Differential Manipulation [*]

Michael Gleicher[†]
Andrew Witkin

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
USA

Abstract

Direct manipulation has proven to be an excellent method for interacting with geometric objects. Unfortunately, traditional approaches for implementing direct manipulation suffer from a lack of generality, requiring the system designer to hand craft interfaces to different types of objects. In this paper we present *differential manipulation*, a new paradigm for direct manipulation of geometric objects. By interpreting graphical entities as physical objects, we obtain a uniform interface to a wide variety of geometric objects, making it simple to add new types of complicated or compound objects. Geometric constraints fit neatly into the paradigm.

Résumé

La manipulation directe est une excellente méthode pour le traitement interactif des objets géométriques. Malheureusement, les approches traditionelles pour l'implémentation de la manipulation directe manquent de généralité en nécessitant que différentes interfaces soient associées à différents types d'objets. Dans cet article nous présentons un nouveau paradigme, la *manipulation différentielle*, pour la manipulation directe des objets géométriques. En interprétant les entités graphiques comme des objets physiques, nous obtenons une seule interface pouvant être utilisée pour une grande variété d'objets géométriques, facilitant ainsi l'addition de nouveaux types d'objets complexes ou composés. Les contraintes géométriques peuvent être proprement incluses avec ce paradigme.

Keywords — Direct Manipulation, Interaction Techniques, Geometric Modeling

[*]This research was sponsored in part by Apple Computer and Silicon Graphics Incorporated. The first author is supported in part by a fellowship from the Schlumberger Foundation, Inc. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Schlumberger, SGI or Apple.

[†]electronic mail: gleicher@cs.cmu.edu

1 Introduction

Direct manipulation interfaces have proven to be superior in a wide variety of tasks[Shn83] and are an appealing method for manipulating shapes. Such interfaces couple the motion of a graphical object to the motion of the pointing device. For example, in a direct manipulation drawing program, such as [LV89, Cor88], we can grab a rectangle by its corner and have it move to follow the mouse or pull a spline by a control point and have it bend in the direction moved.

Unfortunately, direct manipulation can be difficult to extend because each shape has its own degrees of freedom. The coupling between these parameters and the user's control is traditionally designed by hand. The diversity of attributes creates a lack of uniformity which makes it difficult to automate the process of adding shapes to a system.

In this paper we present *differential manipulation*, a new scheme for providing direct manipulation interfaces to geometric objects. Users are permitted to pick points on objects and drag them. Objects respond by changing all of their relevant parameters, allowing control of all aspects of an object's shape through this single interface.

The interface we implement allows the user to manipulate objects by controlling points on them. However, it is not possible to specify an absolute target position for a point on an object in a general way. Instead, we treat the problem of controlling the position of a point on an object differentially: rather than specifying the target position of the point, we specify, from moment to moment, how we would like it to move. To do this requires the derivatives of the function defining the object, not the function's inverse. Derivatives are easy and inexpensive to compute, and, combined with the techniques in this paper, make it possible to provide a general method for grab-and-pull interfaces.

With differential manipulation, we can use a single, uniform interface for a wide variety of objects. Adding a new type of object is made much simpler, requiring the addition of only a few well-defined routines. For many types of objects, these routines can even be automatically generated from mathematical definitions of the shape.

Geometric constraints are handled gracefully by differential manipulation. By maintaining constraints differentially instead of solving them, nonlinear constraints are handled by solving systems of linear equations.

Differential manipulation resembles physical interactions with real-world objects. To move or shape an object we grab it and pull on it. Because we are interested in manipulation, not the quantitative prediction of real physical behavior, we are able to simplify the physical model, making objects easier to control and simulate.

2 Differential Manipulation

In this section we develop a method that allows a user to drag points on arbitrary geometric objects. Objects respond by changing their attributes to cause the dragged point to move towards the pointing device.

The configuration of an object can be described by vector of parameters $\mathbf{q}$. For example, a circle might be described by a vector of length three containing the position of its center and its radius. For each point $\mathbf{p}$ on an object, there is a function which determines the point's position as a function of the vector of the object's parameters

$$\mathbf{p} = \mathbf{x}_p(\mathbf{q}).$$

To specify the position of the point directly would require the inverse of $\mathbf{x}_p$. Not only is it difficult to automate finding inverses, but few of the functions have unique inverses.

