DRAWING WITH CONSTRAINTS *

Michael Gleicher
Andrew Witkin
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
USA
{gleicher|witkin}@cs.cmu.edu

November 19, 1993

Abstract

The success of constraint-based approaches to drawing has been limited by difficulty in creating constraints, solving them, and presenting them to users. In this paper, we discuss techniques used in the *Briar* drawing program to address all of these issues. *Briar's* approach separates the problem of initially establishing constraints from that of maintaining them during subsequent editing. We describe how non-constraint-based drawing tools can be augmented to specify constraints in addition to positions. These constraints are then maintained as the user drags the model, allowing the user to explore configurations consistent with the constraints. Visual methods are provided for displaying and editing the constraints.

CR Categories: I.3.6 Interaction techniques.

Additional Keywords: constraints, drawing, direct manipulation, Snap-Dragging.

1. Introduction

Most drawings contain some precise relationships between parts. Because these relationships are so important, almost all drawing programs provide the user with some type of aid in establishing

* Accepted to *The Visual Computer.* This is a draft version. Comments are most welcomed.

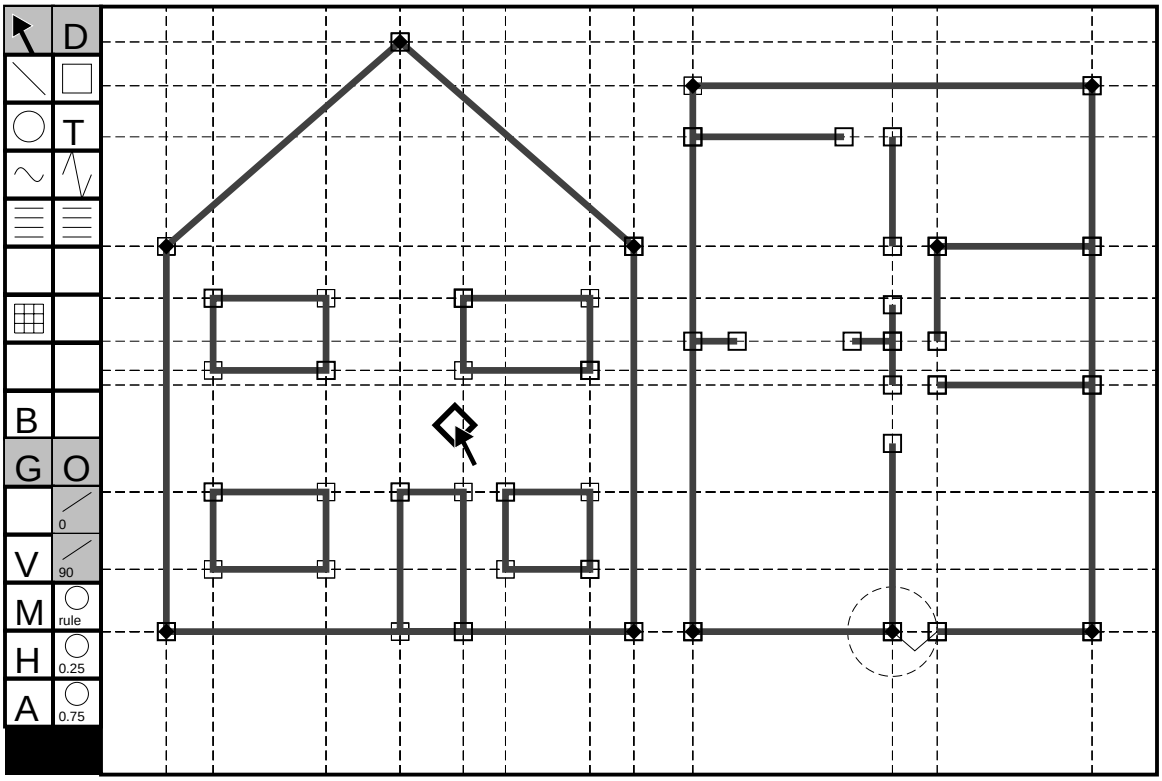

Figure 1: *Briar* editing a constrained drawing.

them. The most common of these aids, such as grids and gravity fields, immediately forget about a relationship after helping the user establish it. Some systems do explicitly represent these relationships as constraints. These constraint-based systems can provide the user with more help with relationships by enforcing them during editing.

Although constraint-based techniques have been used since Sketchpad [Sutherland 1963], the earliest drawing program, they have not been successful in general. Many difficult issues have limited the success of constraint-based systems for drawing. Not only must a system be able to solve constraint satisfaction problems, but it must make it easy for users to specify, debug, and edit constrained models. In this paper, we present an approach to constraint-based systems which addresses all of these issues. We have incorporated the ideas into a drawing program called *Briar,*[1] shown in Figure 1.

