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.