Instead of specifying the position of a point directly, differential control allows users to control the motion of the point. This goal is achieved using simplified physical simulation. Point forces on the object are translated into generalized forces on the parameters and then integrated in time to provide changes in the object's state.

To implement differential manipulation, we consider geometric objects as physical entities and simulate their responses to user applied forces. However, since we are interested not in accurate quantitative prediction, but in manipulating objects, we can simplify the physical model. Since most of the objects which exist in our simulation have no real-world counterparts, there is no expected behavior to duplicate.

Differential manipulation implements a model of physics whose equations of motion are first order differential equations, effectively replacing $f = ma$ by $f = mv$. This first order formulation approximates situations in which frictional effects dominate inertial effects, such as moving heavy objects around on a table.[1] The lack of inertia is good for manipulation because objects remain where they are placed.

The methods of generalized coordinates [Lan86] permit the use of an arbitrary representation for objects and provide a method for computing the effects of forces on these parameters. Given a force on a point, the generalized force can be obtained by multiplying the point force by the transpose of the Jacobian of the point's position function,

$$\mathbf{g} = \left(\frac{\partial \mathbf{x}_p}{\partial \mathbf{q}}\right)^T \mathbf{f},$$

where $\mathbf{f}$ is the applied point force, g is the generalized force, and the point is related to the parameters by the function $\mathbf{p} = \mathbf{x}_p(\mathbf{q})$.

The generalized force governs the rate of change of the state. We can even connect it directly, letting

$$\dot{\mathbf{q}} = \mathbf{g}. \tag{1}$$

This differential equation can be solved to yield changes in the state. This mechanism permits users to pull on points and have the object respond.

2.1 Sensitivity Matrices

The problem with using equation (1) directly is that the resulting behavior depends on an arbitrary choice of scale factors for the parameters. Without accounting for scaling, varying sensitivities in the parameters would determine the proportion of the motion accounted for by each parameter, the choice of units determining the behavior. Consider manipulating a line segment of fixed length. Pulling on an end would cause it to respond by rotating and translating. Without accounting for sensitivities, the programmer's choice to use radians or degrees would determine the way the motion is divided.

To correct this problem, we scale the generalized force to control how the motion is apportioned among the parameters. We normalize each parameter by dividing the corresponding element of the generalized force by the sensitivity of the object to that parameter. A natural criterion for normalization is that equal changes in parameters should induce equal changes in the RMS displacement of the object. The sensitivity of an object to a parameter j can then be found by summing the distance

[1]There is evidence that first order physics is actually more intuitive to many people [Nor90].

that each of a set of point samples $\mathcal{P}$ on the object moves due to a differential change in j,

$$s_j = \sum_{p \in \mathcal{P}} m_p \frac{\partial \mathbf{x}_p}{\partial j} \cdot \frac{\partial \mathbf{x}_p}{\partial j},$$

where $\mathbf{x}_p(\mathbf{q})$ is the function which determines the position of sample p, and m_p is the weighting of sample p. The sensitivities can be written as the diagonal elements of a matrix whose inverse is multiplied by the generalized force to compute the derivative of the state. We call this matrix the sensitivity matrix $\mathbf{M}$.

Specifically, this diagonal matrix approximates the mass matrix for an object with uniformly distributed mass. In omitting the off-diagonal terms, we neglect the interdependence among parameters. In return we have a matrix that is trivial to invert. The full mass matrix is given by

$$\mathbf{M} = \sum_{p \in \mathcal{P}} m_p \mathbf{J}_p^T \mathbf{J}_p,$$

where $\mathbf{J}_p$ is the Jacobian of the relationship between sample p and the parameters. See [WW90] for further discussion.

2.2 Using Simulation for Manipulation

The user manipulates objects by applying forces to them, requiring differential manipulation programs to have force-based interaction techniques. One method we have found successful is to couple the object to the mouse with a spring of zero rest length. The point force can be computed by merely scaling the displacement between the mouse and the point being pulled. It has the advantage that the further an object is from its goal, the faster it moves. This allows for precise positioning without slowing down coarse movement. Spring connection also provides a reasonable behavior when the object is not free to move closer to the mouse.

Numerical stability and processor speed place limits on how quickly objects can move on the screen; as a result, it may happen that the user moves the mouse more quickly than an object can keep up. We find spring coupling preferable to the alternative, keeping the mouse in contact with the object with a reduced display rate. Control techniques, as in [WW90], can be used to better track the mouse, but we still must place limits on how fast objects can move.