Briar separates the task of initially establishing relationships in drawings from that of maintaining them during subsequent editing. It provides Snap-Dragging, a successful non-constraint-based technique introduced in [Bier and Stone 1986], for initially establishing relationships in drawings. By augmenting Snap-Dragging, *Briar* obtains the constraint specification with little or no additional effort from the user. Even with the constraints, the parts of the drawing can be directly manipulated; as the user drags an object, constraint techniques adjust other objects to maintain relationships. A visual representation of the constraints is provided, along with methods for removing constraints.

[1]It is called *Briar* because, like the plant it is named for, things stick together inside it.

Combining Snap-Dragging with constraint techniques significantly changes the nature of the constraints. Unlike most previous constraint-based approaches which rely on solving methods to initially satisfy the relationships, constraint methods in *Briar* are used only to maintain relationships already established using Snap-Dragging. Constraint methods move the drawing through configurations which are consistent with the constraints, rather than moving from configurations inconsistent with the constraints to ones that are. This avoids the difficult problem of solving non-linear equations from arbitrary starting points, and allows the use of differential methods instead. It also allows avoiding many of the other issues which have plagued previous constraint-based systems, such as conflicting constraints and underspecified drawings.

A major goal in the development of *Briar* was to build a system with constraints which provided users with the fluent interface that they have come to expect from direct manipulation drawing programs. The techniques discussed in this paper aim to add at least some of the advantages of constraints without detracting from what has made direct manipulation drawing programs so successful. In short, they aim to make *Briar* a direct manipulation drawing program augmented with constraints, not a drawing program with a constraint-based interface. The interface feels similar to other direct manipulation drawing programs which provide snapping except that once snapped, things can stick together.

1.1. Issues in Constraint-Based Drawing

Previous constraint-based drawing systems have used constraint techniques to initially satisfy geometric relationships in a drawing. We call this a "specify–then–solve" approach, as the user first specifies desired geometric relationships and then the system attempts to solve the constraint equations to establish the relationships.

Establishing a relationship requires the drawing to move from an arbitrary state to one consistent with the constraints. The problem of finding solutions to constraint problems from arbitrary starting points is difficult. In fact, for the general case of non-linear equations, there is no guaranteed way to solve them [Press et al. 1986]. Since traditional constraint-based systems rely on finding solutions from unsolved starting points, they must either place substantial restrictions on the class of constraints they allow, as done by Ideal [Van Wyk 1982] and Overhead [Li 1988], or use temperamental numerical techniques which only work from starting points close to a solution, as done in systems such as relaxation in Sketchpad [Sutherland 1963], Newton-Raphson iteration in Variational Geometry [Lin et al. 1981], Juno[Nelson 1985], and Converge [Sistare 1991], and the non-linear programming methods in Viking [Pugh 1992].

Constraint satisfaction problems are sometimes impossible to solve, if, for example, there are conflicting constraints. Such problems may be caused by complex interactions of many constraints, and therefore can be difficult to diagnose and debug. They can be difficult to detect, as solvers must determine that no solution exists and not just that none have been found yet.

Even if the solver is able to find a solution, it must help the user understand how and why it got to the new state. This task is important and challenging when the solution is not what the user had hoped: either because of a bug in the constrained model, or because the solver has chosen incorrectly in an underconstrained situation.

Underconstrained situations plague constraint-based approaches. Most drawing are not completely specified by their constraints because certain relations do not lend themselves to being expressed with constraints, not all parameters are determined by precise relationships, and the user may not know or care about some aspects of the drawing. Because it is difficult to determine when the scene is exactly and uniquely specified by constraints, forcing a user to create such scenes is liable to lead to redundant and conflicting constraints. Therefore, constraint-based programs must be able to deal with underconstrained cases.

Underconstrained systems pose a problem for algebraic-solving constraint approaches. When the program is searching for a legitimate state from an illegal starting point, the system must guess which of the many possible solutions the user wants. Systems can employ heuristics, such as minimizing distance to the initial configuration, as in [White 1988], minimizing the effort of the solver, as in Magrite [Gosling 1983], or allowing the user to specify optimization criteria, as in Ascend [Piela et al. 1990], but users still must be prepared to deal with incorrect guesses by attempting to either constrain away extra degrees of freedom or try new starting conditions for the solver. Some systems, such as Viking [Pugh 1992], admit the possibility of incorrect guesses and make it easy for the user to request another guess from the system.

