Practical Issues in Graphical Constraints

Michael Gleicher
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
gleicher@cs.cmu.edu

Abstract

Use of constraint-based techniques in interactive graphics applications poses a variety of unique challenges to system implementors. This paper begins by describing how interface concerns create demands on interactive, constraint-based, graphical applications. We will discuss why such applications must be able to handle systems of non-linear constraints, and survey some of the techniques available to solve them. Employing these numerical algorithms in the contexts of interactive systems provides a set of challenges, including dynamically setting up the equations to be solved and achieving adequate performance and scalability. This paper will explore these issues and describe the methods we have used in our efforts to address them.

1 Introduction

The ability to represent and maintain relationships among objects can be an extremely useful tool in a graphical application. Since the earliest interactive graphical applications[29], the use of such constraint techniques has been demonstrated in applications including drawing, 3d modeling, user interface construction, animation, and design.

In employing constraint-based techniques in such graphical applications, system designers must face a variety of new challenges. This paper aims to describe some of these challenges, and discuss some techniques to address them. We begin by looking at how usability concerns for such applications create demands on what systems must do. We will discuss why these applications will often demand the power and generality of non-linear numerical techniques. Issues in employing such algorithms within interactive systems will be surveyed, with an emphasis on how the equations to be solved can be set up and how adequate scalability and performance might be achieved.

For this paper, we are concerned with the class of graphical applications where the user creates and edits models made up of a number of graphical objects. An example is an object-oriented drawing program, where the model (or drawing in this case) is made up of lines and circles, but not a painting program in which the model is a bitmap or image. Also, the constraints that we are concerned with in this paper are those used within the model, for example, to enforce a relationship within a drawing. This is different from the common use of constraints in user interface construction, where a constraint is used by the programmer to enforce internal consistency within the program, as in systems like Thinglab II[20], Garnet[21], or Mel[16]. These two uses of constraints provide different sets of challenges, although some of the issues and solutions presented here apply to both.

2 The Challenges of Constraint-Based Graphical Applications

An interactive graphical application with constraints must contend with the same basic challenges as those without constraints. However, there are challenges which are inherent when constraints among objects are provided. Constraints obviously add to models, in addition to the graphical objects, constrained models also contain constraints which must be stored, displayed, edited, saved, etc. More significantly, 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. It introduces challenges in implementation, in performance, and in usability. The latter is potentially most concerning, not only for its difficulty, but also because usability concerns create further challenges for implementation and performance.

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 isn't what is desired or expected. The most obvious form of bug is when the constraints force the model into a configuration which 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 also must be handled, as it is impractical to expect the user to fully 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. Providing continuous motion animation seems to be important for helping users understand complicated behaviors as it allows them to use their perceptual skills to connect states This places demands on systems to provide rapid enough iterations to provide the illusion of continuous motion. Another important weapon in avoiding constraint bugs is the development of specification techniques which help avoid them; this is evidenced by the large effort in automatically inferring constraints, such as [1, 13, 18].

The interactive nature of constraint-based graphical application also causes the systems of constraints to be dynamic. Typically, as the user edits the model, constraints are added, removed, and altered. While there are some applications, such as [27], where it is possible to separate manipulation and modeling, applications must typically interleave altering the constraints with solving them. The ability to rapidly alter the set of constraints can also be used to create new facilities in the solver. For example, switching a constraint on and off at the correct times permits the creation of inequality constraints, using what are called active-set methods[6].

With all of the discussion of constraints, it is easy to lose sight of the fact that constraints are usually a tool to aid in the process of creating graphical models. The constraint solver must not get in the way of the creative process. This leads to demands of reliability and robustness for solvers. It also means that users should not be forced to deal with equations. Artifacts of the solving process must be hidden from the user. For example, users should not be forced to create constraints which have properties which stem only from solver limitations; for example, some solvers require constraints to be expressible as directed acyclic graphs. Similarly, we cannot expect the user to tweak algorithms on a per problem basis, as is often the case in numerical analysis.

2.1 An Example Application

A constraint-based drawing program demonstrates how the issues in building interactive constraint-based graphical applications manifest themselves. The idea of using constraints in a drawing program is not new; in fact, it dates back to one of the earliest interactive systems[29]. However, despite the nearly universal agreement on their utility, constraints never really caught on in graphical applications. What has been successful are direct manipulation programs.

