it does not convert the connection. In fact, it takes a precaution against the constraint never being satisfied by adding additional damping.

To demonstrate the benefits of the implicit constraint methods in Toys, we consider building an example object. The merry-go-round object, shown in Figure 9.11, is a model that is shown in the instructions to the real Tinkertoy set. It is made of four small rods, four medium rods, a long rod, four connector spools, and a slider connector. These pieces have a total of 93 variables. Toys running on an SGI Indigo 2 Extreme takes approximately 50 milliseconds to redraw the display with these pieces, whether they are attached or not.

Without the automatic implicit constraint techniques, the merry-go-round requires 85 constraints[6]. Dragging this merry-go-round with the mousepole takes approximately 320 milliseconds per 4th order Runge-Kutta step. This provides a frame rate of little more than two frames per second. This performance is unacceptable. However, using the implicit constraint methods, the merry-go-round only requires 40 constraints, and has a much smaller number of variables. The frame rate for this model is over 6 frames per second, as each Runge-Kutta step requires only approximately 100 millisec-

[6]The slider is implicitly connected to the vertical pole.

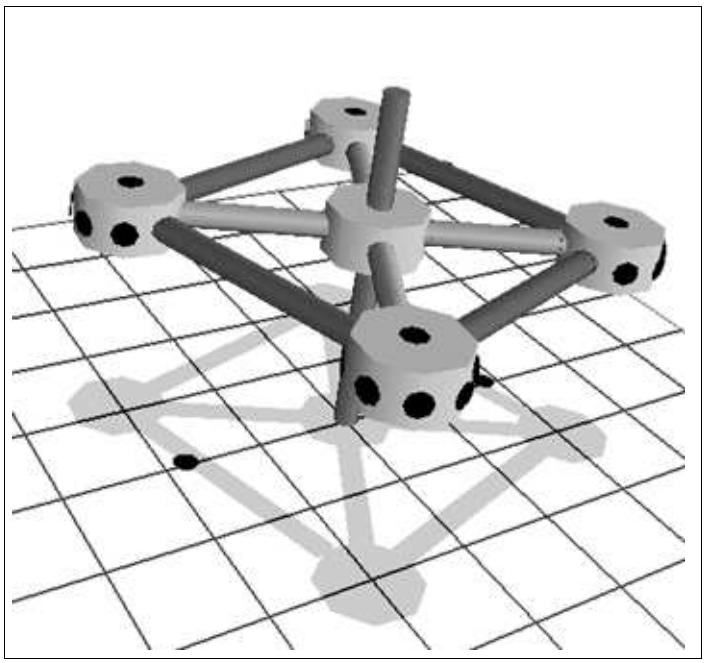

Figure 9.11: A Merry-Go-Round constructed in the toys application from 9 rods, 4 connector spools, and a slider connector.

onds. While interaction is a little sluggish, it seems acceptable.

The merry-go-round illustrates the limits of implicit constraint techniques as well. The merry-go-round actually only has two degrees of freedom: it can slide up and down the pole, or spin around the pole. All of the parts except for the pole form a single rigid body. The merry-go-round could be simulated by grouping these pieces as one unit, rather than using constraints. Identifying that more complex objects are rigid is difficult, and is left for future work.

9.7 Scene Composition

A scene composition system permits a user to position objects in a 3D space, assign properties to them, light them, position the camera, and send the model off to a renderer to have an image created. Scene composition is a good testbed for 3D methods because it requires controlling objects, lights, and cameras, and it avoids many of the issues of how models are to be edited and represented.

The Bramble scene composition application is called *Showoff*. The name comes from the fact that it was originally intended as a viewer for objects in a standard[7] file

[7] I don't know how much of a standard it is, but a large number of sample models in this format can be found in public archives.

format called OFF. Showoff is not an application in the traditional sense, but rather, it is a framework for building demonstrations of interaction techniques for scene composition problems. Unlike other applications, Showoff does not create windows or fill the scene with graphical objects. Instead, it defines Whisper functions that make it easy for scripts to do these tasks. For instance, it defines a function that automatically creates a framed view, complete with a standard camera and buttons along the side to set various viewing modes. Showoff creates commands for finding and manipulating object aspects.

Showoff's main role is for testing and demonstrating attributes and interaction techniques for 3D applications. The typical showoff script creates some objects to provide a sample scene, sets up some initial controls and constraints, and begins the interactive loop. Showoff binds keys and menus to commands for many standard interaction techniques. For example, mouse buttons can be used to grab a point with the mousepole or through-the-lens with either soft or hard constraints.

Showoff has been the primary vehicle for experimentation with controls for positioning objects, lights and cameras. A wide variety of controls are provided for the user, or the script, to mix and match. Figures 8.3, 8.5, 8.10, and 8.8 are all scenes from Showoff demonstrations.

Showoff provides few features aside from those used to manipulate scene objects. For example, the interface does not help the user create objects, they must be created by writing Whisper code. However, Showoff does contain an extensive set of commands that allow the differential approach to be applied to the problem of scene composition directly. Showoff presents the user with a large range of controls, including most of the attributes of Section 8.1, that can be applied as needed. Controls are combined by locking: any control can have its value nailed in place. The idea is that the user will employ a set of constraints and controls that conveniently described the image to be created.

Showoff does not address the difficult problem of determining how to present a wide range of controls to a user. A sufficient number of controls are available in Showoff that accessing them is a problem. Showoff uses a haphazard combination of menus, keys, and buttons to specify operations. The interface is "designed" to maximize the number of controls made available to the user, even at the expense of usability.

There are two times in his life when a man should not speculate: when he can afford to, and when he cannot.
— Mark Twain

Chapter 10

Evaluation and Future Work

In this thesis, I have presented a differential approach to graphical interaction: a systematic approach to realizing graphical manipulation based on numerical constraint techniques. In this concluding chapter, I will review the contributions of the thesis, evaluate the work in light of the goals, and suggest some directions for future work.

10.1 Contributions

The central contribution of this thesis is to develop and demonstrate an approach for realizing graphical manipulation based on numerical constraint techniques. The thesis extends previous work in interactive physical simulation into a general approach for graphical manipulation with contributions in the following general areas:

- abstractions for graphical interaction;
- numerical constraint techniques for graphical applications;
- implementation techniques for numerical computations;
- software architecture for graphical applications;
- interaction techniques;
- graphical applications.

This section reviews the specific contributions of the thesis.

I feel that the biggest contribution of the thesis is not any of the individual pieces, but rather, how all the pieces can be fit together. To create the differential approach requires a new set of abstractions that define graphical manipulation as constrained optimization, mathematical methods to solve the constraint problems, implementation methods to address the practical issues in employing mathematics in interactive systems, and a software architecture to enable applying the approach and building applications. Only with all of these pieces can novel interaction techniques and applications be created. Here we discuss the specific contributions in more detail.

10.1.1 New Abstractions for Graphical Manipulation

At the core of the differential approach is a view of graphical manipulation as a constraint solving problem. Interaction techniques are created from connectors, controllers, and the passage of time.

- **Equation solving as a general abstraction of graphical manipulation:** the differential approach treats graphical manipulation as an equation solving problem. Previous work has used numerical equation solving and optimization for interactive control in specific cases and non-interactive control of geometric objects with general constraints. However, no one has previously explored numerical optimization methods as a general mechanism for implementing graphical manipulation.

- **Connectors and controllers as basic abstractions for building graphical manipulation techniques:** the concise set of abstractions provided by the differential approach are quite different than any previous ones. They can be combined and composed. One key feature of the interactive controls in the differential approach is that they are combinable: if we can control A, and we can control B, then we can control A and B simultaneously.

- **Application of numerical constraint solving to interaction technique design:** interaction techniques are described by sets of controls and constraints that are enabled and disabled at appropriate times. This leads to concise, directly executable descriptions of classes of interaction techniques that have been difficult to specify, such as 3D widgets. Freeman-Benson's Kaleidoscope '91 system [FBB92] defined interaction techniques and widgets by dynamically switching sets of constraints and previous work such as Olsen and Allan [OA90] and the Garnet system [MGD+90] have used constraints to describe the behavior of 2D widgets. However, by employing numerical techniques, the differential approach permits a richer set of constraints including inequalities, to be used to specify widgets in a way that automatically defines its inverse mapping from input to variable.

- **A method for describing interactions independent of the underlying representations of the graphical objects:** interaction techniques are defined in terms of the objects' attributes. The equation solving mechanisms map changes to the underlying parameters.

10.1.2 Numerical constraint methods for graphical applications

The use of numerical constraint techniques for graphical applications began with the relaxation techniques used in Sketchpad[Sut63]. More recently, there has been interest in the related methods of non-linear constrained optimization, non-linear equation

solving, and physical simulation. The problem solved by the differential optimization is very similar to the problems solved at the core of these methods. The solution techniques discussed in this thesis are variants of standard mathematical techniques for solving relatively standard problems.

The methods used in the differential approach are the methods used for interactive simulation. The techniques described in [WGW90] have been extended by applying some standard mathematical techniques, such as damping (Section 3.2.1). This thesis brings these methods to the domain of user interface design and construction. Previous work attempting to apply numerical controls to the design of interactive systems has been plagued by starting from inferior mathematical foundations.

- **Demonstrations of the practicality and utility of the differential approach:** the running systems show that general controls for graphical manipulation can be implemented, and perform acceptably on existing computers.

- **Application of damping to constraints in interactive graphics:** the differential approach uses damping methods (Section 3.2.1) to provide stability in ill-conditioned cases, and to handle over-determined cases. These singular cases have plagued many previous systems. Damping methods are well known in the optimization community, and have been applied to similar domains, such as animation and robotics. This thesis introduces the use of selective damping to achieve constraint-hierarchies.

10.1.3 Implementation Techniques for Numerical Computations in Interactive Systems

Another contribution of this thesis is the methodology for solving the numerical computations required for constraint approaches in a manner that addresses the issues of interactive systems. The work of this thesis refines the earlier work on interactive physical simulation.

- **Snap-Together Mathematics:** the work described in Chapter 5 is a descendant of earlier tools for dynamic function composition and derivative evaluation. The basic ideas behind Snap-Together Mathematics, such as explicit representation of expression graphs, compile time definition of new function blocks using a symbolic math system, and sparse-vector-passing, forward mode, automatic differentiation remain from its earliest ancestor described in [WK88]. Several details, such as how variables are handled, have been refined with evolution. The current implementation is the first to be incorporated into a higher level toolkit, and the first to exploit the generality in a range of applications. Snap-Together Mathematics and its ancestors are very different from most other automatic differentiation tools which focus on large, dense problems statically defined at compile time[BGK93].

- **An encapsulation of numerical constraints:** Snap-Together Math's simple protocol for connector outputs, combined with its objects for constraints and constraint engines, provide facilities for defining (and solving) constraints on equations without seeing the underlying mathematics. The `ConstEngine` class has been used with other solvers, such as Newton-Raphson.

- **Generalization of performance methods for numerical constraints in interactive systems:** the performance methods of Chapter 4 have they been used in previous systems. However, the work of this thesis shows how all can be integrated in a general fashion. In particular, the scatter–gather mechanism of Snap-Together Mathematics, coupled with variable selection mechanisms and variable merging mechanisms provides a generalized substrate for speeding numerical computations through switching between explicit and implicit constraints and partitioning.

10.1.4 Software Architecture for Graphical Applications

This thesis described Bramble, an object-oriented toolkit based on the differential approach. The main contributions of Bramble are:

- **A graphics toolkit encapsulating numerical constraint methods:** Bramble encapsulates the differential approach in a manner that provides services inside the abstractions of an object-oriented graphics toolkit. In fact, Bramble permits building graphical applications with constraints without referring to anything other than the typical abstractions of an object-oriented graphics toolkit. This is important because designers of interactive systems seem to be resistant to learning about mathematical techniques.

- **Demonstrations of the impact of the differential approach on application architecture:** by providing generalized methods for mapping between controls and parameters and by permitting connectors to be attached without knowing what is behind the connector, the differential approach fosters modularity, parameter independence and common code. This is seen in Bramble since objects, constraints, and interaction techniques can all be created independently and hooked together as needed.

- **A graphics toolkit that facilitates providing general constraints and controls to users:** Bramble aids applications in providing constraints and multiple controls to the user. Previous toolkits have used constraints as an internal abstraction to aid the programmer, but Bramble's focus is providing them as a user level service.

- **Use of a time continuous input model in a graphics toolkit:** the differential approach provides a continuous model of time, with events integrated as instantaneous impulses. For a graphics toolkit, the model allows simultaneous asynchronous actions to be handled without explicitly programming them into each event handler or providing for multi-threaded flow of control. A fundamental difference between this strategy and standard approaches is that handlers for continuous actions such as dragging are instantaneous event handlers, they do not remain active over the duration of the operation, for example from mouse down to mouse up.

- **An object oriented graphics toolkit specifically designed for prototyping of 2D and 3D applications and techniques.** Much of the rationale for Bramble is based on the intention of its use for prototyping a variety of interactive graphical applications. This led to exploring a different point in the design space of interface toolkits. Because the art of toolkit design, especially for 3D, is still evolving, explorations of new points in the design space are still useful. Interestingly, other 3D-only graphics toolkits with emphasis on rapid prototyping, including UGA [CSH+92], MR [SLGS92], Alice [PT94], and VB2 [GBT93], have independently made some similar choices.

10.1.5 New Interaction Techniques

In order to demonstrate the differential approach, this thesis presents a variety of interaction techniques, many of which are novel enough to be considered contributions. Without user testing, it is in many cases difficult to claim that these are clear improvements over previous methods. In fact, some of the new techniques may be unusable in practice. However, I think that a few are interesting. Also, I think the differential approach will lead to many new good interaction techniques because it permits providing the advantages of direct manipulation, such as continuous feedback and kinesthetic coupling, to a broader class of interactions and objects.

- **Through-the-lens camera controls:** these techniques permit manipulating virtual cameras and graphical objects by controlling and constraining points in the image seen through the cameras lens. The methods can be used to address important problems in computer graphics, such as the image registration problem of Section 8.2.4.

- **Generalized snapping:** (Section 8.4.1) the snapping controller permits any attribute of a graphical object to be snapped to precise values, even if they are away from the cursor.

- **Interactive manipulation of color to control object and light geometry:** the use of the lighting model equation as a control permits manipulating light color,

light position, surface colors, and surface geometry. Previous systems contain special case code for such techniques for controlling light colors and surface properties.