The above issues of solving arbitrary satisfaction problems and unsolvable constraints only arise when a constraint-based system moves from an inconsistent state to one where its constraints are met. Furthermore, while helping users understand constrained behavior is a difficult problem, jumping to solved states and guessing which solution to jump to both significantly complicate it. Systems which allow users to specify relationships and then solve for configurations which meet these constraints must address these challenges. However, if we do not use constraint techniques to initially establish relationships, instead only using them to maintain previously established ones, these problems can be avoided.

In *Briar,* we only create constraints for relationships which have already been established in the drawing. This way, *Briar* never requires a constraint solver to find a solution from an arbitrary state. We know that at least the initial solution exists, so that there are no conflicts in the constraints. The drawing never needs to jump between states, so these transitions never need to be explained to the user. Since direct manipulation is available to specify aspects of the drawing which cannot be conveniently described by constraints and to explore configurations consistent with the constraints, underconstrained drawings are an asset, rather than a problem.

2. Augmented Drawing Tools

Briar's approach only uses constraint techniques after relationships are already established. To establish the relationships initially, we use the techniques such as grids, gravity, and Snap-Dragging [Bier and Stone 1986] that have been employed by non-constraint based drawing programs. We must avoid giving the user the extra work of specifying both the constraints and an initial solution. To do this, *Briar* provides *Augmented Snap-Dragging,* a variant of Snap-Dragging which has been extended to specify persistent constraints as well as positions. The basic idea is that cursor placement operations of Snap-Dragging contain information about why an object was positioned where it was, and therefore can also provide a constraint specification in addition to positional

information. The technique can also aid traditional constraint-based drawing programs which require good starting points to prevent long jumps which are hard for both users and solvers.

Ours is not the first attempt to spare users from additional effort required to explicitly specify constraints. Previous systems have attempted to infer relationships after drawing operations by looking at the resulting drawing, as in automatic beautification [Pavlidis and Wyk 1985], sequences of drawings, as in Chimera's snapshot mechanism [Kurlander and Feiner 1993], or at a trace of user actions, as in Metamouse [Maulsby et al. 1989]. Because this information typically does not specify the relationships unambiguously, these systems relied on heuristics or asked the user to resolve the ambiguity, as in Peridot [Myers and Buxton 1986] and Druid [Singh et al. 1990]. Our approach provides positioning methods which unambiguously specify constraints, eliminating the need for inferences. We augment drawing aids to specify constraints as well.

In *Briar*, Augmented Snap-Dragging is the only method for specifying constraints. It provides a uniform method for creating a variety of constraints, such as controlling distances, positions and orientations. Other systems which infer constraints from snapping either have a limited vocabulary of constraints, such as the Manhattan gridding rules of the interface builder of [Hudson and Yeatts 1991], or use other methods to specify the complete set of constraints, as in Intellidraw [Aldus 1992] and DesignView [Computervision 1992]. Rockit [Karsenty et al. 1992] also infers gridding constraints from drawing actions, but does not avoid ambiguity, actually averaging multiple possibilities. Chimera [Kurlander 1993] also has both Snap-Dragging and constraints.

When drawing, it is difficult to position a pencil precisely without using some form of aid. Similarly, it is difficult to draw precisely with a mouse or other pointing device unaided. Computer software can provide tools for precise placement by drawing from a software-positioned cursor[2] rather than using the pointing device location. The software cursor's location is influenced by the position of the pointing device, but determined by a function which helps the user position elements precisely.

The uniform grid is the most common function for mapping pointer location to cursor position. It displaces the cursor to points on an equally spaced rectangular grid. "Gravity" is another cursor positioning function. When the pointer is brought sufficiently close to an interesting element in the scene, the cursor snaps to it. The idea of gravity has existed for a long time, having been demonstrated as early as Sketchpad [Sutherland 1963]. Snap-Dragging [Bier 1989, Bier and Stone 1986] enhances the usefulness of gravity. The cursor snaps not only to the edges of objects, but also to interesting points in the scene such as intersections and vertices of objects. The ability to snap to intersections enables the use of traditional drafting compass-and-straight edge constructions.