We have built a drawing program called *Briar*[7, 13], depicted in Figure 1 which aimed to keep the the features of the successful direct manipulation systems, but to augment them with constraints[8]. Briar is based on an existing, highly evolved direct manipulation drawing techniques[2], and augments them by making the relationships between objects persistent. Briar

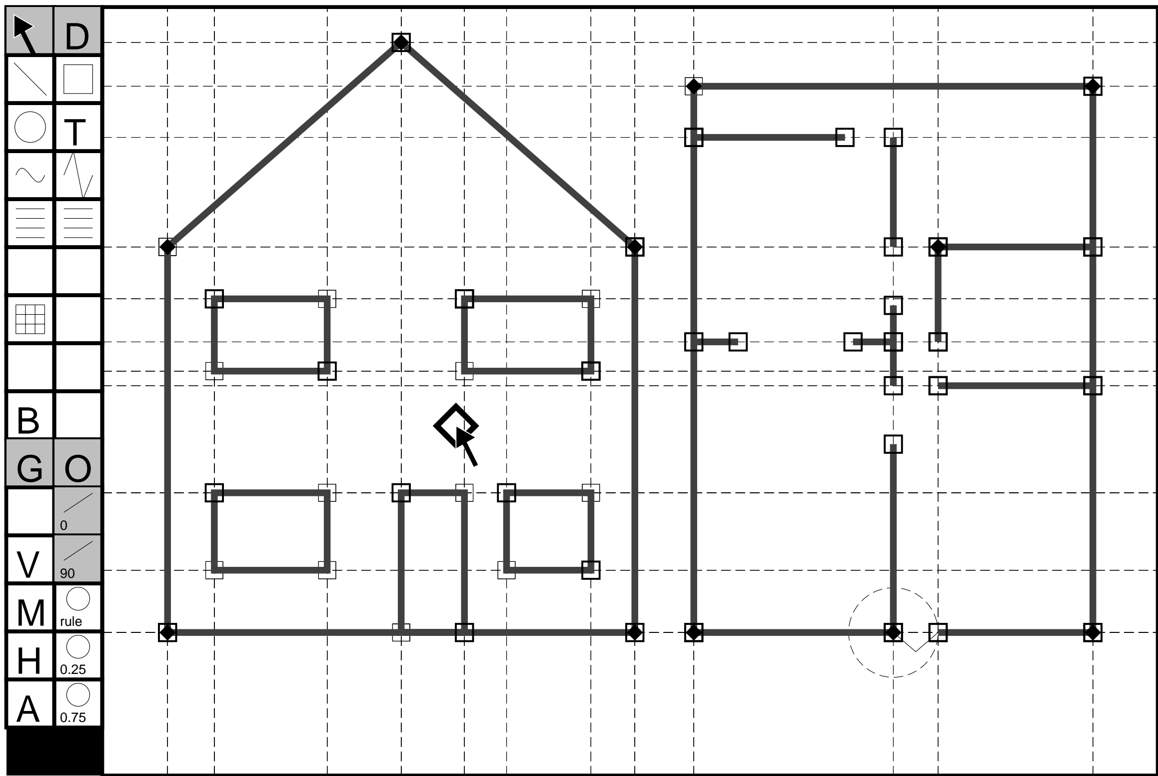

Figure 1: The *Briar* graphical editor editing a constrained drawing.

provides a set of Snap-Dragging features to help users draw precisely and quickly. However, unlike other Snap-Dragging systems, Briar provides the facility of making these snapping operations into persistent constraints. This can be done without the user explicitly specifying the constraints. Briar uses a visual language for displaying the constraints, which closely parallels the snap specifications, and also provides simple methods for deleting constraints based on drawing operations. The avoidance of direct reference to constraints avoids several classes of constraint bugs including conflicts[8].

Despite its constraint features, Briar maintains the fundamental direct manipulation feel of dragging. Like more traditional programs, objects are dragged and move with continuous motion, except that in Briar, constraints among the objects can be maintained. For example, the user can draw a mechanical contraption, and have it stay together when the crank is turned.