- **Generalization of interactive control of reflections:** the methods of Section 8.1.6 permit constraining and directly controlling the positions of reflections on planar and curved mirrors. The only previous technique was restricted to manipulating object translations by dragging reflections in planar surface[HZR+92]. The reflection controls also provide a generalization of the specular highlight positioning methods of Poulin and Fournier[PF92], permitting the positions of specular highlights to be constraints, and to control the geometry of the objects in addition to that of the light source.

- **Generalization of the use of shadows as controls:** Section 8.1.6 introduced techniques to constrain and control object geometry and light positions by controls placed on shadows on both plane surfaces and other objects. This permits a generalization of the Brown interactive shadow widgets[HZR+92] and Poulin and Fournier's positioning of lights by moving shadows[PF92].

- **Scene composition by mixing and matching controls:** (Section 9.7) while some of the scene composition controls have appeared as special purpose hard-coded controls, no previous system has allowed interactively specifying and combining constraints and controls for camera positioning, lighting specification, and object manipulation.

- **Augmented snap-dragging:** Briar (Section 9.1) introduces an extension of previous constraint inferencing techniques that permits automatic generation of a variety of geometric constraints from drawing operations. Conversely, the drawing methods of snap-dragging [BS86, Bie89], are augmented to also specify constraints.

- **Techniques for displaying and editing constraints in a constraint-based drawing program:** Briar (Section 9.1) introduces a number of novel methods for displaying and editing constraints in a drawing program. It introduces a visual syntax for displaying constraints "in-place" in a manner that parallels their definitions, and two methods for deleting constraints.

10.1.6 Graphical Applications

The primary contribution of the applications described in this thesis is to demonstrate the viability of the differential approach, and the flexibility of the implementation. Almost all of the applications are re-implementations of other applications using the techniques of the differential approach.

- **An interactive Tinkertoys application:** the Tinkertoys application permits interactive assembly and experimentation with 3D objects mimicking the real toys, including their kinematic behavior. Previous systems, such as those by Surles [Sur92c] and Schroeder [SZ90], either do not provide for interactive editing of the objects or they do not permit the simulation of the kinematic behavior.

- **User defined shapes with constrained behaviors:** the `define-shape` facility in Bramble permits end users to dynamically define new types of shapes in a 2D drawing program, and have these objects be manipulated without specifying the inverse transformations. Some systems, like the commercial drawing package Visio [Sha93], permit users to define shapes with equations, but do not allow these shapes to be manipulated except by directly controlling the parameters, and do not allow the use of constraints in defining shapes.

10.2 Evaluation

This thesis takes a step towards the goal of improving the quality and range of interactive graphical applications, by providing a substrate on which interaction techniques and applications can be built.

10.2.1 Basic Questions

The central premise of this thesis is that the differential approach can provide an implementation of direct manipulation that uses numerical constraint methods to map controls to object parameters in a systematic fashion, and that several issues in direct manipulation interfaces can be addressed by such an approach.

The differential approach does provide a way to implement direct manipulation, as shown by the recreations of other direct manipulation interfaces using the approach. The approach uses a single numerical solver to map a wide variety of controls to a wide variety of objects. This thesis presents a sufficient set of mathematical and implementation techniques to realize the approach, as illustrated by the prototype implementations. These prototypes show that the methods have acceptable performance on current computers. Traditional direct manipulation operations take only a few milliseconds of computation per frame, and reasonable sized constraint problems can be handled at interactive rates (performance of the prototypes is discussed in Appendix B).

The examples show that the differential approach is interesting. The approach can create desired interfaces, often in ways that are more flexible and general than traditional implementations. The approach also permits the creation of new interfaces that would be difficult or impossible to create with conventional means. For example, despite the utility of methods like the image alignment technique of Section 8.2.4, users are typically provided with controls less suited for the task because of the difficulty

in deriving the mapping from the controls to the underlying parameters. While the special-case example of the table controls for image alignment might have been derived by hand, the more general approach of mixing and matching though-the-lens controls could not have been. In fact, without the view that anything that can be computed can be a control provided by the differential approach, techniques like through-the-lens camera control might never have been dreamed of.

10.2.2 Evaluating Abstractions

The abstractions for creating interfaces provided by the differential approach have many desirable properties:

- the set of abstractions is small;
- they provide concise definitions for many interaction techniques;
- they describe manipulation without reference to the underlying parameterization;
- they can be combined and composed;
- they allow modularity by permitting objects, constraints, and controls to be defined independently and hooked together as needed;
- they map directly to the data structures in the implementation;
- they foster a view of manipulation that helps lead to novel interaction techniques.

The abstractions have their problems and limitations, which will be surveyed in Section 10.2.7.

10.2.3 Evaluating Differential Techniques

Mathematical and numerical methods are crucial to the differential approach. Without methods that solve the constrained optimization problems in a manner that meets the demands of interaction, an approach to graphical manipulation based on constrained optimization is impossible. The prototypes show that methods exist that permit the differential approach to be realized. Section 1.1.4 introduced a number of goals for the methods. Here, we review them:

1. **Flexibility in the types of controls:** the methods permit the use of arbitrary differentiable functions over continuous valued variables as controls. The range of controls discussed in Section 8.1 illustrates this flexibility. In practice, the range of controls is limited by numerical considerations.

2. **Freedom to combine controls arbitrarily and dynamically:** the solver handles arbitrary numbers of simultaneous controls, without concern for what they are. This allows controls to be combined arbitrarily. In contrast to many other methods, there are no restrictions on acyclic dependencies or existence of a procedural solving plan. Adding a new control requires only passing more equations to the solver. The flexibility in combining controls is an essential part of the approach.

3. **Keep the good properties of direct manipulation:** the differential approach allows the creation of interfaces with the good features of direct manipulation such as rapid feedback, continuous motion, and kinesthetic correspondence.

4. **Choose the "best" solution in under-determined cases, and find a "reasonable" answer if there is no exact solution:** in underdetermined cases, the optimization objective defines the "best" solution. The damping methods of Section 3.2.1 handle overdetermined situations, providing robustness in ill-conditioned cases, allowing for redundant controls, and blending conflicting controls.

5. **Provide freedom in picking representations independent from user concerns:** because controls are defined in terms of object attributes, rather than their internal parameters, programmers have the freedom to select representations based on other concerns.

6. **Allow a standard procedure for defining new controls that minimizes the amount of difficult mathematical work in defining a new type of control:** to serve as a control, a function must compute its value and the derivatives of its value with respect to its variables. The derivatives can be computed using automatic methods given the attribute's function, which must be known anyway.

7. **Allow a solving mechanism that is general purpose and encapsulatable:** the solving mechanism for the differential approach works on a set of functions, variables, and controllers that specify desired derivative values. Therefore it applies to problems of controlling differentiable functions of continuous variables. Snap-Together Mathematics encapsulates the solving mechanism, providing the solver as a black box object whose details are hidden from the programmer using the toolkit. Bramble demonstrates this: it does not even provide the application programmer with the ability to look at solver internal data structures.

8. **Work in a variety of domains:** the approach is not specific to any particular domain. A wide range of graphical applications can make use of it.

9. **Be fast and scale well:** as detailed in Appendix B, the prototype implementations show that the methods can provide sufficient performance on the present generation of computers. The $O(n^2)$ computational complexity is a potential

problem. At the present time, models with 50-100 simultaneous constraints can be handled.

10. **Not require sophisticated numerical techniques:** the methods only require numerical algorithms to solve easy forms of standard numerical problems. Standard algorithms from textbooks were sufficient for the prototype implementations. Fancier numerical algorithms, such as an efficient sparse singular-value decomposition solver or a better ODE solver, may improve performance.

10.2.4 Evaluating the Interaction Techniques

This thesis provides tools for creating graphical interaction techniques. The tools do not necessarily lead to better interfaces, in fact, the tools give interface designers new ways to create bad interfaces, for example by applying controls "behind the users back." However, the differential approach would not be interesting if it only permits the creation of bad interfaces.

The differential approach can be used to create good interaction techniques. Although no formal evaluations were done, the techniques demonstrated with the approach include several time-tested standard interfaces, as well as some interfaces that apply direct manipulation to problems that it previously could not be applied to.

Using the differential approach to recreate existing interaction techniques is not necessarily overkill: the methods provide a new way to define these techniques in a representation-independent fashion and permit the techniques to be coupled with other constraints. The approach can also make implementation easier as it does not require the programmer to derive the mappings from controls to parameters.

A more exciting use of the differential approach than recreating existing interfaces is creating of new interaction techniques and constraint-based interfaces. I believe that the differential approach can often lead to interesting new techniques for graphical manipulation tasks. The approach has the benefits that:

1. it preserves the continuous motion animation and rapid feedback properties generally desired in graphical interfaces;

2. it permits the creation of controls directly relevant to the users task;

3. it speeds experimentation with new control types as the mathematics does not need to be derived for each new one;

4. it speeds experimentation by allowing controls to be combined.

The image registration interface of Section 8.2.4 exemplifies all these points. It provides an interface to an important problem, for which conventional interactive controls are not convenient. The through-the-lens controls are directly relevant to the task, as

they permit directly specifying where in the image a point in the 3D scene should appear. The creation of the through-the-lens control required knowing only functions that were needed for drawing[1]. Initial experimentation with the alignment technique merely required applying 4 through-the-lens controls. Since through-the-lens controls were already available in the toolkit, the only new code required to create the interaction technique was code to place the video image as the screen background.

The image alignment technique would have been very difficult to create with traditional approaches for implementing direct manipulation. Deriving the mapping from controls to camera parameters is extremely difficult, as can be seen in the photogrammetry literature mentioned in Section 2.4.2. The more general approach of mixing and matching controls, used in Showoff (Section 9.7), would be impossible because such derivations must be done for every new combination of controls.

Unfortunately, the range of interaction techniques that I have developed with the tools of this thesis is not as extensive as I had hoped. Much more of the effort of the thesis research went into tool building. Through-the-lens controls were the biggest success. Most other techniques, such as generalized snapping, the scene composition controls such as shadows and reflections, and 3D widgets were not explored sufficiently well to make persuasive evaluations of. The success of color controls, something I had high hopes for initially, was thwarted by technical problems. Numerical issues cause the lighting equations not to serve well as controls. Further exploration of these controls may find solutions to these difficulties, especially now that the solver has damping correctly implemented.

Generalized snapping, described in Section 8.4.1, is another interaction techniques that I was unable to explore sufficiently. Some issues are described in Section 10.3.

10.2.5 Evaluating the Architecture and the Prototypes

The prototype implementations described in this thesis consist of a few major parts. Code sizes listed are for C++ source files, with comments, not counting headers.

- **Math and Data Structures Library —** (5K lines of C++ in 22 files) My C++ math and data structures library includes various basic mathematical elements such as matrices, vectors, and ODE solvers as well as more general data structures such as lists, queues, and hash tables.

  The math library has been used extensively over the past 5 years in a variety of applications. Particularly good design features were the object-oriented encapsulation of sparse matrices and ODE solvers. The code is very portable, and runs on a range of machines from notebook PCs to high performance workstations.

[1] Arguably, the viewing transformations are in the graphics toolkit, and therefore not known to the applications programmer. However, under this argument, through-the-lens controls are also in the toolkit, so their functions are not known to the programmer.

It is significant to note that despite the more than 50,000 lines of C++ code in the complete system, for reasonable sized problems, the majority of time is spent in a few inner loops of the routines in the library. A fast and robust implementation of basic mathematical data types such as vectors and matrices is crucial to the differential approach. Fortunately, such routines are not hard to create and can be nicely encapsulated in a manner that allows reuse.

- **Snap-Together Math Library —** (3K lines of C++ in 16 files, several hundred automatically generated by BlockMaker, 300 lines of Mathematica to implement BlockMaker) The Snap-Together Mathematics library is discussed in Chapter 5. It supports function composition and evaluation, the differential solver, and a number of predefined function blocks.

  Snap-Together Mathematics has proven to work really well. The tool has evolved since the earliest versions, but the basic design is the same. Sparse sparse vector passing, global time-stamps, and the minimal protocol have all remained. The library has been used for a number of applications besides Bramble, including Spacetime motion control, physical simulation, and statistical data fitting.

  The biggest successes of Snap-Together Mathematics are the places where it is the simplest, such as the minimal protocol and the global time-stamp cache validation. Better support for features like partitioning and representation switching should be easy to add. One design flaw is that Snap-Together Mathematics objects, in particular function blocks, are a bit "heavy." That is, they can be expensive to create and destroy.

- **Whisper Interpreter —** (9K lines of C++ in 18 files, 3K of which were generated semi-automatically by EMACS macros to implement primitives for Snap-Together Mathematics and GL, 800 lines for an interface to an image processing library) The Whisper interpreter is a C++ library for embedding in interactive applications. The library contains interfaces for Snap-Together Mathematics, the GL graphics library, and a small image processing library. There are approximately 400 built-in primitives and 20 predefined types, not including what Bramble adds.

  Whisper has been a very useful tool for the research described in this thesis. It has been particularly pleasant to program in and to use as a basis for Bramble. Because it was designed for embedding, it provided the right set of functionality to the systems it was put into. It is extremely easy to extend. The simple dynamic object systems is particularly well suited for the kinds of exploratory programming done in an experimental toolkit like Bramble.

- **Bramble —** (40K lines of C++: including 15K lines of C++ for basic parts including some automatically generated code, 5K lines C++ for the 2D library including automatically generated code for a variety of objects, 7K lines C++ for

the 3D library, 7K lines of automatically generated function blocks, 700 lines of Whisper library code) Bramble is the C++ interactive graphics toolkit described in Chapter 7.

Bramble has been an interesting testbed for the development of the differential approach. It has facilitated experimentation with the variety of interaction techniques and applications discussed in the thesis. I think the design of Bramble is quite solid. However, the implementation itself was hastily put together, and shows signs of its unusual evolution, for example before Whisper, other mechanisms for many of the object services had to be provided and remnants of these still haunt Bramble.

The abstractions of Bramble were sufficient to create a variety of applications without bypassing the approach. This is best illustrated by Toys which contains many features, but is written entirely in Whisper. Bramble is missing many of the features of standard toolkits, such as text handling. This makes it hard to build complete applications.

- **Applications —** (230 lines of Whisper for `Boxer`, 800 lines of Whisper for `Mechtoy`, 500 lines of Whisper for `Poly`, 500 lines of Whisper for `ShowOff`, several hundred lines Whisper for `Showoff` demos, 900 lines of Whisper for `Tinkertoys`) The complaint with the applications is that there are too few, that they are too small, and not "complete" enough. They do demonstrate the differential approach. However, there is much more to a graphical application than manipulating graphical objects. I have been disappointed by my inability to find a "killer" 3D application.