Since forces can be summed, it is easy to pull on shapes at more than one point. This provides the opportunity to develop novel ways to sculpt objects such as allowing a user to stretch an object by holding one side and pulling another.

We are not limited to pulling points on shapes. We could control other interesting aspects of geometry, such as area, or even non-geometric entities. The techniques of differential manipulation permit us to control the output of any function of a number of variables that has a derivative. All that is required is a method for rapidly displaying the relevant output and receiving the user's desire for the direction of change in that output. We have experimented with an application that connects the outputs of mathematical functions, such as an economic model, to sliders [WGW90, GW91].

2.3 Implementing Differential Manipulation

Implementing differential manipulation involves integrating the differential equations of motion in time and periodically displaying the results to the user. These equations are most conveniently expressed in terms of the inverse of the mass matrix,

$$\dot{\mathbf{q}} = \mathbf{W}\mathbf{g}, \qquad (2)$$

where $\mathbf{W} = \mathbf{M}^{-1}$ and g is the generalized force.

Like physics, differential manipulation is a continuous process. We can only directly control the motion of an object, not its absolute position. Therefore, we can only move towards the goal, not jump to it. The continuous process, however, gives us an opportunity to place the user in the loop, controlling the object as it moves.

Equation (2) must be solved numerically[PFTV86], simulating the continuous process by taking finite steps. For each step, we compute the forces on the objects, convert them to generalized forces, compute the mass matrix, invert it, and use it to find the derivative. If we update the display rapidly enough, we achieve the illusion of continuous motion and continuous control. This tight feedback loop is important for giving users a good feel for the object's motion.

Given the function which maps the parameters to the output point, everything else necessary for controlling a specific class of objects can be computed. The Jacobian can be determined by symbolic differentiation or by numerical techniques. A mass matrix can be computed from the Jacobian. When the objects have a regular structure, such as parameterized curves, all of the methods required of an object class, such as drawing and picking, can also be coded automatically, as we will show in the next section.

2.4 Parametric Curves

An important advantage to the differential approach is its regularity, permitting new types of objects to be incorporated into a system without designing interfaces for each. In this section we describe how parametric curves are handled in a differential manipulation system to show by example how easily new shapes can be added to a geometric modeler.

Parametric curves can be defined by characteristic functions of the form

$$(x, y) = \mathbf{f}(\mathbf{q}, u),$$

for $u \in [0, 1]$. The free parameter u selects points along the curve, while the configuration vector $\mathbf{q}$ determines the curve's attributes. Different shapes such as circles, ellipses, line segments and splines, are created by different parametric functions. Each different type of curve has its own set of configuration parameters which control its attributes.

Differential manipulation provides a single interface for a wide variety of curves: users can grab and pull objects which respond by changing their configurations. The technique permits this diversity of shapes to be supported without resorting to hand-crafting interfaces for each type or limiting the way curves can vary. This is possible because different varieties of curves are mathematically the same, except for their characteristic functions.

In order to allow the user to manipulate an object, a modeling program must be able to perform a few basic operations with it. The program must be able to draw the object, permit the user to grab a point on it, apply a force to its parameters, and compute a sensitivity or mass matrix for it. Given the ability to evaluate a parametric curve's characteristic function and its derivatives, these capabilities can be provided by structure common to all types of parametric curves.

Given the function defining a curve, the curve can be drawn by sampling at various u values and connecting the dots. When the user grabs an object to drag, the place on the curve closest to the pointing device can be approximated by sampling u and selecting the closest sample or can be found more precisely by root finding.