Briar only represents two types of constraints: point–on–object and point–on–point. More complicated relationships, such as distance, orientation, or co-linearity, are created by combining these simple elements with special alignment objects. These more complex relationships lead to non-linear equations which Briar's solver must handle. Hierarchical grouping with rotation and interesting objects also lead to non-linearities. Briar uses a visual representation for constraints so the user is never confronted with equations.

Another important aspect of the constraints in Briar is that they are dynamic. The user is continually creating and destroying constraints. These changes occur during drawing operations, so it would be unacceptable if adding or deleting a constraint were time consuming.

3 The Need for General Purpose Solving

The interface needs of interactive graphical applications place difficult performance demands on constraint algorithms. In order to keep up with interactive rates, it is tempting to make restrictions on the types of constraints which the solver can handle. However, the things which it is most tempting to restrict are exactly those which things from which a constraint-based approach derives much of its value. Namely, it is tempting to limit the class of equations that the solver can handle, ad it is tempting to limit the ways that constraints can be combined simultaneously.

One of the most obvious simplifications to a constraint system is to restrict the class of constraints it can handle, as simpler types of constraints usually have simpler, faster solving algorithms. However, for interactive graphical applications, restricting the set of equations is unacceptable. Even in the most basic 2d applications, non-linear equations arise. Simple geometric relationships, such as distance and orientation, give rise to non-linear equations. Many graphical objects are most easily represented in ways that give rise to non-linear functions. Similarly in 3D, many of the interesting relationships among objects require non-linear equations.

A general purpose non-linear solver frees interface developers to experiment with novel and

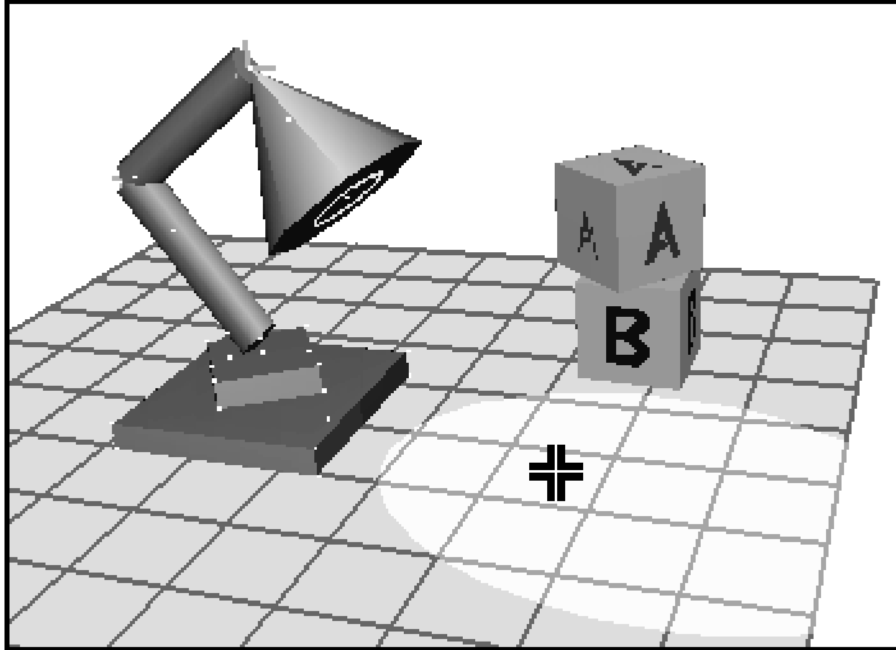

Figure 2: A lamp is manipulated by dragging the point at the center of where its light hits the floor. The constraint solver can adjust joint angles based on the manipulation of the light's target, as well as positions of the light.

unusual constraints since they need not worry about how to solve new types of constraints. For example, in our work we have experimented with constraints on such things as the outcome of viewing transformations[11], on the positions of reflections and shadows, and on the results of lighting calculations, permitting constraining the colors that objects appear[9]. Such constraints are interesting because permit users to control models directly in terms of aspects they are interested in, as demonstrated by the example of Figure 2. Such constraints are also interesting because they can be used in combination with one another to create other interesting interaction techniques.