I believe that the architectural organization of the system works out very well. Snap-Together Mathematics provides a good degree of separation between the solver and the client application. It also allows objects to be defined independently, yet still connected together, by providing a common protocol.

Besides Snap-Together Mathematics, the most significant architectural decision in Bramble was the use of Whisper. I believe that this is a good strategy. The dynamic object system is extremely useful in a graphics toolkit. The facility for rapid prototyping is extremely useful in an experimental system. The embedded interpreter approach also facilitated additions to the toolkit: each new feature or object type did not require extensive installation into all of the applications. Instead, each new feature simply defined new primitive function calls that could be called as needed from applications, or even interactively from the interpreter prompt. The same functionality might have been attainable if a development environment such as LISP or Scheme were used, however, such environments did not offer the floating point performance of C++, the ease of transporting executables, the availability of graphics and numerics packages, nor were good compilers available at the time that the project was begun.

As in the process of constructing any system of this size, a lot of design decisions were made along the way. Some of the best design decisions that I made were (in no particular order):

- the simple protocol for Snap-Together Mathematics;
- using an embedded interpreter in the toolkit;
- using the Whisper object system for graphical objects;
- half sparse matrices;
- using a conjugate-gradient solver;
- the scatter gather handling of state variables.

Some of the worst design decisions I made were:

- using the simple GL event model of handling events globally, rather than handling events on a per-window or per-object basis. Also, GL does not time-stamp events, nor record the mouse state when an event occurs. This causes problems when event handling is delayed as the mouse may move between when the event happens and when the event is processed;
- having a single world, rather than explicit scene graphs like Inventor [SC92]. This makes it impossible to edit multiple documents, and difficult to show different data in different views.
- creating my own widget library. Many toolkits containing basic widgets such as buttons and sliders are available. Using these would have given more attractive looking widgets, and probably some richer functionality such as file-selector dialogs.

10.2.6 Experience with the Implementations

Because of the large overhead of making tools available to others, an explicit decision was made in the research plan for this thesis not to make the tools available to others.

After many requests, I have made the Snap-Together Mathematics library publicly available by anonymous FTP access. Because I did not offer support or documentation, I suspect that few people have made extensive use of it. A small number have reported successful experiments built with Snap-Together Mathematics. One CMU undergraduate student was able to build a portable 2D version of Point Tinkertoys that ran under the X window system, without learning anything about the constraint mathematics. I

also know of at least three re-implementations of Snap-Together Mathematics by researchers who had access only to the papers. These researchers were able to build 3D constraint demonstrations.

Bramble was not offered to others because of the amount of support it would require. Because of this, I cannot make any claims about how "usable" Bramble is. Personally, I find Bramble a joy to use. Part of this may be attributable to the fact that I designed Bramble based on my personal tastes and program development style. The interpretive environment and high level language of Whisper extremely appealing compared to C++. However, I believe that a good part of Bramble's attraction is how well the abstractions fit the needs of graphical application development.

Other than myself, the main users of the applications in Chapter 9 have been people who have "taken over the drivers seat" during demos. This has allowed informal assessments of how users react to interaction techniques created with the differential approach. Such users are atypical: since they are only playing with the applications for a few minutes, their usage patterns are different than those of users' attempting to solve real problems. For example, many people permitted to use the systems immediately to try to torture the solver by making impossible connections to see what happens.

Response to the applications has been very positive. Few complain about the lag between the pointer and the object being dragged. In fact, many comment on how they like the feel of it. Some things, such as dragging the spiral of Section 8.1.1 generally require a little practice before people can achieve the effects that they intend. However, almost everybody has been able to figure it out without instruction. There is little else to compare these systems to in terms of usage.

I am consistently impressed by how quickly people learn to use the mousepole after getting past the initial issue of using multiple buttons on the mouse simultaneously. In general, people seem to like the Bramble standard 3D interface. Part of this may be attributed to relative novelty of interactive 3D graphics, and to the attention to aesthetic details such as the use of shadows and the selection of colors.

10.2.7 Limits and Drawbacks

The drawbacks of the differential approach, as seen in this thesis, fall into three categories:

1. fundamental limitations or drawbacks of the approach;

2. limitations and drawbacks of the techniques that should be resolved by future work;

3. artifacts of the prototype implementations.

The fundamental limitations of the approach are:

- **The differential approach is time continuous.** Things do not happen instantaneously. While this is generally a feature, some times we want objects to jump to their destinations, for example, when a new object is created.
- **The differential approach only is applicable to continuously valued parameters and attributes.** The numerical methods do not apply to discrete data. Some alternate mechanisms, such as propagation constraints, must be used.
- **The methods do not scale as well as those for other approaches.** Without restricting the problems, the complexity bound of $O(n^2)$ seems to be unavoidable.

Some major limitations that I think can be addressed in future work are:

- **scalability:** While $O(n^2)$ is the limiting factor, methods to reduce n and the constants can allow solving larger problems. Implicit constraint methods can potentially enhance scalability considerably, as demonstrated in the Tinkertoys application. Also, it is unclear how large the number of controls a system must handle. For predefined objects and interaction techniques, the number of controls will be a (usually small) fixed number (e.g. it will not grow with problem size). However, with a constraint-based interface, a user may create arbitrary numbers of controls. But, this number may not be unbounded as eventually, the limits of how much simultaneous behavior a user can comprehend may be reached.
- **precision:** Methods for more precisely achieving the user's goals will facilitate many applications. Generalized snapping is a first attempt.
- **generality:** As described in Section 8.1, mathematical considerations cause some functions to work better as controls than others. Some formal characterization for which controls which and what do not would further simplify the construction of interfaces.
- **real-time:** Adding a coupling to real (e.g. wall-clock) time would increase the expressibility of the abstractions. Interface designers would gain control over the time constants involved in the approach.
- **integration:** The differential approach applies only to the continuous motion dragging parts of applications. Therefore, integration with other techniques, such as propagation constraints, to handle other parts of applications is essential.
- **hierarchy:** The methods for differential constraints provide only two levels of constraint hierarchy. The work of Borning et al.[BFBW92] shows the usefulness of more levels.

Many of the problems in the prototypes are artifacts of the implementations, rather than problems with the approach. They fall into three general categories:

- **Bad design decisions:** Several bad design decisions were listed in Section 10.2.5.

- **Artifacts of evolution:** The prototypes evolved from earlier versions, so in many places there are leftovers from early versions. Bramble's property sheets and Snap-Together Mathematics's confusing mechanisms for soft controls and generalized snapping are two examples.

- **Incompleteness:** The prototypes were built in time to finish a thesis, so many features that are not essential to the points of the thesis have been omitted, such as text-handling in Bramble.

10.3 Directions for Future Work

This section discusses some topics to extend the differential approach, make effective use of it, or repair its deficiencies.

For this thesis, a number of prototype implementations were constructed. An entire class of future work involves the creation of more "industrial strength" tools and applications that can be widely distributed. For Snap-Together Mathematics, this may mainly involve completion of some partially implemented features, documentation and support. However, for Bramble, it is probably not worthwhile to go through the effort of making a distributable tool since too many design decisions were made with a "get it done for the thesis" attitude. This section focusses on future work on the differential approach more generally, rather than on the specific artifacts of the toolkit.

10.3.1 General Issues

The differential approach's model of time does not correspond with real time. This limits what can be expressed with the approach. For example, it makes it impossible to specify desired rates in a meaningful way. Some mechanism for correlating simulation time and real time would be a useful addition to the approach. However, it would require tackling a number of difficult technical issues in synchronization, especially with the variability of the solving methods. This issue will also be increasingly important in future multi-media applications. Faster processors might help with synchronization issues since programs will have the possibility of finishing their work quickly and then waiting for synchronization. However, such "busy waiting," is not the most effective use of processor cycles, and fails to handle cases where the fast processor's performance is needed to handle larger problems.

Future high-performance computers will make the lack of real-time coupling difficult because things may happen too fast. It will be important to provide the interface designer with some control over the time constants of the interactions.

The approach lacks a good characterization for what makes a function work well as a control. The rules of thumb from Section 8.1 need to be formalized. If the characterization could be sufficiently codified it would allow an even more automated tool for creating new controls to be created.

Precision in manipulation is another issue for the differential approach. Even for `Snap` and `Click` controllers (Section 6.4), the exactness of solving is limited because controllers only get to specify velocities at sampled increments. For greater precision, controllers might need to be given some information about step size. This is also important for better tracking of input devices as it would permit better prediction. Enhancing mouse tracking, for example by predictive filtering, is another area for study.

The lag between pointing device and dragged object needs to be better understood. In applications where the lag is detrimental, it might be reduced by techniques such as predictive tracking. Developing predictive tracking strategies for input devices will be generally useful, not only for the differential approach, but also for things such as remote collaboration.

10.3.2 Mathematical Techniques

The weakest link in the mathematical techniques is the way that over-constrained systems are handled. Many of the deficiencies of the damping techniques are listed in Section 3.2.1. An alternate method of dealing with the ill-conditioned or singular matrices might be found. To date, I have not found any that retain the performance characteristics of the methods discussed in this thesis on well-conditioned problems.

A primary drawback of the damping methods as described in this thesis is that they also affect constraints that are not ill-conditioned. This drawback can be avoided by using methods, such as the one presented by Nakamura[Nak91], that adaptively set the amount of damping. These methods are expensive, as they effectively must solve the system to determine its condition. Another strategy would perform some potentially expensive computation to determine if there are bad constraints, and applying the damping methods only in these cases. The hope would be that the expensive computations would not be needed often or could be computed incrementally.

Alternate methods for handling over-constrained matrices, such as singular value decomposition, have better numerical properties than damping, but would require work to make them as efficient. Similarly, more reliable methods for handling inequalities, such as those of Baraff [Bar94], could be applied if efficiency concerns were addressed.

Another deficiency of the methods presented in Chapter 3 is that they offer few opportunities for levels of importance of controls (hierarchies in the terminology of Borning et al[BFBW92]). Soft controls provide two levels of constraints, but require better methods to give them the degree of control of hard controls. Extending differential techniques to support more levels of hierarchy would be useful.

The methods of the differential approach are designed to work with the lowest com-

mon denominator of general functions, avoiding special cases and anything beyond the minimal information about the functions. Each of these restrictions could be relaxed to create methods that better handled important special cases. For example, special methods for articulated figures exist. Also, more information about functions, such as higher derivatives and intervals, could also be used.

The techniques provided for handling inequality constraints are really a hack, and would be improved by using numerical methods better suited for this task. Finding methods with suitable performance characteristics could be challenging.

10.3.3 Numerical Techniques

As computers become faster, the size of the problems that can be handled at interactive rates will increase. This means even more of the computation time will be spent in the complexity limiting step of solving the linear systems. Selecting appropriate algorithms will be increasingly important. Other conjugate-gradient solvers, such as those from [PS82] might apply. In particular, some of these methods may better handle ill-conditioned or singular matrices, allowing less use of damping techniques. Dynamic selection among multiple solvers, as done in Converge[Sis90], may also provide better performance, particularly when the problem can be partitioned.

The best weapon against the computational complexity of solving the linear systems will be to reduce their size while giving the user the illusion that the entire problem is being solved. There are many possibilities for new implicit constraint mechanisms. For example, representations of objects could be dynamically switched to maximize the number of constraints that are represented cheaply. Algorithms similar to multiple-output multi-way local propagation solvers, like [San94], could implicitly solve as many constraints as possible. Techniques that divide the problem based on numerical results are possible. For example, Sistare[Sis90] partitions constraints empirically. The solver attempts to solve the constraints on a small subset of the objects. If these objects do not have sufficient degrees of freedom, more objects are added to the subset.

Better methods for solving the ODEs would enhance the stability of the constraints and permit a wider class of controls. Making use of knowledge about the expected behavior of the objects should provide information that can be used to adapt the step size of the solver. Finding ways of making adaptive step sizes unobtrusive to the user might be required to make them acceptable.

The simple caching mechanisms of Snap-Together Mathematics are reasonable for the experiments conducted so far with the differential approach. Cache analysis shows varying performance of the global cache validation scheme. A better caching scheme, perhaps based on incremental attribute evaluation[Hud91], might be more efficient. However, as problem sizes grow, the cost of the Snap-Together Mathematics computations will become smaller relative to that of solving the linear system. Therefore, extensive optimizations are unwarranted. Better support for switching representations

would be another improvement in the Snap-Together Mathematics implementation.

10.3.4 Interface Toolkits

The differential approach addresses only a part of the problem addressed by modern interface toolkits. If the differential approach is to be used in a complete toolkit, it must be made to integrate cleanly with existing approaches. Employing propagation solvers for discrete data inside the differential approach seems to primarily be an engineering problem. However, incorporating the differential approach, with its different model of time, into a conventional event-driven toolkit poses difficult questions such as how to integrate discrete state changes into the continuous flow of time.

The ways that the Whisper interpreter supports Bramble bring interesting questions as to the architecture of graphics toolkits. Fast turnaround is only one of the reasons for the widespread acceptance embedded interpreters in graphics toolkits. However, much of the utility of the Whisper/Bramble connection is that many of the services that Bramble must provide, such as object management and on-the-fly definition of behaviors are similar to those provided by the run-time support for a modern programming language. An interesting strategy might be to design run-time support specifically for interactive graphical applications.

10.3.5 Interaction Techniques and Applications

While the differential approach has many flaws that may be addressed with future research, the techniques and tools are evolved enough that they can be used to create novel interaction techniques and applications. The invention of new graphical techniques will be the real payoff of the approach. Developing new interaction techniques may require finding new attributes of objects to constrain and control, new combinations of these controls, and new ways to present the controls and constraints to users.

Generalized snapping is a feature of the differential approach that has not been sufficiently explored. There are many issues in making it into a successful interaction technique, such as how to provide adequate feedback to the user and how to determine the many parameters. However, I think it is still a promising technique.

The availability of a variety of controls and the ability to combine these controls as constraints opens up a new domain of questions about how to select, specify, display, and edit the controls and constraints. Some tasks, like scene composition, seem particularly appropriate for interfaces that give palettes of controls to users. I think the strategy of manipulation from structure is a promising way to help make these tasks easier by building more constraints into the system before the user has to specify anything.