Once the point $\mathbf{p}$ is picked and its corresponding u_p is found, the process of dragging the point towards the target position can begin. For each iteration, the current position of $\mathbf{p} = \mathbf{f}(\mathbf{q}, u_p)$ is computed and compared to the position of the target to find a point force to apply to $\mathbf{p}$, which is then translated to a generalized force on the parameters of the curve by multiplying by the transpose of the point's Jacobian,

$$\mathbf{g} = \left(\frac{\partial \mathbf{f}(\mathbf{q}, u_p)}{\partial \mathbf{q}}\right)^T \mathbf{f_a}.$$

We can approximate the sensitivity matrix for the curve by choosing n samples and summing the effects of the parameters on these points,

$$\mathbf{M} = \frac{1}{n} \sum_{u_r \in [0,1]} \left(\frac{\partial \mathbf{f}(\mathbf{q}, u_r)}{\partial \mathbf{q}}\right)^T \frac{\partial \mathbf{f}(\mathbf{q}, u_r)}{\partial \mathbf{q}},$$

which allows us to update the parameters of the curves as

$$\dot{\mathbf{q}} = \mathbf{M}^{-1}\mathbf{g}.$$

The same procedure applies to any parametric curve. To define a new one, we need only provide routines to evaluate $\mathbf{f}$ and $\partial \mathbf{f}/\partial \mathbf{q}$. The creation of this code can be automated using symbolic differentiation so a new variety of curve can be added to a system by merely specifying the expression for its characteristic function. We have implemented automatic code generation systems which allow users to enter mathematical expressions which define parametric curves and have them added to our geometric modeling programs.

3 Constraints

Constraints provide a powerful mechanism for restricting how objects change, which is important in differential manipulation: pulling on an object can affect all attributes of the object, which is often not what the user wants. Constraints also provide a mechanism for building compound objects by specifying how the components are related. Differential manipulation includes constraints in a natural way: just as geometric objects can be viewed as physical entities, geometric constraints can be viewed as mechanical interactions.

Constraints can be written as relations on the state. If we start in an unknown state, the constraints might be violated, requiring the solution of the constraint equations. Since there are no good general methods for solving systems of nonlinear equations[PFTV86], most constraint solving systems limit the class of equations they admit.

Instead of relying on nonlinear system solving, we approach the constraint problem differentially. Rather than solving the constraints, we *maintain* them as the system moves. We call this approach *constrained dynamics* since it simulates the behavior of a constrained physical system.

To visualize the difference between constraint solving and constrained dynamics, imagine doing geometric design by placing two pieces of string on a table and pulling them around. Suppose we would like to have the constraint that the ends of the two strings touch. A constraint solver, if it succeeds, will re-arrange the strings on the table so that the constraints are met. After the solver does its work, if we move a string the constraints will be violated again. In contrast, constrained dynamics ties the ends of the strings together so that pulling on a string reshapes the whole system without violating the constraint.

In a constrained dynamics application, the constraints must still be solved because the system must initially get to a feasible state. However, this only needs to happen incrementally when a new constraint is applied.

Placing the user in the solving loop by making the constraints gradually come together has several advantages as well. The system does not jump to a new state, which may surprise the user. The user can help guide the solution away from incorrect solutions and past places where the solving process gets stuck.

3.1 Techniques for Constrained Dynamics

The techniques for implementing constrained dynamics in physical simulations have been presented elsewhere [WGW90, GW91, Pla89]. The presentation here follows that of [WGW90], except with an emphasis on first order physics and the demands of differential manipulation. We consider here equality constraints, which can be written as $f(\mathbf{q}) = 0$, noting that the techniques extend to inequalities.

A simple way to implement constraints is to use springs to enforce them approximately. This approach is called the penalty method because it uses the springs as a penalty for the system's violation of the constraints. This method has the advantage that if the constraints are not met, the spring can pull the object together. Unfortunately, it has the problem that the constraints will not be maintained precisely. Making the springs stiffer reduces the amount that things pull apart, but also makes the differential equations more difficult to integrate.

Our formulation for constrained dynamics removes the part of the applied force $\mathbf{f_a}$ which would cause the system to move in a way which would violate the constraints. We do this by computing a constraint force $\mathbf{f_c}$ which counteracts this component of the applied force.

To see how the method works, consider constraining a point to lie on a circle, and suppose that the constraint is met, as in Figure 1. We know that if the point moves, it must move in a way that will not violate the constraint: its motion must be tangent to the circle. Since it will move in the direction of net force, the force must be tangent to the circle[2]. If the applied force is not in this direction, the constraint must provide a force to cancel out the illegal component.

We write the constraints as a function of state $\mathbf{c} = f(\mathbf{q})$. Assume that, at the present instant, all of the constraints are met ($\mathbf{c} = 0$). In order for the constraints to be maintained, $\mathbf{c}$ must be unchanging, so $\dot{\mathbf{c}}$ (and higher derivatives) must equal zero. From the chain rule,

[2]This is exactly true in the first order case. The second order case is slightly more complicated. See the derivation in [WGW90] for details.