The ability to solve constraints simultaneously is one of the biggest advantages for the user of a constraint-based system. Simultaneity helps avoid redundant work, as constraints can keep things previously done on the drawing persistent over subsequent editing operations. Being able to arbitrarily combine constraints permits the user to specify parts of the model in an unordered fashion. It permits users to mix and match specification tactics to meet their needs. It also aids in explorations of design spaces, by permitting the user to constrain aspects of the design and altering the free parameters to explore what configurations are consistent with these constraints. The ability

to combine constraints also can also aid interface designers as it provides an interesting way to define interaction techniques[9].

Although the ability to solve constraints simultaneously is crucial to interactive graphical applications, some common classes of solvers severely restrict how constraints can be combined in an effort to achieve better performance. For example, the popular propagation solvers can only solve combinations of constraints which can be solved one at a time by some ordering. While some solvers, such as Delta Blue[5], employ sophisticated algorithms to find these orderings, they still place substantial restrictions on what constraints can be combined. Also, the distinction between what the solvers can and cannot do is based on aspects of the data structures, for example that the constraint graph is acyclic, which is probably not what the user of a graphical editor is thinking about.

3.1 Properties of Constraint Problems

Not only must the constraint solver in a constraint-based graphical application handle non-linear equations, but it must also manage equations with some other difficult properties. Underconstrained, overconstrained, and cases of redundant constraints may arise.

It is impractical to require the user to specify all aspect of a geometric model. For example, there may be some uncertainty in the design. In general, it is difficult to tell if a model is fully-specified. It is therefore important that the constraint solver can handle these underdetermined cases. What this requires is that the solver is able to choose from all of the possible solutions in the underconstrained case. The best choice would be for the solver to select the solution which the user wants, something that is impossible without some mind reading. This means that a heuristic is needed. Even if the goal of choosing an underconstrained solution is relaxed to simply pick the one which will surprise the user the least, e.g. the "Principle of Least Astonishment"[4], some for of operation metric is needed.

A common heuristic for selecting an underconstrained solution is minimize the amount that the model changes in order to meet the constraints. A justification for this heuristic is that if the user did not ask for something to change, the system should minimize the amount that it is changed. There are many ways to measure the amount the model changes, ranging from minimizing the amount of work done by the solver[14] to optimization metrics such as least-squares norms or even user-defined optimization criteria, as provided by Ascend[23]. A discussion of some of the options for these optimization criteria are given in [**?**]. In our work, we have chosen least-squares as it provides a global metric which distributes change in a predictable and uniform way, and it can be implemented easily in the context of non-linear constraint techniques.

Although they are often easier to prevent than underconstrained cases, overdetermined constraints still arise in interactive graphical applications. Such situations are often caused when users

attempt to specify more and more of their model when the heuristics used to choose underconstrained solutions fail. Overconstrained cases are a problem both in the case of redundancy, where many constraints all lead to a consisted solution, and conflicts, where there will be no solution which satisfies all the constraints. Redundant constraints are difficult for solvers, as a little numerical inaccuracy can make them appear to be conflicts. Conflicting constraints are difficult for non-linear solvers as it is not always possible to determine if there is no solution, or if the solver merely has no found it yet.

There are several possible behaviors which may be desirable when constraints conflict. One useful way to resolve conflicts is to attempt to minimize the error residual, for example in a weighted least-squares sense. This might be unacceptable if the constraints are meant to represent rules which cannot be violated. Another option is to provide a preference ordering, or hierarchy, for the constraints[4]. In such a scheme, more important constraints must be satisfied before less important ones. Providing a general hierarchy mechanism for non-linear constraints appears to be a difficult problem. However, the techniques which we have used provide a two-level hierarchy of "hard" and "soft" controls[11], with weighted least-squares minimization at each level of the hierarchy. It is possible to achieve a third level to this hierarchy by using effectively infinite weights on some of the hard constraints, but this can lead to numerical problems when there are conflicts or redundancy among these constraints.

4 Solving Constraints

A constraint in an interactive graphical application is a relationship, typically geometric, among objects. A constraint is represented by an equation which must hold for the constraint to be satisfied. This equation is over the variables which determine the configuration of the model, called the *state vector.* The standard form of these equations is to write them as some function of the state vector equals a constant, often zero with no loss of generality. Inequalities are similarly handled by an equation which states that the function is greater than zero. This function is called the *constraint function.* The job of a constraint solver is to find configurations of the state vector for which the constraints hold. If continuous motion is to be achieved, the solver must be called for each frame.