Good controls might also serve to facilitate manipulation when the "user" is some computational mechanism automatically controlling the graphical objects. For example, an automatic picture composition system like those described by Feiner[Fei93]

might be easier to develop in terms of controls like "point the light at this object" rather than directly altering the underlying parameters of the objects.

The ability to handle controls asynchronously may help explore multi-input device interaction techniques. In general, the approach may serve to help incorporate new input mechanisms as it decouples the input device from the objects being controlled.

There are many questions which must be addressed before constraint-based interfaces for drawing and modelling can be successful. Constraints must be made easy to display, debug, understand, and edit. While restricted problems, such as permitting users to define the behaviors of individual objects, simplify these issues, they do not remove them.

The techniques for specifying, displaying, and editing constraints introduced in Briar also deserve to be examined in more detail. Future work must address issues in extending the techniques to larger classes of constraints, scaling them to larger models, and applying them in 3D.

10.3.6 Usability Evaluation

This thesis provides little in the way of formal evaluation of the differential approach. Because the new interaction techniques were meant only as examples of the differential approach, not as wide-ranging solutions to interface problems, it was not appropriate to study the usability of these techniques. However, because some show promise as interaction techniques, more formal evaluation may be useful.

Much of the lag between pointing device and target object in the differential approach can be reduced with better tracking. However, many users report enjoying this "spring" behavior. A study to evaluate the usability of the spring dragging might be interesting. Similarly, the usability of constraint-based graphics has never been formally studied. Such a study might determine people's abilities to use multiple simultaneous controls, or help understand how complicated a constrained model might be created by a user before comprehensibility is sacrificed.

10.4 Final Remarks

This thesis has provided an approach to implementing direct graphical manipulation that uses the numerical and graphical performance of modern processors. The approach views manipulation as an equation solving or constrained optimization problem, allowing interactions to be defined in terms of objects' attributes, rather than their representations. To implement the approach, mathematical techniques for solving the optimization problems had to be selected and implemented in a way that addresses the issues of interactive systems. This implementation permitted encapsulating the mathematics in a manner that allowed the construction of a graphics toolkit that hid the mathematics of

the approach from the applications programmer. With the approach and the prototype implementations, a number of interaction techniques and applications were created.

Direct graphical manipulation has been widely successful, and future systems offer new possibilities and issues for these interfaces. As decreasing cost makes high performance computers more widely available, graphics tools will have a wider potential audience. Many of these users will require more fluent and transparent interfaces for types of tools now only used by experts. New classes of applications, such as virtual reality, offer whole new classes of issues. New users and new applications will challenge interface designers. The ad-hoc methods traditionally used to devise manipulation techniques will hinder the development of new interfaces. This thesis has offers an alternative that can provide a substrate for future direct manipulation interfaces.

The world is your exercise book, the pages on which you do your sums. It is not reality, although you can express reality there if you wish.
You are also free to write nonsense, or lies, or to tear the pages.

— Richard Bach
Illusions, p. 127

Appendix A

The Whisper Programming Language

An important tool in the development of the differential approach has been a simple interpreter for a language called *Whisper.* Whisper serves four main purposes in this work:

1. An embedded interpreter is important for interactive systems to realize features such as saving and loading of models and configuration files.

2. The language runtime provides useful tools for system construction, such as a dynamic object system and dynamic function definition.

3. The language provides a convenient notation for describing interaction techniques in this thesis.

4. The interpreter permitted avoiding many of the problems with the programming environment available to develop this work, such as slow turnaround and bad debuggers.

Whisper is not directly connected to the differential approach. However, it is discussed here for several reasons. Foremost, the language is discussed enough so that a reader can understand the examples written in Whisper in various sections of the thesis.

The Whisper interpreter is specifically designed for being embedded into applications. The primary design goal were simplicity and extensibility, even at the expense of performance. Although performance of the interpreter is poor, the ease of extensibility allows adding speed critical code as new primitives written in the host language, C++. New data types can also be added easily to the interpreter. Once the core interpretive language was built, extensions were created to provide access to the Iris GL graphics library, Snap-Together Mathematics, an image processing library (that is not used in this thesis), and Bramble.

A.1 Whisper Basics

Whisper is a lexically scoped variant of LISP, like scheme. Whisper is very similar to other languages in this family. A basic introduction to programming in such a language is provided by Friedman and Felleisen [FF87]. Rather than providing yet another tutorial for such languages, we instead look at the differences between Whisper and more common dialects that are relevant to the code examples in the thesis.

Like other LISP dialects, Whisper has dynamic types. All data are tagged with their type, and the types can be determined for any object. Whisper's type system is extensible at compile time only. The basic set of types provided by Whisper includes integers, floating point numbers, strings, pairs, points (3 floating point numbers), closures, built-in primitive functions, clocks (special objects for timing) and environments. Types for basic Snap-Together Mathematics classes are provided, as well as for Bramble objects. Almost all Bramble objects have the Whisper type `IDObj`, but the subtypes can easily be determined.

Whisper's special forms for setting variable values are different from most LISP dialects. Whisper provides a `set` function that sets the value of a variable. If the variable is defined in the current scope, it is simply rebound to the new value. If the variable is not defined, it is created as a *global* variable. The `defun` function used to define a function is simply shorthand for `set`. The `bind` function acts like set, but it always sets a variable in the most local scope.

Whisper is lexically scoped. Variable bindings are determined by the code that surrounds them in the program text. For example,

```
(let ((a 5))
    (defun f (x) (+ x a)))
```

defines a function that adds 5 to its argument. An *environment* is the object that embodies a scope, that is, it is a pairing of variable names and values. Environments are hierarchical, that is, each environment refers to the environment it was defined in. Unbound references are deferred up the chain of environments until the global scope is reached.

One important feature of Whisper is that it permits treating environments as first class objects. Many dialects of Scheme, such as MIT Scheme [HtM93], do this as well, and the introductory programming text by Abelson and Sussman [AS85] provides an introduction to the use of first-class environments. Whisper's syntax and operations for first-class environments is different from Scheme's.

The `environment` special function returns a reference to the current environment that can be passed around as any other data element. For example,

```
(set e (let ((a 5)
             (b 6))
         (environment)))
```

sets the variable `e` to an environment that binds `a` and `b`. Various operations can be performed on environments, including adding additional bindings, inquiring as to their contents, combining two environments, and printing the contents of an environment.

The two most important operations that can be performed are to evaluate an expression within an environment, and to bind a value in an environment. To add a binding to an environment, a special version of the `bind` primitive function is used that takes an extra argument for environment, for example,

```
(bind-in e c 10)
```

which would add the binding $c = 10$ to the environment defined in the previous example. A similar version of `eval` is provided,

```
(eval-in e (+ a b))
```

which for the example would return 11. Whisper provides the syntax

(*env* $expr_1$ $expr_2$ ... $expr_n$)

to evaluate expressions $expr_1$ through $expr_n$ in succession inside of the scope of environment env. The value of the evaluation of the last expression is returned. This construction violates lexical scoping, for example,

```
(let ((a 1)
      (b 2))
  (e (+ a b)))
```

with `e` defined as in the above example evaluates to 11, not 3. This can cause an important distinction between the use of `bind` and `bind-in` . For example,

```
(let ((a 10))
  (bind-in e f a)
  (e (bind g a))
```

creates bindings for `f` and `g` inside of `e`, however, `f` will be bound to 10, while `g` will be bound to 5, because its binding statement was evaluated inside of `e`.

First class environments function as Whisper's object system. Fields and methods are stored as bindings in the environment. The object creator makes an environment and binds variables to their initial configurations. While there is no explicit mechanism

to provide inheritance or delegation, such functionality can be provided in the creator functions.

A Bramble `IDObj` is not an environment, however each object carries an environment around. The `get-env` function returns the environment carried by an `IDObj`. Whisper syntax permits using the evaluation notation for Bramble objects as environments, so

(*IDObj* *expr*)

is a shorthand notation for

((get-env *IDObj*) *expr*).

A.2 Some Examples

In this section, we take a few examples of code fragments from Whisper programs to explain various features of the language, and how they are used with Snap-Together Mathematics and Bramble.

This first example defines a function that builds the function block graph for the line segments shown in Figure 5.1. The line segment is returned as a Whisper object, that is, as an environment. Notice that an environment is made first and then the intermediate values are defined in a different environment. This way the environment that will represent the line segment does not contain the intermediate values, only the connectors.

```
(defun make-line ()                                          ;
  (let* ((q (make-stobj 3))                                  ;state vector w/3 spaces
         (e (environment)))                                  ;empty "object" to put things in
    (let* ((c  (cos-block (signal q 2)))                     ;make blocks to compute
           (s  (sin-block (signal q 2)))                     ;   intermediate results
           (lc (times-block c (signal q 3)))
           (ls (times-block s (signal q 3))))
      (bind-in e left-x  (plus-block lc (signal q 0))) ;make connectors
      (bind-in e left-y  (plus-block ls (signal q 1))) ;   put the blocks directly into
      (bind-in e right-x (plus-block lc (signal q 0))) ;   the object
      (bind-in e right-y (plus-block ls (signal q 1))))
    e))
```

Calling this procedure returns an environment that contains 6 bindings: 4 for the connectors, 1 for the state vector and an extra binding that is a reference to itself. To use these objects to create the wiring of Figure 5.1, given a function to perform the

attach operation which would take 4 connectors as inputs and create the function block tree:

```
(set rod1 (make-line))
(set rod2 (make-line))
(set nail (attach (rod1 left-x) (rod1 left-y)
                  (rod1 right-x) (rod1 right-y)))
```

Attach would return an object (a Whisper Environment) that contained two connectors. It might be defined as follows:

```
(defun attach (x1 y1 x2 y2)
   (let ((c1 (minus-block x1 x2))
         (c2 (minus-block y2 y2)))
     (environment)))
```

The following code fragment is a more expanded example of constraint inferencing using Bramble's snapping than the one given in Section 7.7.3. It is a Bramble event handler that is used to draw lines with rubber banding and automatic inference of constraints.

```
(1)  (add-key dev-rightmouse k-none k-down              ; define button down handler
(2)    (lambda (v)                                      ;
(3)      (let* ((s (snapdp))                            ; get snap point
(4)             (p (if s (where s) (cursor-mapw view))) ; start drag at snap or mouse
(5)             (l (make-2d-line (p-x p) (p-y p)        ; create object at start
(6)                              (p-x p) (p-y p)))      ;
(7)             (d (pt-eq-2d (l end1) (v mouse-port)))) ; connect one end to mouse
(8)        (if s (pt-eq-2d (l end2) s))                 ; if snap, infer constraint
(9)        (add-key dev-rightmouse k-any k-up           ; define button up handler
(10)         (lambda (v)                                ;
(11)           (let ((s (snapdp)))                      ; get snapped point
(12)             (delete d)                             ; delete drag constraint
(13)             (if s (pt-eq-2d (l end1) s))))))))     ; if snap, infer constraint
```

This fragment is a call to `add-key` , the Bramble primitive for defining key handling events. The first three arguments specify that this call is to define the right mouse button press (down) event with no modifier keys. The last argument to the call is a closure that is to be called with one argument when the event occurs. The argument specifies the view that the event occurs in.

Lines 2-12 define the procedure that is called when the right mouse button is pressed. First, a number of local variables are bound to various quantities useful in the operation: `s` is bound to the current state of the snap-server; `p` is bound to either the position of `s`, if there is a point snapped to, or to the position of the cursor; `l` is bound to a newly

created line segment, created with both endpoints at `p`; and a constraint is created connecting one endpoint of `p` to the position of the mouse in the current view and stored in `d`.

Line 7 performs a constraint inference. If the cursor is snapped to a point when the line is created, the end of the line is connected to that point with a point equality constraint.

Lines 8-12 redefine the event handler that is called when the right mouse button is released. Because of Whisper's lexical scoping, this code is executed in the environment created by the key down handler and has access to those local variables. It first deletes the constraint that was connecting the line segment to the cursor. If the cursor was snapped to a point when the button release occurs, the line segment is attached to this point in line 12.

A.2.1 The Define-Shape Syntax

A special facility is provided in Bramble to facilitate the creation of 2D shapes. The process is inspired by the shape spreadsheets in the Visio drawing program [Sha93], but has greater utility because it permits placing constraints inside objects and using any point as a handle. The `define-shape` special function takes a description of a shape and automatically generates two functions: one that creates instances of the object, and another that draws a prototype version of the shape suitable for displaying in an icon. The `define-shape` primitive is special because it does not evaluate its arguments in the standard way.

The syntax of `define-shape` are as follows:

(define-shape $name$ $variables$ $defaults$ $lets$ $command_1$ $command_2$ ...)

where:

name is a string that names the type;

variables is a list of state variable names;

defaults is a list of initial values for each variable;

let is a list of let pairs to define internal variables;

command is a pairing of commands and data.

For example, a simple rectangle can be defined by:

```
(define-shape rectangle (w h) (.5 .25)
    ((w2 (/ w 2))
     (h2 (/ h 2))
     (mw (- 0 w2))
     (mh (- 0 h2)))
    (spath ( (w2 h2) (mw h2) (mw mh) (w2 mh) )))
```

All shapes are defined in their local coordinates, so the rectangle only has 2 state variables, width and height. Default values for these are provided. The let list defines 4 local variables. The `spath` command defines a list of vertices that are connected to draw a polygon, with handle points placed at each vertex. The command is equivalent to separate commands to define a drawing function and handle points. Literally, the definition used for drawing is also used for manipulation.

The `define-shape` mechanism makes extensive use of Whisper first-class environments to make the concise specification possible. Each variable declaration and let list clause defines a new symbol in the objects local environment. Each of these is bound to a signal: when the `define-shape` is executed, it is run in a special environment that shadows all of the basic arithmetic operations with their Snap-Together Mathematics block generating counterparts. Because the drawing routines do not execute during the `define-shape` , they can do normal arithmetic.