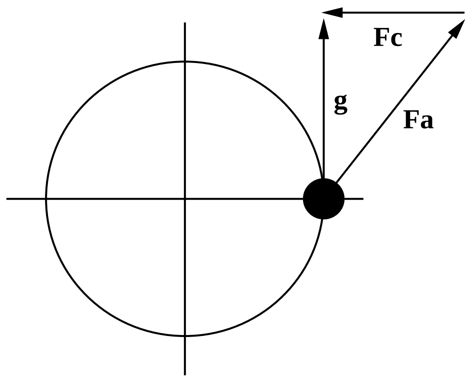

Figure 1: A point is constrained to a circle. When a force $\mathbf{f_a}$ is applied, a constraint force $\mathbf{f_c}$ is computed to ensure that the net force **g** lies in a legal direction, tangent to the circle.

$$\dot{\mathbf{c}} = \frac{\partial \mathbf{c}}{\partial \mathbf{q}} \dot{\mathbf{q}}.$$

Denoting the constraint Jacobian by $\mathbf{J} = \frac{\partial \mathbf{c}}{\partial \mathbf{q}}$, we obtain

$$\dot{\mathbf{q}} = \mathbf{JW}\mathrm{g} = 0$$

by substitution with the equations of motion (2). Since net force is the sum of the applied and constraint forces, $\mathrm{g} = \mathbf{f_a} + \mathbf{f_c}$, we get

$$\mathbf{JWf_c} = -\mathbf{JWf_a}, \tag{3}$$

which is a system of *linear* equations with only the constraint force vector $\mathbf{f_c}$ unknown.

In words, equation (3) just says that the constraint force, added into the applied force, must cause the first time derivative of the constraints to be zero. This condition is generally too weak: if the system is underconstrained, as is usually the case (otherwise nothing can move at all!) we have fewer equations than unknowns, and there exist many values for $\mathbf{f_c}$ that satisfy equation (3). One way to handle this ambiguity is to modify the applied force as little as possible, computing the least squares solution to the linear system. Some linear system solvers, such as conjugate gradient-based approaches[PFTV86], have this property.

Alternatively, we can remove the ambiguity in a more physically correct manner by requiring that the system obey D'Alembert's Principle of Virtual Work[Lan86] which requires that the constraints do not change the energy in the system. For this to occur, the constraint forces must lie in the null space complement of the Jacobian, so that $\mathbf{f_c} = \lambda\mathbf{J}$ for some vector λ. We write the equation as

$$\mathbf{JWJ}^T\lambda = -\mathbf{JWf_a},$$

and solve for λ, the vector of *Lagrange multipliers.*

Computing the constraint forces requires solving a *linear* system, even though the constraints may be nonlinear. The linear equations can be solved efficiently by exploiting the sparsity of the matrix.

The constraint formulation assumes that the constraints are met. In order to allow for numerical drift and initial conditions which violate the constraints, we use feedback to pull the system back into a legal state. This is implemented by adding a force in the direction of the constraint Jacobian proportional to the deviation of the constraint,

$$\dot{\mathbf{q}} = \mathbf{W}(\mathbf{f_a} + \mathbf{f_c}) - k\mathbf{c}\mathbf{J}^T,$$

effectively using a spring to pull the system together.

Like the objects they connect, constraints have a very regular structure. Given the function which defines the constraint, the Jacobian can be computed. The constraint function will be a composition of some functions of the objects and the function particular to the type of constraint. This makes it very useful to have tools, such as [GW91], for composing function and evaluating them and their derivatives.

4 An Example Application

As an example of how differential manipulation provides a direct interface to a wide variety of geometric objects without having the programmer develop object-specific interaction techniques, we will describe our prototype two dimensional geometric modeler, shown in Figure 2.

The two dimensional modeler lets the user create a wide variety of parametric curves, grab them, and pull on them. Each type of object is manipulated in exactly the same way: the user can grab any point on it and pull it. Splines are manipulated freely, without regard to their control points, and all of the degrees of freedom of ellipses can be controlled. The program also provides a reasonable interface to many more unusual objects, such as spirals.