A guaranteed general method for solving systems of non-linear equations does not exist, and there are arguments that such a method cannot exist[25]. However, there are methods, which despite their lack of total generality and guarantees, perform reasonably on realistic problems. These methods iteratively converge on a solution to the equations. A well known iterative method for solving non-linear systems is Newton's method, which has many variants and is the basis for many of the more sophisticated techniques. For each iteration, these methods solve a linear system based on the derivatives of the constraint equations to compute the next value.

We have been using a variant of these iterative approaches in our work, which we call *differential*

methods. Rather than specifying what the desired values for constraint functions are, we instead specify how they are to change, e.g. their time derivatives. For example, we might specify that something is not to change, or is to move towards a target. Such an approach is useful for interaction because we want our objects to move continuously, rather than jump to their goal positions. The derivatives of variables can be computed from the derivatives of the constraint functions by solving a system of linear equations, even if the constraint functions themselves are non-linear. Differential methods are detailed in [10] and [11].

Differential methods and Newton-like methods are very similar. Both repeatedly solve linear systems to iteratively move towards satisfying non-linear equations. One way to describe the difference might be that a Newton-style method attempts to race towards the goal as fast as possible, regardless of the route taken, while the differential methods try to take a good continuous route, even if it takes longer. The former method runs more of the risk of speeding off in a wrong direction, but may arrive at its goal sooner.

The method actually used to solve constraints is an implementation detail to which the user of a constraint-based application. What is important to making such a system successful, however, is that constraints are solved in a manner that is fast, robust, and reliable.

4.1 Solving Linear Systems

Almost any method chosen will repeatedly solve systems of linear equations based on the derivatives of the non-linear equations.[1] Linear system solving dominates the computational complexity of constraint-based graphical applications (see section 5), and is a key place where stability and reliability concerns must be met.

The linear system solver at the core of the constraint solver must be able to handle ill-conditioned and singular cases. Such cases arise not when the constraints are over-specified, but also arise from certain types of geometric configurations[19]. To better handle these cases, we use a variant of damping, a technique seen in robotics[22, 30] and in the Levenberg-Markardt method[25]. Such methods add small amounts to the diagonal elements of the matrix raising their condition number and permitting them to be solved more easily. The method effectively trades some precision in the constraint calculation for better performance and stability.

To solve linear systems in our constraint applications, we have used a conjugate-gradient algorithm. This algorithm is particularly attractive because it permit exploiting sparsity without pre-analysis. This is useful because the structure of the sparsity of our matrices changes as the user alters the constraints. Because it conjugate-gradient an iterative technique, we have the ability to trade precision for performance by adjusting tolerance parameters. While there are techniques,

[1] There are non-global methods for solving non-linear equations which do not solve linear systems. In [24] these *relaxation* or *penalty* methods are described, including a discussion of why they are not good.

such as singular value decomposition, which better handle singular and near-singular matrices, these techniques do not exploit sparsity as easily nor permit the performance adjustments. In [26], the tradeoffs between SVD and conjugate-gradient are explored more closely.

5 Scalability and Performance

Performance is important to interactive graphical applications so they can achieve the appearance of continuous motion. In a conventional drawing program this can be achieved easily as only one object is moving at a time. If it is a complicated or compound object, it can be drawn in a simpler form, such as a bounding rectangle, because it cannot change internally. Because of this constant $O(1)$ complexity in the interactive loop, direct manipulation drawing is practical on small computers.

In a constraint-based system, the constant time portions of drawing systems no longer exist. Many objects can potentially change at once. Where this complexity hurts the most is in constraint solving, but the fact that many object move simultaneously also makes other things, like redraw, more complicated. Multiple objects moving also raises the cognitive complexity of drawing, as the behaviors become quite complicated.