Other commands permit more general drawing commands, explicit specification of handle points, and generation of constraints on the object. The fuel-gauge widget of page 162 demonstrates many of the features of `define-shape` . Here we present a more complete version:

```
(1)  (define-shape gas-gauge (sz theta) (.35 0)            ; 2 params
                                                           ; local variables
(2)    ((l (* (one-way sz) .9))                            ; size
(3)     (t (+ .2 (* 2.7 theta)))                           ; angle (in radians)
(4)     (x (- 0 (* l (cos t))))                            ; positon of needle
(5)     (y (* l (sin t))))
                                                           ; draw function
(6)    (drawf (prog (color gl-white) (arcf 0 0 sz 0 1800)  ; white arc for back
(7)                 (color gl-black) (linewidth 3)         ; black border
(8)                 (arc 0 0 sz 0 1800)                    ; arc border
(9)                 (move (- 0 sz) 0) (draw sz 0)          ; bottom border
(10)                (icon-font)                            ; set text font
(11)                (cmov (* l -.9) (* l .2)) (fmprstr 'E) ; place labels
(12)                (cmov (* l .8) (* l .2)) (fmprstr 'F)  ;  for E and F
(13)                (move 0 0) (draw x y) ))               ; draw needle
                                                           ;
(14)   (> theta 0)                                         ; limit to legal values
(15)   (< theta 1)                                         ;
(16)   (handle x y))                                       ; create handle on needle
```

The gas gauge has 2 parameters, one for the size of the gauge, and one for its current value. The let clause of the `define-shape` defines a number of intermediate variables using arithmetic operations. One special operation is `one-way` that defines that controls on its output should not effect its input. The `one-way` function block returns the value of its input, but always returns a zero derivative. For the gas gauge, the one-way is used so that dragging the needle of the gas gauge does not change the size of the gauge. The `drawf` command defines a code fragment that is used to draw the gauge. The list of statements are calls to the GL graphics library. The > and < commands define constraints, and the `handle` command creates a `DistinguishedPoint` connector at the specified position on the object, which in the example is the tip of the gauge's needle.

The `install-shape` function is defined by certain applications to automatically install an icon for creating shapes defined with `define-shape` . It simplifies using `define-shape` in an application. The `install-shape` function takes the values returned by `define-shape` and creates an icon and the proper handlers for creating new instances of the shape.

The real purpose of scientific method is to make sure Nature hasn't misled you into thinking you know something you don't actually know.
If you get careless or go romanticizing scientific information, Nature will soon make a complete fool out of you. It does it often enough anyway when you don't give it opportunities.

— Robert Pirsig
Zen and the Art of Motorcycle Maintenance, p. 94

Appendix B

Performance of the Implementations

This appendix provides some empirical tests measuring the performance of the prototype implementation. The absolute performance numbers are, in a sense, not important as the hardware and the implementation are continually changing. These benchmarks are provided to:

- Provide empirical evidence of the scaling properties of the techniques, as analyzed in Section 4.2.
- Give some idea of the scale of problems solvable with currently (circa 1993) available hardware.
- Provide some intuition for where the performance bottlenecks are.

Understanding the performance of a program running on a modern, high-performance workstation is a difficult task. Complicated factors such as memory hierarchy organization interact in complex ways. For example, different problems may distribute themselves differently along the lines of the data cache. Although we have attempted to design benchmarks that reduce these types of effects, any numbers must be viewed with some caution.

The benchmarks were run on a Silicon Graphics Indigo 2 workstation, except where noted. The machine had a 150MHz R4400 processor and 32 megabytes of main memory. All of the problems fit into main memory on the machine, so no paging occurred. The clock accuracy on the workstation is 10 milliseconds. Where greater precision is reported, it was found by averaging over enough trials that the extra digit remains stable as more trials are run. Some of the simpler benchmarks were run on a Silicon Graphics Personal Iris 4D/25, with a 20MHz R3000 processor, 16 megabytes of main memory,

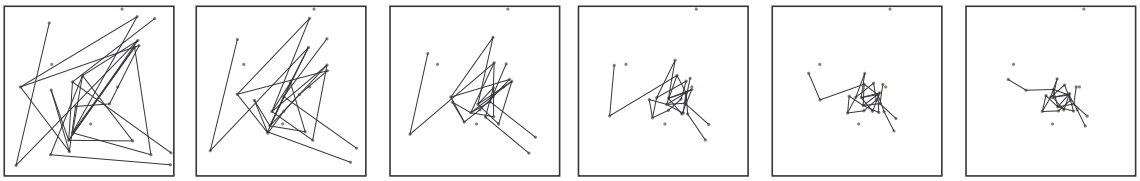

Figure B.1: A sample run of the synthetic benchmark with $m = 50$ and $n = 25$. Lines are used to denote pairs of points with a control placed between them. Each step, the controls push the points closer to one another.

and small instruction and data caches. Current generation personal computers provide significantly better performance than this older machine. Where figures are provided for this machine, they will be explicitly noted.

B.1 A Synthetic Benchmark

In order to evaluate the performance and scalability of the solver, a contrived problem was developed that can be arbitrarily scaled. The problem is designed to mimic realistic problems in which the constraints may have random structure. An instance of the problem can be defined for any number of variables m and constraints n. The problem places $m/2$ points on the plane and places n controls, each connecting 2 points. A control computes the distance between the points and has a `GoTowards` controller placed on it.

A synthetic workload generator creates instances of the problem. It randomly distributes the points in a 10x10 square. Pairs of points are selected to have the distance controls placed on them. The workload generator insures that there are no duplicate controls, but makes no other checks on the distribution of the constraints. An example run is shown in Figure B.1.

Generating arbitrarily sized synthetic workloads for the solver that are reliable measures of performance is difficult. The speed of the solver depends almost as much on the structure of the constraint problem as it does on its size. This can create a bias based on the density of constraints in variables. For example, consider an example with 2 constraints. If there are only 3 variables, no matter how the 2 constraints are placed, the constraints will be in a single partition. However, if there are 4 variables, the constraints might not access any common variables, so the solver will partition the matrix. As the number of variables goes up, the probability of this goes up as well. This can lead to the counterintuitive result that for a fixed number of constraints, larger numbers of variables might actually be faster to solve. For randomly generated problems of the same size, there can be substantial differences in solving time based on how "hard" the system is to solve: if one set of constraints is more tightly connected than another.

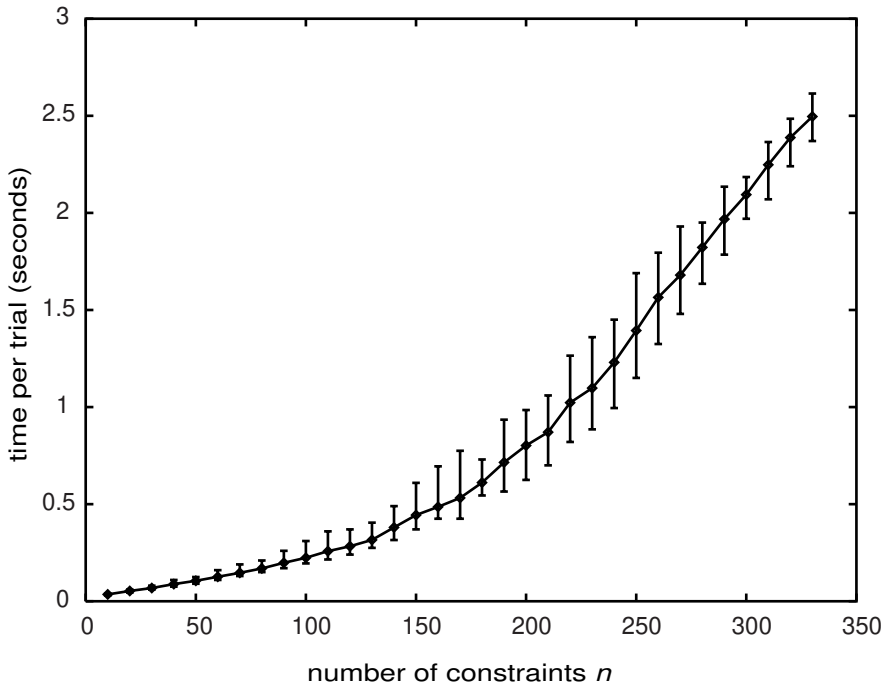

Figure B.2: Results of the synthetic benchmark. The ordinate enumerates the number of controls, the abscissa is the time required to run 5 4th order Runge-Kutta steps. Error bars represent the range of time for each value. The central ticks mark the mean values for the trials. $m = 400$ variables were used in all trials.

To run the trials, 5 constraint configurations and 5 initial positions were generated for each problem size. This leads to 25 different trials per problem size. Each trial was run for 5 Runge-Kutta 4 steps (that is, the differential optimization was solved 20 times). Each trial was duplicated a small number of times.

Graphing the results of running the trials for a fixed number of variables and varying number of constraints yields an expected result, as shown in Figure B.2. The error bars represent the range of time for each problem, while the ticks and line graph show the mean value. The graph shows the expected quadratic performance, but also shows a wide variance of times for a given problem size. Some of this variance can be attributed to how different randomly generated constraint sets can be partitioned.

A similar experiment fixed the number of constraints, but varied the number of variables. The results shown in Figure B.3 are inconclusive: the effects of problem "hardness" are more significant than the number of variables. This is sensible because there are very few parts of the algorithm that are $O(m)$, and all of these have very small constants. While the $O(n^2)$ is able to dominate the problem hardness, the $O(m)$ terms are not.

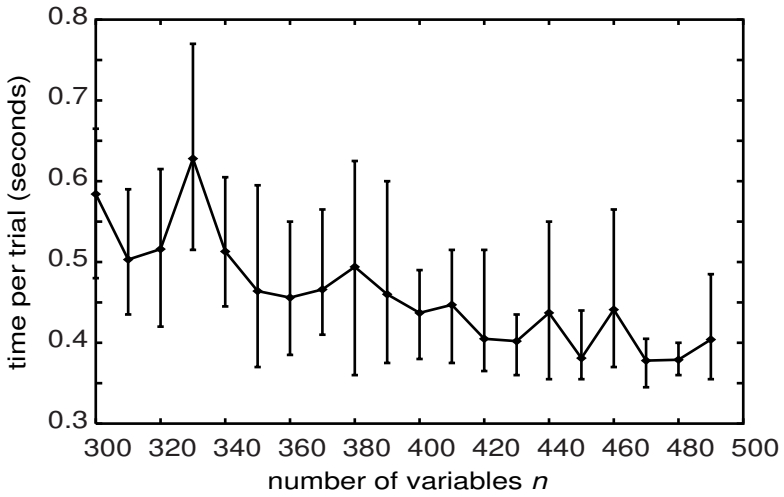

Figure B.3: Results of the synthetic benchmark for 150 constraints and a varying number of variables.

In absolute terms, if we require 5 Runge-Kutta steps per second, an Indigo 2 can handle approximately 200 point distance constraints on 400 variables. This, of course, leaves no time for redraw. More realistically, if we wanted to leave half of the time for drawing and other system functions, the benchmarks could handle approximately 150 constraints. The absolute performance numbers are not really what is important here as they depend heavily on the implementation, the machine, and the problem.

To understand the performance of the benchmarks, a number of trials were run with a version of the benchmark driver compiled with the `pixie` profiling tool available on the Iris. Pixie provides detailed information about where a program spends it time by instrumenting the code. Pixie's output is not hierarchical, so only low level procedures can be accurately monitored.

Table B.1 shows the results of running a number of trials through pixie. The table displays the time in percent that the program spent in the most used basic blocks. The top two lines of the table are important: a very large part of the program's running time is spent in two lines of code. These two lines of code are the half-sparse matrix times vector and half-sparse matrix transpose time vector inner loops. This is not surprising because these form the inner loops of the $O(n^2)$ part of the algorithm, so as the number of constraints grow, the percentage of time in these loops also grows. The `linComb` function computes the linear combination of vectors, and is used to compute the gradients of function blocks. The `cgradI` procedure actually executes the conjugate-gradient solver, but its time does not include time spent in the procedures it calls, including the matrix vector multiply functions, and `dot`, a function that computes the dot product of two vectors. All calls to `dot` in this program occur inside of

procedure	% time, $n = 150$	% time, $n = 200$	% time $n = 250$
`HSpMat::multT`	27.01	28.66	29.96
`HSpMat::mult`	18.62	21.21	23.04
`linComb`	8.39	6.49	4.84
`cgradI`	5.08	6.49	6.58
`dot`	4.16	4.76	5.22

Table B.1: Profiling results for the synthetic constraint benchmark. Numbers represent the percent of total running time spent in the basic procedure blocks. A few of the most time consuming parts of the program are shown.

the conjugate-gradient solver.

B.2 Application Benchmarks

To provide a more realistic evaluation of the absolute performance of the prototypes running on the Iris, the performance of various Bramble applications was measured on a number of examples. In each case, the example object was created beforehand. The timings are measured only while the solver is running with all of the controls, for example during dragging or while a mechanism is being driven by motors.

B.2.1 Direct Manipulation Interaction Techniques

Traditional direct manipulation interaction techniques involve a small number of controls, usually the same number as the degrees of freedom of the input device. When implemented in the differential approach, these direct manipulation techniques usually require some slightly larger number of controls, but still a small constant.

Table B.2 describes the performance of Bramble while executing some of the direct manipulation methods discussed this thesis. All of the techniques require short enough periods of time such that solving will not be the bottleneck in interactive performance. Redraw, which must be done in a conventional direct manipulation implementation as well, is more likely to limit the frame rate. Statistics are also provided for the Personal Iris.

B.2.2 Constrained Models

One use of the differential approach is to permit the user to specify an arbitrary number of controls, in order to provide a constraint-based interface. Here we discuss the performance for constraint benchmarks using Bramble applications with constraint-based interfaces. The important concern here is how large a model can the user create

		Indigo 2 times per		Personal Iris times per	
	controls	RK4 step	redraw	RK4 step	redraw
dragging a spiral Section 8.1.1	2	.001	.01	.01	.05
3D Jack Widget Section 8.3.6	2	.003	.02	.03	.11
Image Alignment Section 8.2.4	8	.009	.02	.06	.07

Table B.2: Performance of various direct manipulation techniques on two different computers. Time is in seconds per 4th order Runge-Kutta step, and for complete redraw of the view window.

before performance becomes unacceptable. The frame rate at which direct manipulation becomes unacceptable seems to vary by application, task, and user. However, the experiments here show that the prototype implementations can permit the direct manipulation of models with dozens of interactive controls, and this number can be raised substantially using the methods of Section 4.4.

A set of benchmarks was run with the MechToy application. For the first set of tests, a number of 5 bar linkages were animated by enabling their motors. An example is shown in Figure B.4. Because the mechanisms are all independent, we would expect that the performance would be linear. Even though the MechToy program does not use partitioning, the conjugate-gradient solver does partitioning automatically for this problem. The expected linear behavior is evidenced in the performance of the system, plotted in Figure B.5. For the case of 9 mechanisms (the most that fit easily on the screen), the Runge-Kutta 4 steps averaged 56 milliseconds, each call to the conjugate-gradient solver averaged 8 milliseconds, and each Jacobian construction averaged 3 milliseconds. Redrawing averaged 46 milliseconds.