The set of geometric entities which the user can put in a drawing is easily extended. The parametric function defining a curve, along with some auxiliary information

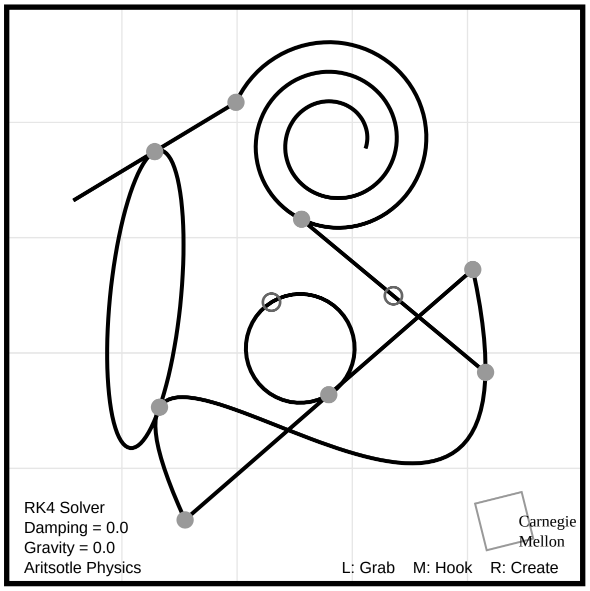
RK4 Solver
Damping = 0.0
Gravity = 0.0
Aritsotle Physics

Figure 2: Our 2D geometric modeler allows the user to manipulate a wide variety of shapes and connect them together with constraints.

such as the name to put on the menu, is given to a program which automatically creates the required code and extends the geometric modeler. No code needs to be manually modified to add a new object to the system, and no thought needs to be given to designing an object-specific interface. We could have implemented this process dynamically, allowing the user to add new varieties of objects.

Constraints, such as connecting objects or making them parallel or equidistant, can be easily specified, allowing the user to create compound objects with non-trivial behaviors. Implementing a new variety of constraint is also simple; most of a constraint's implementation is created by a code generation program. However, in our present implementation addition of new constraint types is not yet fully automated; the developer must hand-code the user interface for specifying the constraints as well as some utility methods.

The ease of extending this system is made possible by an object-oriented decomposition of the problem specifying a well defined set of operations which classes of constraints and objects must provide [WGW90, GW91].

5 Other Applications

Differential manipulation appears to have applicability in a wide range of domains. In our group we have explored applications including geometric design in two and three dimensions[WGW90, GW91], animation[WW90], and motion tracking [RW91]. The techniques of differential manipulation are not limited to geometry. We can use the paradigm to manipulate any mathematical model in which a user is interested in outputs which are differentiable functions of state.

Differential manipulation attempts to mimic interactions with physical objects in the real world. Since all of us have substantial experience in dealing with the physical world, programs with physical interfaces can potentially draw on our skills and intuitions[Smi87].

By providing a uniform mechanism for interacting with a wide class of geometry, differential manipulation provides a useful extension to direct manipulation. Users can manipulate all objects the same way: by grabbing them and pulling on them. Developers need not design object-specific interfaces and are freed to use objects and representations for which no direct mapping of parameters to user controls is available. Implemented by a simplified physical simulation, the paradigm includes constraints gracefully.

References

[Cor88] Claris Corp. Macdraw II, 1988. Computer program.

[GW91] 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.

[Lan86] Cornelius Lancoz. *The Variational Principles of Mechanics*. Dover Publications, 1986.

[LV89] Mark Linton and John Vlissides. Idraw. Computer program, 1989.

[Nor90] Donald Norman. *The Design of Everyday Things*. Doubleday, 1990.

[PFTV86] William Press, Brian Flannery, Saul Teukolsky, and William Vetterling. *Numerical Recipes in C*. Cambridge University Press, Cambridge, England, 1986.

[Pla89] John Platt. *Constraint Methods for Neural Networks and Computer Graphics*. PhD thesis, California Institute of Technology, 1989.

[RW91] James Rehg and Andrew Witkin. Visual tracking with deformation models. In *Proceedings of the IEEE International Conference on Robotics and Automation*, April 1991.

[Shn83] Ben Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer Graphics and Applications*, 16(8):57–69, August 1983.

[Smi87] Randall Smith. Experiences with the alternate reality kit: An example of the tension between literalism and magic. In *Proceedings CHI + GI 87*, pages 61–67, 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.

[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.