5.1 Computational Complexity

The computational complexity of constraint-based graphical applications using iterative numerical algorithms is dominated by the linear systems which must be solved in order to solve the non-linear equations. Solving a system of linear equations is, in the most general case, an $O(n^3)$ problem. However, because the matrices which arise in graphical constraint problems are typically sparse, the complexity can be lower. Because each constraint only affects at most a small constant number of objects, its function can only depend on a small constant number of variables so the corresponding row of the matrix can only have a similarly small number of entries. Therefore, the matrices only have $O(n)$ entries in them, and can the linear systems can be solved in $O(n^2)$ time[26]. For certain classes of constraint problems, the linear system can be solved in linear time[28].

To maintain interactive performance, it is critical to reduce the complexity of solving algorithm by exploiting the sparsity of the systems which are solved. However, without severely restricting the class of models which the user can build, this still leaves greater than linear complexity. Since we cannot reduce the polynomial coefficient, one must instead reduce the size of the problems which are solved, without imposing size restrictions on the user. Tactics available for this include partitioning the constraints into smaller subproblems, which is explored in [26], and removing "dead" objects and constraint from the computations.

The basic strategy for reducing problem size is to determine which objects might move and only operate on these objects. Once the set of objects is pared, constraints which depend only

on dead objects can be removed. Constraints which depend on both live and dead variables will only alter the live ones. Information about whether or not an object is alive can be used for other purposes, such as snap-target pruning[13], or simplifying displays.

There are many sources of objects to remove from the constraint system to reduce the size of the problems which must be solved. One obvious source is the user, who, for example, might want to freeze an object. It is also useful to cluster variables and allow the user to select whether these classes should be enabled or not. For example, in a 3d modeling system, the constraint solver might be used to control lighting, position the camera, and shape object geometry. However, the user will often only want to control one of these at a time. It is therefore useful to allow disabling entire classes of objects. This is especially important with constraints such as through-the-lens camera controls[11] which can affect many different types of variables.

The are two reasons to automatically disable an object or variable. The constraints might completely restrict an object (or a variable) from changing, or the object might not be connected to anything which might cause it to move, such as the mouse. For these two questions, exact answers are unavailable in general. It might require proving an arbitrarily hard geometric theorem, or doing expensive numerical calculations. However, in practice one can remove many objects, although it is difficult to guarantee the smallest possible set of active variables. In our systems, we have used techniques such as dependency analysis and simple geometric theorem proving.

5.2 Cheap Constraints

Disallowing a parameter to change by removing it from the list of variables is an example of what we call a *cheap constraint*. Rather than constraining the behavior by adding another constraint equation, cheap constraints alter the set of variables to make it smaller. Where adding a regular constraint causes the system to slow down because it must solve a larger set of equations, cheap constraints can actually cause things to speed up as they reduce the size of the problem.

The simplest form of a cheap constraint is to freeze an object. If the user does not want an object to change, its parameters are merely removed from the variable list so that the solver is unable to change them. Subsets of parameters can similarly be frozen. For example, if a line segment is represented by the position of its center, its length, and its orientation, a constraint maintaining the length of the line segment to stay at its present value could be created as a cheap constraint. Notice that such cheap constraints are representation dependent: 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 a cheap constraint. It is conceivable to build a system which changes the representation of objects to maximize the number of cheap constraints, however, we believe the combinatorial optimization problem to be intractable. What is commonly done, however, is to pick representations which are most likely to yield the most cheap constraints.

Another form of cheap constraint is merging, that is having multiple parameters access the same variable, as seen in Thinglab[3]. Merging is a cheap constraint for equating parameters. Because merged parameters share a single variable, they have exactly the same value. While this exact equality can be an asset, it can also be a problem as it means the constraint behaves differently than its non-cheap 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 overconstrained problem. Most forms of cheap constraints have similar difficulties.

5.3 Trading Accuracy for Performance

For many applications, users are not concerned with extreme accuracy — we are often willing to let our objects be an imperceptible tenth of a pixel apart if it allows our solver to be faster. When we employ iterative numerical solvers, there are several ways in which accuracy can be traded for performance. The simplest is to raise the solvers tolerance, causing it to stop iterating sooner. When an iterative linear system solver is used within the non-linear method, the tolerance for this solver can also raised.