The next mechanism example is more tightly coupled: all the pieces are interconnected to form a single 4 bar linkage. As the motor rotates, each "truss" rocks back and forth. No matter how big the mechanism, it only has a single degree of freedom. As more parallel trusses are added, the number of variables and constraints grow. A picture of the mechanism with 5 trusses is shown in Figure B.6. Performance figures are given in Table B.3, and graphed in Figure B.7 This example shows that even with completely connected constraints, models with around 100 constraints are practical on a machine such as the Indigo 2.

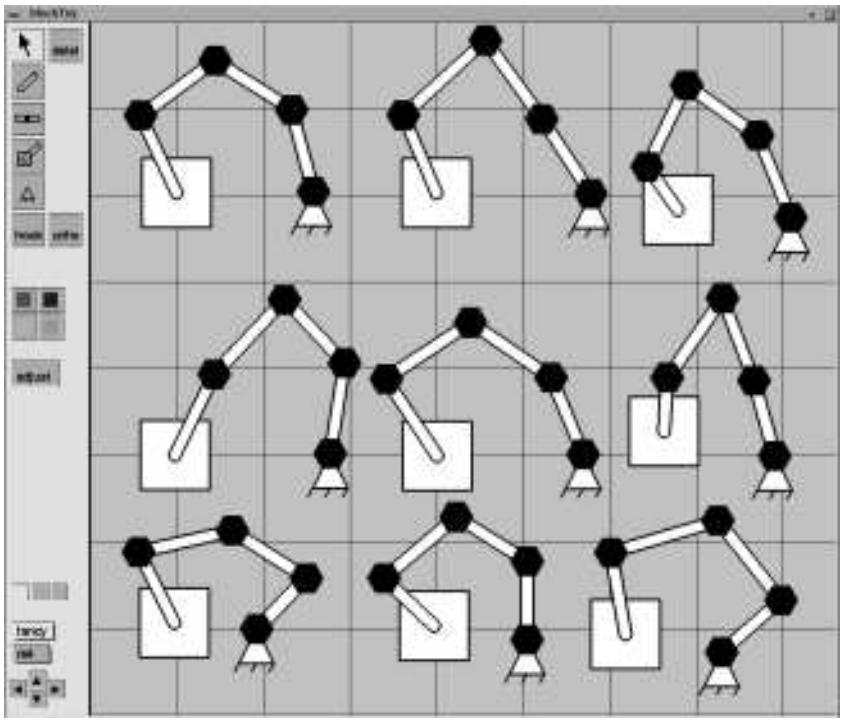

Figure B.4: MechToy animating 9 5-bar linkage mechanisms. Each mechanism is independent of the others, although mechtoy simulates them simultaneously.

.

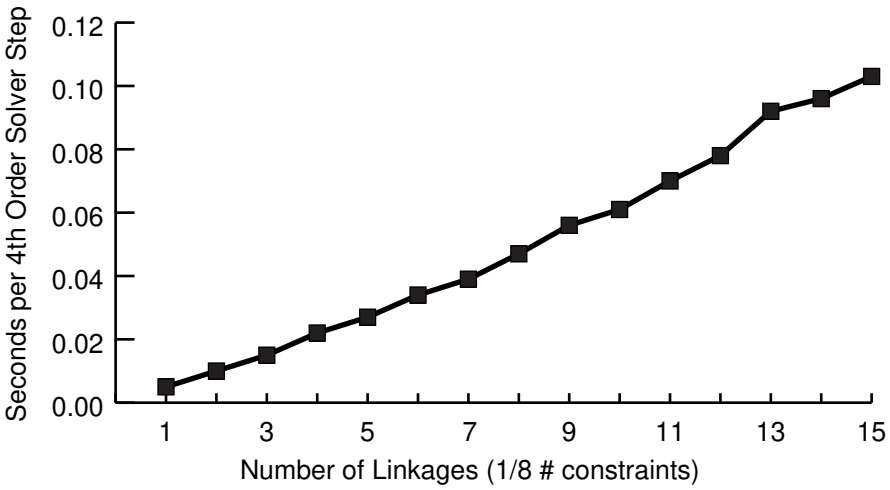

Figure B.5: Performance of MechToy simulating a number of 5 bar linkages simultaneously. The number of constraints is 8 times the number of linkages.

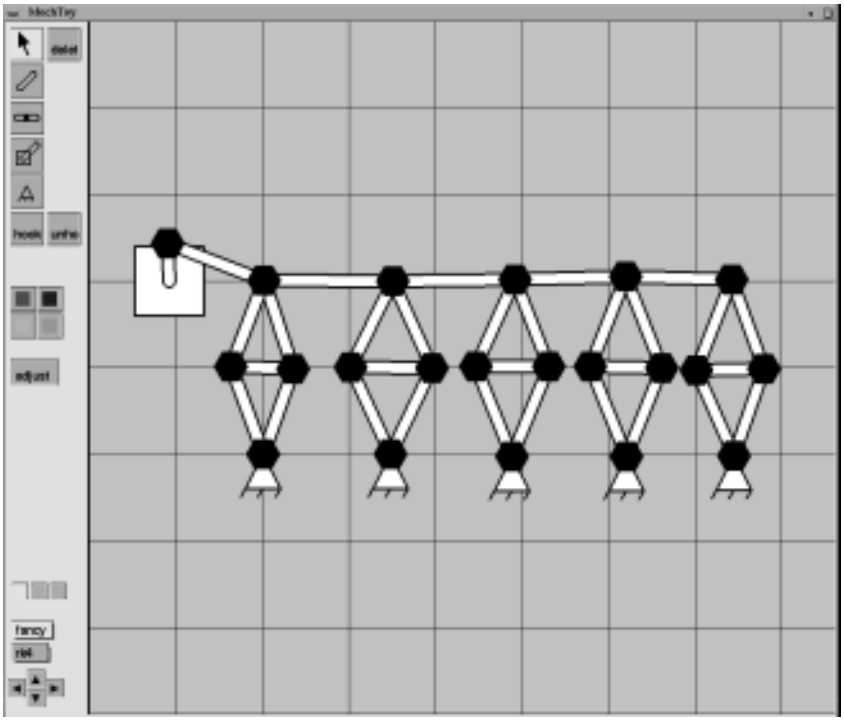

Figure B.6: A 4-bar linkage with 5 parallel trusses.

number of			seconds per				frames/sec	
trusses	vars	consts	step	cgrad	jac	draw	1 step	2 steps
1	19	18	.013	.002	.001	.015	35.4	25.1
2	37	36	.029	.005	.002	.022	19.8	12.9
3	55	54	.048	.008	.002	.029	12.9	8
4	73	72	.068	.012	.003	.038	9.6	5.8
5	91	90	.093	.017	.004	.044	7.3	4.3
6	109	108	.121	.023	.005	.051	5.5	3.4
7	127	126	.159	.031	.005	.062	4.7	2.6

Table B.3: Performance figures for MechToy simulating a mechanism with varying numbers of parallel trusses. Columns denote the time for an average 4th order Runge-Kutta step, solving the linear system with conjugate-gradient, forming the Jacobian, and redrawing the entire view. 4 calls to `cgrad` and Jacobian formation are required for each step. The rightmost columns show the frame rates using 1 and 2 solver steps per redraw.

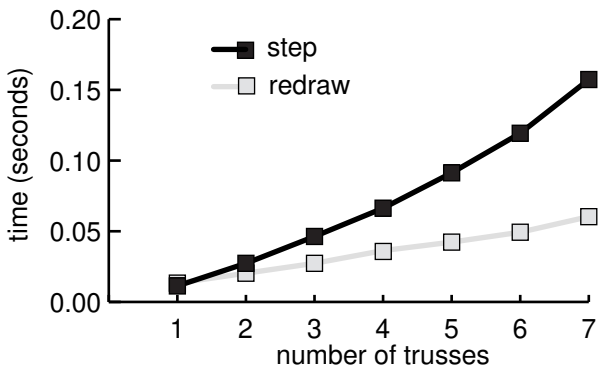

Figure B.7: Performance running the simulation of the truss mechanism. $O(n^2)$ solving time quickly grows to dominate the $O(m)$ drawing time.

References

[AGL87] William Armstrong, Mark Green, and Robert Lake. Near-real time control of human figure models. *IEEE Computer Graphics and Applications*, pages 52–61, June 1987.

[Ald88] B. Aldefeld. Variation of geometries based on a geometric-reasoning method. *Computer Aided Design*, 20(3):117–126, April 1988.

[Ald92] Aldus Corporation. Intellidraw. Computer Program, 1992.

[Alp93] Sherman R. Alpert. Graceful interaction with graphical constraints. *IEEE Computer Graphics and Applications*, pages 82–91, March 1993.

[AS85] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[Bar86] Paul S. Barth. An object-oriented approach to graphical interfaces. *ACM Transactions on Graphics*, 5(2):142–172, April 1986.

[Bar89] David Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *Computer Graphics* (*Proc. SIGGRAPH*), volume 23, pages 223–232. ACM, July 1989.

[Bar90] David Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. In *Computer Graphics* (*Proc. SIGGRAPH*), volume 24, pages 19–28. ACM, August 1990.

[Bar91a] David Baraff. Coping with friction for non-penetrating rigid body simulation. In *Computer Graphics* (*Proc. SIGGRAPH*), volume 25, pages 31–40. ACM, July 1991.

[Bar91b] Joel Bartlett. Don't fidget with widgets, draw! Technical report, DEC Western Research Laboratory, May 1991.

[Bar92a] David Baraff. *Dynamic Simulation of Non-Penetrating Rigid Bodies*. PhD thesis, Department of Computer Science, Cornell University, March 1992. Appears as technical report 92-1275.

[Bar92b] David Baraff. *Dynamic Simulation of Non-Penetrating Rigid Bodies*. PhD thesis, Cornell University, March 1992. Appears as Department of Computer Science Technical Report 92-1275.

[Bar92c] Ronen Barzel. *Physically-Based Modeling for Computer Graphics*. Academic Press, 1992.

[Bar94] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 23–34, July 1994.

[Bau72] J. Baumgarte. Stabilization of constraints and integrals of motion in dynamical systems. *Computer Methods in Applied Mechanics and Engineering*, 1:1–16, 1972.

[BB88] Ronen Barzel and Alan H. Barr. A modeling system based on dynamic constaints. *Computer Graphics*, 22:179–188, 1988. Proceedings SIGGRAPH '88.

[BD86] Alan Borning and Robert Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, 5(4):345–374, October 1986.

[BDFB+87] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. In *Proceedings OOPSLA*, pages 48–60, October 1987.

[Ben89] M. Benyon. Evaluating definative principles for interaction in graphics. In R. A. Earnshaw and B. Wyvill, editors, *New Advances in Computer Graphics: Proceedings of CG International '89*. Springer Verlag, 1989.

[BFBW92] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Functional Programming*, 5:223–270, 1992.

[BG88] Nathaniel Borenstein and James Gosling. UNIX Emacs: A retrospective. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 95–101, 1988.

[BGK93] Christian Bischoff, Andreas Griewank, and Peyvand Khademi. Workshop report on first theory institute on computational differentiation. Technical Report ANL/MCS-TM-183, Argonne National Laboratory, December 1993. Abstracts from the workshop held at Argonne May 24-26,1993.

[Bie86] Eric Bier. Skitters and jacks: Interactive 3d positioning tools. In *Proceedings of the 1986 Workshop on Interactive 3d Graphics*, pages 237–249, October 1986.

[Bie89] Eric Bier. Snap-dragging: Interactive geometric design in two and three dimensions. Technical Report EDL-89-2, Xerox Palo Alto Research Center, 1989.

[Bie90] Eric Bier. Snap-dragging in three dimensions. *Computer Graphics*, 24(2):193–204, March 1990. Proceedings 1990 Symposium on Interactive 3D Graphics.

[Bli88a] James Blinn. Me and my (fake) shadow. *IEEE Computer Graphics and Applications*, pages 82–86, January 1988.

[Bli88b] Jim Blinn. Where am I? What am I looking at? *IEEE Computer Graphics and Applications*, pages 76–81, July 1988.

[BMB86] Norman Badler, Kamran Manoocherhri, and David Baraff. Multi-dimentional input techniques and articulated figure positioning by multiple constraints. In *Proceedings of the 1986 Workshop on Interactive 3d Graphics*, pages 151–170, October 1986.

[BMW87] Norman Badler, Kamran Manoocherhri, and Graham Walters. Articulated figure positioning by multiple constraints. *IEEE Computer Graphics and Applications*, pages 28–38, June 1987.

[Bor81] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, 1981.

[Bor86] Alan Borning. Defining constraints graphically. In *Proceedings CHI 86*, pages 137–143, April 1986.

[Bro86] Frederick Brooks. Walkthough – a dynamic graphics environment for simulating virtual buildings. In *Proceedings of the 1986 Workshop on Interactive 3d Graphics*, pages 9–22, October 1986.

[Bru86] Beat Brudelin. Constructing three-dimensional geometric objects defined by constraints. In *Proceedings of the 1986 Workshop on Interactive 3d Graphics*, pages 111–129, October 1986.

[BS86] Eric Bier and Maureen Stone. Snap-dragging. *Computer Graphics*, 20(4):233–240, 1986. Proceedings SIGGRAPH '86.

[BW92] David Baraff and Andrew Witkin. Dynamic simulation of non-penetrating flexible bodies. *Computer Graphics*, 26(2):303–308, July 1992. Proceedings Siggraph '92.

[CFV88] U. Cugini, F. Folini, and I Vicini. A procedural system for the definition and storage of technical drawings in parametric form. In D. A. Duce and P. Jancene, editors, *Eurographics '88*, pages 183–196. Elsevier Science Publishers, 1988.

[CG91] George Celniker and David Gossard. Deformable curve and surface finite-elements for free-form shape design. In *Computer Graphics (Proceedings SIGGRAPH 91)*, pages 257–266, 1991.

[CMS88] Michael Chen, S. Joy Mountford, and Abigail Sellen. A study in interactive 3D rotation using 2D input devices. *Computer Graphics*, 22(4):121–130, August 1988. Proceedings SIGGRAPH '88.

[Com88] PHIGS+ Committee. Phigs+ functional description, revision 3.0. *Computer Graphics*, 22(3):125–215, 1988.

[Com92] Computervision Corporation. DesignView. Computer Program, 1992.

[Cor89] Thomas H. Cormen. *Introduction to Algorithms*. MIT Press, 1989.

[Cra86] John Craig. *Robotics: Mechanics and Control*. Addison-Wesley, 1986.

[CSH+92] D. Brookshire Conner, Scott Snibbe, Kenneth Herndon, Daniel Robbins, Robert Zeleznik, and Andries van Dam. Three-dimensional widgets. In *Proceedings of the 1992 Workshop on Interactive 3D Graphics*, pages 183–188, March 1992.

[CW92] George Celniker and William Welch. Linear constraints for deformable b-spline surfaces. In *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, pages 165–170, March 1992.

[DER86] J. S. Duff, A. M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Oxford, UK, 1986.

[DGZ92] Steven Drucker, Tinsley Gaylean, and David Zeltzer. CINEMA: a system for procedural camera movements. In *Proceedings of the 1992 Symposium on Interactive Computer Graphics*, pages 67–70, 1992.

[DNN+93] Roger B. Dannenberg, Tom Neuendorffer, Joseph M. Newcomer, Dean Rubine, and David B. Anderson. Tactus: toolkit-level support for synchronized interactive multimedia. *Multimedia Systems*, 1:77–86, 1993.

[End90] Eric Enderton. Interactive type synthesis of mechanisms. Master's thesis, University of California, Berkeley, April 1990. Also appears as Report No. UCB/CSD 90/570.

[Eng86] Douglas Englebart. The augmented knowledge workshop. In *ACM Conference on the History of Personal Workstations*, pages 73–83, January 1986.

[ETW81] Kenneth B. Evans, Peter P. Tanner, and Marceli Wein. Tablet-based valuators that provide one, two, or three degrees of freedom. *Computer Graphics*, 15(3):91–97, August 1981.

[FB93] Bjorn Freeman-Benson. Converting an existing user interface to use constraints. In Randy Pausch, editor, *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 207–215, 1993.

[FBB92] Bjorn Freeman-Benson and Alan Borning. Integrating constraints with an object oriented language. In *Proceedings ECOOP '92*, 1992.

[FBMB90] Bjorn Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint hierarchy solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[Fei93] Steven Feiner. Knowledge-based design of 3D graphics and virtual worlds. In *Proceedings Graphics Interface*, pages 51–52, 1993.

[FF87] Daniel Friedman and Matthias Felleisen. *The Little Lisper*. MIT Press, 1987.

[FFD93] Mingxian Fa, Terrence Fernando, and Peter Dew. Direct 3D manipulation techniques for interactive solid modelling. In *Proceedings Eurographics*, pages 237–248, 1993.

[Fle87] Roger Fletcher. *Practical Methods of Optimization*. John Wiley and Sons, 1987.

[Fow92] Barry Fowler. Geometric manipulation of tensor-product surfaces. In *Proceedings, Interactive 3D Workshop*, 1992.

[FP88] N. Fuller and D. Prusinkiewicz. Geometric modelling with euclidean constructions. In M. Magnenant-Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics: Proceedings of CG International '88*. Springer-Verlag, 1988.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, second edition edition, 1990.

[FW88] Kurt Fleischer and Andrew Witkin. A modeling testbed. In *Proc .Graphics Interface*, pages 127–137, 1988.

[Gan84] Sundaram Ganapathy. Decomposition of transformation matrices for robot vision. In *International Conference on Robotics*, pages 130–139, March 1984.

[Gas93] Marie-Paule Gascuel. An implicit formulation for precise contact modeling between flexible solids. *Computer Graphics*, 27:313–320, August 1993. Proceedings Siggraph '93.

[GBT93] Enrico Gobberti, Jean-Francis Balaguer, and Daniel Thalmann. VB2: An architecture for interaction in synthetic worlds. In Randy Pausch, editor, *Proceedings UIST '93*, pages 167–178, 1993.

[Gen79] Donald Gennery. Stereo-camera calibration. In *Proc. DARPA Image Understanding Workshop*, pages 101–107, 1979.

[GL89] Gene Golub and Charles Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1989.

[Gle92a] Michael Gleicher. Briar - a constraint-based drawing program. In *SIGGRAPH Video Review*, volume 77, 1992. CHI '92 Formal Video Program.

[Gle92b] Michael Gleicher. Through-the-lens camera control. In *SIGGRAPH video review 86*, 1992.

[GMW81] Phillip Gill, Walter Murray, and Margret Wright. *Practical Optimization*. Academic Press, New York, NY, 1981.

[Gol80] Herbert Goldstein. *Classical Mechanics*. Addison Wesley, 1980.

[Gos83] James Gosling. *Algebraic Constraints*. PhD thesis, Carnegie Mellon University, May 1983.

[Gri89] Andreas Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic, 1989.

[Gro89] Mark Gross. Relational modeling: A basis for computer-assisted design. In Maclcolm McCullough, William J. Mitchell, and Patrick Purcell, editors, *The Electronic Design Studio (Proc. CAAD Futrues '89)*, pages 123–146. MIT Press, 1989.

[GW91a] Michael Gleicher and Andrew Witkin. Differential manipulation. *Graphics Interface*, pages 61–67, June 1991.

[GW91b] Michael Gleicher and Andrew Witkin. Snap together mathematics. In Edwin Blake and Peter Weisskirchen, editors, *Advances in Object Oriented Graphics 1: Proceedings of the 1990 Eurographics Workshop on Object Oriented Graphics*. Springer Verlag, 1991. Also appears as CMU School of Computer Science Technical Report CMU-CS-90-164.

[GW92] Michael Gleicher and Andrew Witkin. Through-the-lens camera control. *Computer Graphics*, 26(2):331–340, July 1992. Proceedings Siggraph '92.

[GW93] Michael Gleicher and Andrew Witkin. Supporting numerical computations in interactive contexts. In Tom Calvert, editor, *Graphics Interface*, pages 138–145, May 1993.

[GW94] Michael Gleicher and Andrew Witkin. Drawing with constraints. *The Visual Computer*, 11(1), November 1994. to appear.

[Hah88] James Hahn. Realistic animation of rigid bodies. *Computer Graphics*, 22:299–308, 1988. Proceedings SIGGRAPH '88.

[Hal89] Roy Hall. *Illumination and Color in Computer Generated Imagaery*. Springer-Verlag, New York, 1989.

[HBP+93] Ralph Hill, Tom Brinck, John Patterson, Steven Rohall, and Wayne Wilner. The Rendezvous language and architecture. *Comminications of the ACM*, 36(1):62–67, January 1993.

[Hei93] Jeff Heisserman. Boeing computer services. Personal Communication, 1993.

[HH88] Tyson Henry and Scott Hudson. Using active data in a UIMS. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 167–178, 1988.

[HH90] Pat Hanrahan and Paul Haeberli. Direct WYSIWYG painting and texturing on 3D shapes. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 215–223, August 1990.

[HHK92] Willaim M. Hsu, John F. Hughes, and Henry Kaufman. Direct manipulation of free-form deformations. *Computer Graphics*, 26(2):177–182, July 1992. Proceedings Siggraph '92.

[HHN90] Tyson R. Henry, Scott E. Hudson, and Gary L. Newell. Integrating gesture and snapping into a user interface toolkit. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 112–121, October 1990.

[Hil91] Ralph D. Hill. A 2-d graphics system for multi-user interactive graphics based on objects and constraints. In E. Blake and P. Weisskirchen, editors, *Advances in Object Oriented Graphics 1: Proceedings of the 1990 Eurographics Workshop on Object Oriented Graphics*, pages 67–92. Springer Verlag, 1991.

[Hor91] Bruce Horn. Siri: a symbolic reduction interpreter for object oriented constraint programming. Technical Report CMU-CS-91-152, CMU School of Computer Science, June 1991.

[Hor92] Bruce Horn. Constraint patterns as a basis for object oriented programming. In *Proceedings OOPSLA '92*, pages 218–233, October 1992.

[Hor93] Bruce Horn. *Constrained Objects*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1993. Appears as CMU SCS technical report CMU-CS-93-154.

[Hou92] Stephanie Houde. Iterative design of an interface for easy 3D direct manipulation. In *Proceedings CHI '92*, pages 135–142, May 1992.

[HtM93] Chris Hanson and the MIT Scheme Team. Mit Scheme. online reference manual, October 1993.

[Hud90] Scott E. Hudson. Adaptive semantic snapping – a technique for semantic feedback at the lexical level. In *Proceedings CHI '90*, pages 65–70, April 1990.

[Hud91] Scott Hudson. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM Transactions on Programming Languages and Systems*, 13(3):315–341, 1991.

[Hud92] Scott Hudson. Adding shadows to a 3D cursor. *ACM Transactions on Graphics*, 11(2):193–199, April 1992.

[HY91] Scott E. Hudson and Andrey K. Yeatts. Smoothly integrating rule-based techniques into a direct manipulation user interface builder. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 145–153, November 1991.

[HZR+92] Kenneth Herndon, Robert Zeleznik, Daniel Robbins, D. Brookshire Conner, Scott Snibbe, and Andries van Dam. Interactive shadows. In *Proceedings of the 1992 ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 1–6, November 1992.

[IC87] Paul Issacs and Michael Cohen. Controlling dynamics simulation with kinematic constraints, behavior functions and inverse dynamics. *Computer Graphics*, 21(4):215–224, 1987. Proceedings SIGGRAPH '87.

[Iri91] Masao Iri. History of automatic differentiation and rounding error estimation. In Andreas Griewank and George Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, pages 3–16. SIAM, January 1991.

[Joh63] Timothy E. Johnson. Sketchpad III: A computer program for drawing in three dimensions. In *Conference Proceedings, Spring Joint Computer Conference*. IEEE Computer Society, 1963. Reprinted in Herbert Freeman, ed, Tutorial and Selected Readings in Interactive Computer Graphics, 1980, pp20–26.

[Jue91] David W. Juedes. A taxonomy of automatic differentiation tools. In Andreas Griewank and George Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, pages 315–329. SIAM, January 1991.

[Kas92] Michael Kass. CONDOR: constraint-based data flow. *Computer Graphics*, 26:321–330, July 1992. Proceedings SIGGRAPH '92.

[Kau91] Henry Kaufman. Constraint techniques for interactive physically-based modeling. Master's thesis, Brown Unversity, July 1991.

[KF93] David Kurlander and Steven Feiner. Inferring constraints from multiple snapshots. *ACM Transactions on Computer Graphics*, 12(4), October 1993.

[KLW92] Solange Karsenty, James A. Landay, and Chris Weikart. Inferring graphical constraints with Rockit. In *HCI'92 Conference on People and Computers VII*, pages 137–153. British Computer Society, September 1992.

[KNK89] Nami Kim, Tsukasa Noma, and Tosiyasu L. Kunii. PictureEditor: A 2D picture editing system based on geometric constructions and constraints. In R. A. Earnshaw and B. Wyvill, editors, *New Advances in Computer Graphics: Proceedings of CG International '89*, pages 193–207. Springer Verlag, 1989.

[Kol91] Craig Kolb. Rayshade. Computer Program, 1991.

[KP88] Glenn Krasner and Stephen Pope. A cookbook for using the Model–View–Controller user interface paradigm in smalltalk-80. *The Journal of Object Oriented Programming*, pages 26–49, August/September 1988.

[KPC93] John Kawai, James Painter, and Michael Cohen. Radioptimization – goal based rendering. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, pages 147–154, 1993.

[Kra90] Glenn A. Kramer. Solving geometric constraint systems. In *Proceedings AAAI-90*, pages 708–714, 1990.

[Kur93] David Kurlander. *Graphical Editing by Example*. PhD thesis, Columbia University, 1993.

[KW93] Larry Koved and Wayne Wooten. GROOP: an object-oriented toolkit for animated 3D graphics. In Andreas Paepcke, editor, *OOPSLA '93 Conference Proceedings*, pages 309–325, October 1993.

[KWT88] Michael Kass, Andrew Witkin, and Demetri Terzopoulis. Snakes: Active contour models. *International Journal of Computer Vision*, 321–331, 1988.

[Lel88] Wm. Leler. *Constraint Programming Languages: Their Specification and Generation*. Addison-Weseley, 1988.

[LGL81] V. C. Lin, D. C. Gossard, and R. A. Light. Variational geometry in C.A.D. *Computer Graphics*, 15(3):171–177, 1981. Proceedings SIGGRAPH '81.

[Low80] David Lowe. Solving for the parameters of object models from image descriptions. In *Proc. DARPA Image Understanding Workshop*, pages 121–127, 1980.

[Mac90] Anthony Maciejewski. Dealing with the ill-conditioned equations of motion for articulated figures. *IEEE Computer Graphics and Applications*, May 1990.

[Mal91] John Harold Maloney. *Using Constraints for User Interface Construction*. PhD thesis, University of Washington, 1991. Appears as Computer Science Technical Report 91-08-12.

[MB86] Brad A. Myers and William Buxton. Creating highly-interactive and graphical user interfaces by demonstration. *Computer Graphics*, 20(4):249–258, 1986. Proceedings SIGGRAPH '86.

[McG89] Chris McGlone. Automated image-map registration using active contour models and photogrammetric techniques. In *Proceedings of the SPIE, Volume 1070*, January 1989.

[MCR90] Jock Mackinlay, Stuart Card, and George Robertson. Rapid controlled movement through a virtual 3d workspace. *Computer Graphics*, 24(4):171–176, August 1990.

[Mer50] Mildred P. Merryman. *Children's Stories*, chapter "Quack!" said Jerusha, pages 117–136. Whitman Publishing Co., Racine, WI, 1950.

[MGD+90] Brad A. Myers, Dario Guise, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Ed Pervin, Andrew Mickish, and Phillipe Marchal. Comprehensive support for graphical, highly-interactive user interfaces: The Garnet user interface development environment. *IEEE Computer*, November 1990.

[MKW89] D. L. Maulsby, K. A. Kittlinz, and I. H. Witten. Metamouse: Specifying graphical procedures by example. *Computer Graphics*, 23(3):127–136, July 1989. Proceedings SIGGRAPH '89.

[Mof59] Francis H. Moffitt. *Photogrammetry*. International Textbook Company, 1959.

[MW88] P.M. Moore and J. Wilhelms. Collision detection and reponse for computer animation. In *Computer Graphics* (*Proc. SIGGRAPH*), volume 22, pages 289–298. ACM, August 1988.

[Mye90] Brad Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(3):289–320, July 1990.