The damping techniques mentioned in section 4.1 are another example of trading accuracy for performance. The perturbed, or damped, linear system does not exactly model the problem being solved. However, it does prevent the linear systems from becoming singular which most solvers cannot tolerate. Also, some iterative linear system solvers solve better conditioned linear systems faster.

6 Formulating Equations

Most of the methods for solving systems non-linear equations described in section 4 have similar requirements for what the equations need to provide. They need to be able to evaluate the constraint functions and their derivatives to form the linear systems which are solved. These evaluations must be able to take advantage of sparsity in the resulting derivative matrices to achieve needed performance in evaluations. Since these functions are defined in response to dynamic creation and destruction of constraints, the creation of the functions themselves must be dynamic.

Constraint-based applications must be able to dynamically define functions. They need to be able to rapidly evaluate these functions and their derivatives. These functions are built by composing other functions together. At a low level, one might consider building functions out of basic mathematical primitives such as addition; however, this composition occurs at higher levels of abstraction in constraint systems too. For example, many constraints are typically defined in terms of points on objects. It is then the job of the objects to compute these point positions in terms of their state variables. This provides an important layer of modularity: the constraints can be defined independently of the objects.

A function can be represented as a directed acyclic graph[2] with composed functions at the nodes, and arcs representing composition. Our approach to providing function composition in the dynamic setting of interactive applications is to provide a tools for managing these function graph structures. In *Snap-Together Math*[12], function elements are "wired" together to make more complicated functions.

Evaluating the values and derivatives of an expression involves traversing the graph. To compute a value, a node requests the values from its predecessors and then performs its local function on these results. The chain rule for derivatives provides a similar method for computing derivatives. To compute the derivative of a node with respect to the global inputs, a node asks its predecessors for their derivatives, and multiplies these intermediate results by the derivative of its internal derivative matrix. This process is called *automatic differentiation,* and is superior to building the global derivative matrix symbolically in most situations[15, 17].

At the leaves of the function graph are the variables over which the function is computed, the state vector of the model. The state vector contains the parameters of the graphical objects which make up the model. The state of the system is distributed among objects, however, numerical algorithms require state to be gathered into a single global vector. The positions of variables in this vector are significant as they determine which columns of the derivative matrices correspond to which variables. It would be possible to keep variables in a large vector and have the objects simply look in this larger structure for values. Such an approach, however, violates encapsulation of objects and makes it more difficult to switch variables on and off. In our approach, variables from object state vectors are gathered into a larger vector when needed. By selecting which variables are gathered, it is simple and fast to switch among sets of variables.

We have implemented function composition and automatic differentiation in an object-oriented tool called *Snap-Together Math.* Rather than requiring special graph node objects, it allows application objects to mix in the ability to "speak mathematics." This permits application objects to participate directly in calculations. They must only respond to a simple protocol. This simplifies applications by reducing the need for special math objects which must be allocated and maintained. Snap-Together Mathematics uses a specially designed sparse matrix representation and does extensive caching to achieve better performance.

7 Putting it Together

A wide variety of graphical applications might employ constraints. Any of these applications will face the same issues previously described. Fortunately, the solutions proposed are general enough to apply across many applications, and can be encapsulated into a toolkit to support a variety of applications.

[2] It is not a tree as common subexpressions might be shared.

At a low level, any application which employs numerical constraint will gain leverage from a library of mathematical structures and algorithms. Our mathematical toolkit provides support for such things as vectors, matrices, and differential equations in an object oriented manner. It includes several varieties of sparse matrices, and a variety of linear system, non-linear system, and ordinary differential equation solvers.

Snap-Together Math, described in section 6 is built on top of the mathematical library. In addition to the support for dynamically composing and rapidly evaluating functions, it also includes interface to differential and Newton-style solvers. Many applications, including Briar, have been built with these tools.

The Bramble graphics toolkit[9], built on top of Snap-Together Math, aims to provide a framework for building graphical applications with constraints. Previous toolkits, such as Garnet[21], Thinglab II[20], and Mel[16], employ constraints to aid programmers in application development. Bramble, in contrast, primarily aims to provide constraints as for user level services. Although, it does appear that the the differential constraint techniques provided in Bramble simplify the task of building graphical applications by helping separate manipulation from representation and facilitating general purpose interaction techniques.