[Mye93] Brad Myers. State of the art in user interface software tools. In H. Rex Hartson and Deborah Hix, editors, *Advances in Human-Computer Interaction*, volume 4, pages 110–150. Ablex Publishing, 1993. Appears as CMU School of Computer Science technical report CMU-CS-92-114.

[Mye94] Brad Myers. Challenges of HCI design and implementation. *Interactions*, pages 73–83, January 1994.

[Nak91] Yoshiko Nakamura. *Advanced Robotics: Redundancy and Optimization*. Addison-Wesley, 1991.

[Nel85] Greg Nelson. Juno, a constraint based graphics system. *Computer Graphics*, 19(3):235–243, 1985. Proceedings SIGGRAPH '85.

[NKK+88] T. Noma, T. L. Kunii, N. Kin, H. Enomoto, E. Aso, and T. Yamamoto. Drawing input through geometrical constructions: Specification and applications. In M. Magnenant-Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics: Proceedings of CG International '88*. Springer-Verlag, 1988.

[NO86] Gregory M. Nielson and Dan R. Olsen. Direct manipulation techniques for 3d objects using 2d locator devices. In *Proceedings of the 1986 Workshop on Interactive 3d Graphics*, pages 175–182, 1986.

[Nor90] Donald Norman. *The Design of Everyday Things*. Doubleday, 1990.

[OA90] Dan R. Olsen and Kirk Allan. Creating interactive techniques by symbolically solving geometric constraints. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 102–107, 1990.

[OA92] James R. Osborn and Alice M. Agogino. An interface for interactive spatial reasoning and visualization. In *Proceedings CHI '92*, pages 75–82, May 1992.

[Ous91] John K. Ousterhout. An X11 toolkit based on the Tcl language. In *1991 Winter Usenix Conference Proceedings*, 1991.

[Pau81] Richard Paul. *Robot Manipulators: Mathematics, Programming, and Control*. MIT Press, Cambridge, MA, 1981.

[PB88a] Cary Phillips and Norman Badler. Jack: A toolkit for manipulating articulated figures. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 221–229, 1988.

[PB88b] John Platt and Alan Barr. Constraint methods for flexible models. *Computer Graphics*, 22:279–288, 1988. Proceedings SIGGRAPH '88.

[PB91] Cary Phillips and Norman Badler. Interactive behaviors for bipedal articulated figures. In *Computer Graphics (Proceedings SIGGRAPH 91)*, pages 359–362, 1991.

[PF92] Pierre Poulin and Alain Fournier. Light from highlights and shadows. In *Proceedings of the 1992 Symposium on Interactive Computer Graphics*, pages 31–38, 1992.

[PFTV86] William Press, Brian Flannery, Saul Teukolsky, and William Vetterling. *Numerical Recipes in C*. Cambridge University Press, Cambridge, England, 1986.

[Pla92] John Platt. A generalization of dynamic constraints. *CGVIP: Graphical Models and Image Processing*, 54(6):516–525, November 1992.

[PS82] Christopher Paige and Michael Saunders. LSQR: an algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software*, 8(1):43–71, March 1982.

[PT94] Randy Pausch and The University of Virginia User Interface Group. Personal communication, 1994.

[Pug92] David Pugh. Designing solid objects with interactive sketch interpretation. In *Proceedings of the 1992 Symposium on Interactive Computer Graphics*, pages 117–126, March 1992.

[PW85] Theo Pavlidis and Christopher Van Wyk. An automatic beautifier for drawings and illustrations. *Computer Graphics*, 19(3):225–234, 1985. Proceedings SIGGRAPH '85.

[RK77] A. J. Rubel and R. E. Kaufman. Kinsyn III: A new human-engineered systems for interactive computer aided design of planar linkages. *Transactions of the ASME: Journal of Engineering for Industry*, pages 440–448, May 1977.

[Ros86] Jarek Rossignac. Constraints in constructive solid geometry. In *Proceedings of the 1986 Workshop on Interactive 3d Graphics*, pages 93–110, October 1986.

[Rub91] Dean Rubine. *The Automatic Recognition of Gestures*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991. Appears as CMU SCS technical report CMU-CS-91-202.

[San94] Michael Sannella. The SkyBlue constraint solver and its applications. In Pascal Van Hentenryck and Vijay Saraswat, editors, *Principles and Practice of Constraint Programming*. MIT Press, 1994. to appear.

[Sap93] Mark Sapossnek. *Virtual Prototyping: An Interactive Approach to Geometric Tolerance Design and Analysis*. PhD thesis, Carnegie Mellon University, 1993.

[SB91] Wolfgang Sohr and Beat Bruderlin. Interaction with constraints in 3D modeling. In *Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 387–396, June 1991.

[SB92] Michael Sannella and Alan Borning. Multi-Garnet: Integrating multi-way constraints with garnet. Technical Report 92-07-01, Department of Computer Science, University of Washington, 1992.

[SC92] Paul S. Strauss and Rikk Carey. An object-oriented 3d graphics toolkit. *Computer Graphics*, 26(2):341–349, July 1992. Proceedings SIGGRAPH '92.

[Sch59] K. Schwidefsky. *An Outline of Photogrammetry*. Pitman Publishing Corporation, first english edition, 1959.

[Sch83] Ben Schneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, pages 57–69, August 1983.

[SDS+93] Chris Schoeneman, Julie Dorsey, Brian Smits, James Arvo, and Donald Greenberg. Painting with light. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, pages 143–146, 1993.

[SG86] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[Sha93] Shapeware Inc. Visio. Computer Program, 1993.

[She94] Jonathan Shewchuck. An introduction to the conjugate gradient method without the agonizing pain. Technical Report CMU-CS-94-125, School of Computer Science, Carnegie Mellon University, 1994.

[Sho85] Ken Shoemake. Animating rotations with quaternion curves. *Computer Graphics*, 19(3):245–254, July 1985. Proceedings SIGGRAPH '85.

[Sho92] Ken Shoemake. ARCBALL: a user interface for specifying three-dimensional orientation using a mouse. In *Proceedings Graphics Interface '92*, pages 151–156, May 1992.

[Sil91] Silicon Graphics Inc. *Graphics Library Programming Guide*, 1991.

[Sis90] Steven Sistare. *A Graphical Editor for Three-Dimensional Constraint-Based Geometric Modelling*. PhD thesis, Harvard University, 1990.

[Sis91] Steven Sistare. Interaction techniques in constraint-based geometric modeling. In *Proceedings Graphics Interface '91*, pages 85–92, June 1991.

[SKN90] Gurminder Singh, Chun Hong Kok, and Teng Ye Ngan. Druid: A system for demonstrational rapid user interface development. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 167–177, October 1990.

[SKvW+92] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. *Computer Graphics*, 26(2):249–252, July 1992. Proceedings Siggraph '92.

[SLGS92] Chris Shaw, Jiadong Liang, Mark Green, and Yunqi Sun. The decoupled simulation model for virtual reality systems. In *Proceedings CHI '92*, pages 321–328, May 1992.

[SM88] Pedro A. Szekely and Brad A. Myers. A user interface toolkit based on graphical objects and constraints. In *OOPSLA '88 Proceedings*, pages 36–45, September 1988.

[SMFBB93] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *Software—Practice and Experience*, 23(5):529–566, May 1993.

[Sof93] Softimage Inc. Softimage creative environment. Computer Program, 1993.

[Sur92a] Mark Surles. Interactive modeling enhanced with constraints and physics – with applications in molecular modeling. In *Proceedings of the 1992 Symposium on Interactive Computer Graphics*, pages 175–182, March 1992.

[Sur92b] Mark C. Surles. An algorithm for linear complexity for interactive, physically-based modelling of large proteins. *Computer Graphics*, 26(2):221–230, 1992. Proceedings SIGGRAPH '92.

[Sur92c] Mark C. Surles. *Techniques for Interactive Manipulation of Graphical Protein Molecules*. PhD thesis, University of North Carolina at Chapel Hill, 1992. Appears as TR93-016.

[Sut63] Ivan Sutherland. *Sketchpad: A Man Machine Graphical Communication System*. PhD thesis, Massachusetts Institute of Technology, January 1963.

[SZ90] Peter Schroeder and David Zeltzer. The virtual erector set: Dynamic simulation with linear recursive constraint propagation. *Computer Graphics*, 24(2):23–31, March 1990. Proceedings 1990 Symposium on Interactive 3D Graphics.

[TBGT91] Russell Turner, Francis Balaguer, Enrico Gobbetti, and Daniel Thalmann. Physically-based interactive camera motion using 3d input devices. In N. M. Patrikalakis, editor, *Scientific Visualiztion of Physical Phenomena: Proceedings of CG International 1991*, pages 135–145, Tokyo, 1991. Springer-Verlag.

[TTA91] Konstantinos Tarabamis, Roger Tsai, and Peter Allen. Automated senor planning for robotic vision tasks. In *Proceedings of th 1991 IEEE International Conference on Robotics and Automation*, pages 76–82, April 1991.

[Ups89] Steve Upstill. *The Renderman Companion*. Addison-Wesley, 1989.

[Ven93] Dan Venolia. Facile 3D manipulation. In *Proceedings INTERCHI '93*, pages 31–36, 1993.

[VL89] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain specific graphical editors. In *Proceedings of the 1989 ACM SIGGRAPH Symposium on User Interface Software and Technology*, November 1989.

[VW82] Christopher J. Van Wyk. A high level language for specifying pictures. *ACM Transactions on Graphics*, 1(2):163–182, April 1982.

[vWJB85] J. J. van Wijk, F. W. Jansen, and W. F. Bronsvoort. Some issues in designing user interfaces to 3d raster graphics. *Computer Graphics Forum*, 4:5–10, 1985.

[VZ88] Bradley T. Vander Zanden. Constaint grammars in user interface management systems. In *Proc. Graphics Interface*, pages 176–184, 1988.

[VZ89] Bradley T. Vander Zanden. Constraint grammars – a new model for specifying graphical applications. In *Proceedings CHI '89*, pages 325–330, April 1989.

[VZMGS91] Brad Vander Zanden, Brad A. Myers, Dario Guise, and Pedro Szekeley. The importance of pointer variables in constraint models. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 155–164, November 1991.

[VZMGS94] Brad Vander Zanden, Brad Myers, Dario Guise, and Pedro Szekely. Integrating pointer variables into one-way constraint methods. *ACM Transactions on Computer Human Interaction*, 1(2), June 1994.

[Wam86] Charles W. Wampler. Manipulator inverse kinematic solutions based on vector formulations and damped least-squares method. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):93–101, January 1986.

[Wan92] Leonard Wanger. The effect of shadow quality on the perception of spatial relationships in computer generated imagery. In David Zeltzer, editor, *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, volume 25, pages 39–42, March 1992.

[Wav94] Wavefront Inc. Kinemation and dynamation. Computer Programs, 1994.

[Wel93] Chris Welman. Inverse kinematics and geometric constraints for articulated figure manipulation. Master's thesis, Simon Fraser University, September 1993.

[Wer94] Josie Wernecke. *The Inventor Mentor*. Addison-Wesley Publishing Company, 1994.

[WFB87] Andrew Witkin, Kurt Fleischer, and Alan Barr. Energy constraints on parameterized models. *Computer Graphics*, 21(4):225–232, July 1987.

[WG87] Dennis Wixon and Michael Good. Interface style and eclecticism: Moving beyond categorical approaches. In *Proceedings of the Human Factors Society – 31st Annual Meeting*, pages 571–575, 1987.

[WGW90] Andrew Witkin, Michael Gleicher, and William Welch. Interactive dynamics. *Computer Graphics*, 24(2):11–21, March 1990. Proceedings 1990 Symposium on Interactive 3D Graphics.

[WGW91] William Welch, Michael Gleicher, and Andrew Witkin. Manipulating surfaces differentially. In *Proceedings, Compugraphics '91*, September 1991.

[Whi88] R. M. White. Applying direct manipulation to geometric construction systems. In M. Magnenant-Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics: Proceedings of CG International '88*. Springer-Verlag, 1988.

[Wil87] Jane Wilhelms. Using dynamic analysis for realistic animation of articulated bodies. *IEEE Computer Graphics and Applications*, pages 12–27, June 1987.

[Wit89a] Andrew Witkin. Personal Communication, 1989.

[Wit89b] Andrew Witkin. Physcially-based modeling: Past, present future. In *SIGGRAPH Panel Proceedings*, pages 203–205, 1989. Part of panel chaired by Demetri Terzopoulis and John Platt.

[WK88] Andrew Witkin and Michael Kass. Spacetime constraints. *Computer Graphics*, 22:159–168, 1988. Proceedings SIGGRAPH '88.

[WKTF88] Andrew Witkin, Michael Kass, Demetri Terzopoulos, and Kurt Fleischer. Physically based modeling for vision and graphics. In *Proc. DARPA Image Understanding Workshop*, pages 254–278, 1988.

[WO90] Colin Ware and Steven Osborne. Exploration of virtual camera control in virtual three dimensional environments. *Computer Graphics*, 24(2):175–184, March 1990. Proceedings 1990 Symposium on Interactive 3D Graphics.

[Wol88] Stephen Wolfram. *Mathematica*. Addison Wesley, 1988.

[WR87] Catherine G. Wolf and James R. Rhyne. A taxonomic approach to understanding direct manipulation. In *Proceedings of the Human Factors Society – 31st Annual Meeting*, pages 576–580, 1987.

[WW90] Andrew Witkin and William Welch. Fast animation and control of non-rigid structures. *Computer Graphics*, 24(4):243–252, August 1990. Proceedings SIGGRAPH '90.

[WW92] William C. Welch and Andrew Witkin. Variational surface modelling. *Computer Graphics*, 26(2):157–166, July 1992. Proceedings SIGGRAPH '92.

[WW94] Will Welch and Andrew Witkin. Free form shape design using triangulated surfaces. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 247–256, July 1994.

[ZCW+91] Robert C. Zeleznik, D. Brookshire Conner, Matthias M. Wloka, Daniel G. Aliaga, Nathan T. Huang, Philip M. Hubbard, Brian Knep, Henry Kaufman, John F. Hughes, and Andries van Dam. An object-oriented framework for the integration of interactive animation techniques. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 105–112, July 1991.

[ZHR+93] Robert Zeleznik, Kenneth Herndon, Daniel Robbins, Nate Huang, Tom Meyer, Noah Parker, and John Hughes. An interactive 3d toolkit for constructing 3d widgets. *Computer Graphics*, 27:81–84, August 1993. SIGGRAPH '93 video paper.