There are many drawbacks to the use of iterative numerical algorithms in interactive graphical applications. Such computations can be expensive and have worse computational complexity than their non-numerical counterparts. The techniques only apply to real numbers, which means other techniques must be used for continuous data within applications. Care must be taken to avoid numerical instability. Also, the set of equations for which non-linear constraint solvers will perform well is not a well defined set – often such determinations must be done empirically.

Despite these drawbacks, it is still advantageous to employ numerical non-linear constraint methods in graphical applications. The generality in the class of equations allows the constraints typically required by graphical editors, and allows experimentation with new types of constraints. Such constraints can be handled simultaneously. Most non-linear methods require very little information about the constraint functions. That information which they do can be computed automatically. The iterative non-linear methods can be made to gracefully handle under and overdetermined constraints, using least-squares techniques.

The addition of constraints to an interactive graphical application creates a variety of issues which system builders must be concerned with. Many of these issues stem from interface concerns and the need to handle non-linear relationships. Achieving the needed performance and dynamicness from the non-linear solvers requires careful attention. However, support for these can be provided in a general purpose manner.

References

[1] Sherman R. Alpert. Graceful interaction with graphical constraints. *IEEE Computer Graphics and Applications*, pages 82–91, March 1993.

[2] Eric Bier and Maureen Stone. Snap-dragging. *Computer Graphics*, 20(4):233–240, 1986. Proceedings SIGGRAPH '86.

[3] 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.

[4] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. In *Proceedings OOPSLA*, pages 48–60, October 1987.

[5] Bjorn Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint hierarchy solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[6] Phillip Gill, Walter Murray, and Margret Wright. *Practical Optimization*. Academic Press, New York, NY, 1981.

[7] Michael Gleicher. Briar - a constraint-based drawing program. In *SIGGRAPH Video Review*, volume 77, 1992. CHI '92 Formal Video Program.

[8] Michael Gleicher. Integrating constraints and direct manipulation. In *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, pages 171–174, March 1992.

[9] Michael Gleicher. A graphics toolkit based on differential constraints. In Randy Pausch, editor, *Proceedings UIST '93*, November 1993.

[10] Michael Gleicher and Andrew Witkin. Differential manipulation. *Graphics Interface*, pages 61–67, June 1991.

[11] Michael Gleicher and Andrew Witkin. Through-the-lens camera control. *Computer Graphics*, 26(2):331–340, July 1992. Proceedings Siggraph '92.

[12] Michael Gleicher and Andrew Witkin. Supporting numerical computations in interactive contexts. In Tom Calvert, editor, *Graphics Interface*, pages 138–145, May 1993.

[13] Michael Gleicher and Andrew Witkin. Drawing with constraints. *The Visual Computer*, To Appear.

[14] James Gosling. *Algebraic Constraints*. PhD thesis, Carnegie Mellon University, May 1983.

[15] Andreas Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic, 1989.

[16] 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.

[17] 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.

[18] David Kurlander and Stephen Feiner. Inferring constraints from multiple snapshots. Technical Report CUCS-008-91, Columbia University, May 1991.

[19] Anthony Maciejewski. Dealing with the ill-conditioned equations of motion for articulated figures. *IEEE Computer Graphics and Applications*, May 1990.

[20] John Harold Maloney. *Using Constraints for User Interface Construction*. PhD thesis, University of Washington, 1991. Appears as Computer Science Technical Report 91-08-12.

[21] 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.

[22] Yoshiko Nakamura. *Advanced Robotics: Redundancy and Optimization*. Addison-Wesley, 1991.

[23] P. Piela, T. Epperly, K. Westerberg, and A. Westerberg. Ascend: An object-oriented computer environment for modeling and analysis. part 1 - the modeling language. Technical Report EDRC 06–88–90, Engineering Design Research Center, Carnegie Mellon University, 1990.

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

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

[26] Steven Sistare. Interaction techniques in constraint-based geometric modeling. In *Proceedings Graphics Interface '91*, pages 85–92, June 1991.

[27] 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.

[28] 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.

[29] Ivan Sutherland. *Sketchpad: A Man Machine Graphical Communication System*. PhD thesis, Massachusetts Institute of Technology, January 1963.

[30] 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.