Cursor placement operations contain information about why an object was positioned where it was, and therefore can also provide a constraint specification. Suppose the user, while dragging an object, moves the pointer near another object so that the cursor, and the point being dragged, snap to the second object. This might have been an accident, but the user might have been trying to

[2]This differs from the original Snap-Dragging terminology [Bier 1989] where the position of the hardware pointing device is known as the cursor and the software cursor is known as the caret.

achieve this relationship. We provide the user with the option of making the relationship persistent, so if it was intentional it can be preserved during subsequent editing. We call the extension of snapping to specify the relationship in addition to a position *augmented snapping.*

When a new relationship is established by snapping, the system acknowledges it by displaying a symbol indicating the constraints which the snapping operation implies. The user can accept the new constraint, by pressing a key, to make it persistent or ignore it. If this automatic constraint generation process works well, the user will want to accept most constraints so the option of making this the default should be provided.[3] In such a mode, it must be easy for the user to reject an action was an accident. In *Briar,* a key is used to toggle new relationships between the accepted state, in which they are made into persistent constraints, and the ignore state, in which the symbols are removed when the next drawing operation begins.

Snap-Dragging provides two basic operations for positioning points in two dimensions: snapping the cursor to a point, such as a vertex, and snapping the cursor to an object's edge or curve. These operations correspond directly to the constraints "points–coincident" and "point–on–object" respectively.

Relations other than contact are created in Snap-Dragging through *alignment objects:* objects that are not part of the drawing *per se,* but exist only to be snapped to. The original Snap-Dragging work includes several types of alignment objects, each corresponding to types of relationships which are useful in drawings. For example, distance from a point can be specified by placing an alignment circle around that point. The usefulness of alignment objects is further enhanced by making them easy to place. In fact, they can often be placed where needed automatically.

The two simple snapping operations combined with alignment objects allow a user to establish a wide variety of relationships. The simple mapping from snaps to constraints extends to a variety of constraints. For instance, snapping to an intersection is a conjunction of the simpler constraints. By using the simple constraints with alignment objects, the relationships specified by snapping to these objects can be made persistent. For example distance-from-point constraints are created by snapping to an alignment circle. The Gargoyle editor [Bier 1989] shows how Snap-Dragging can be used to create most of the relationships which are needed in drawings. Augmented Snap-Dragging extends this to inferring a similarly complete set of constraints.

Hidden state is a fundamental problem in the interface between man and machine; therefore, feedback is an important aspect of any user interface [Norman 1990]. To make Augmented Snap-Dragging work, feedback is crucial. Our feedback mechanisms (Figure 2) ensure that the snapping state is not hidden from the user, by highlighting the object snapped to and changing the cursor's shape to show if it is snapped, and whether or not it is snapped to a point or an edge. Good feedback makes snapping easier to use because the user never needs to guess what relationships the system is establishing, or which relationships can be made persistent when the snapping operation is complete. The other benefits of snapping feedback described in [Hudson 1990] also apply as it shows the user what can be snapped to, not just what is snapped to.

There are many advantages to augmented snapping as a front end to a constraint system.

[3] In our experience, the automatic constraint generation is so good that we make it the default.

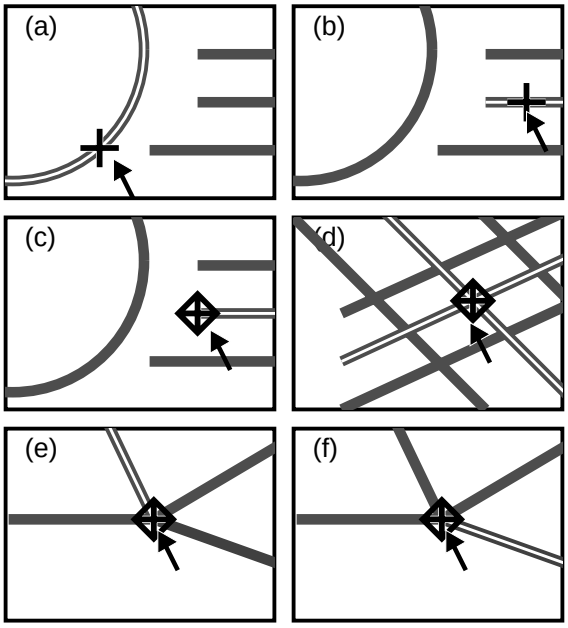

Figure 2: Feedback mechanisms display exactly what is snapped to. The cursor changes shape depending on whether it is snapped to a curve or edge (a,b), or a point such as an endpoint or intersection (c,d). The object snapped to is brightly lit. If several objects are close together, the one desired can be selected by cycling (e,f). Color is used for feedback when available.

Augmented snapping is opportunistic, creating constraints where it can. Constraints are specified with little additional user effort beyond what is required to initially establish them. The constraint creation process can be quite transparent so it does not interfere with the user's drawing process. Augmented snapping still provides the user with the snapping interface which has proved so successful constraints need not be used. *Briar* augments Snap-Dragging, which by itself is an extremely powerful drawing tool.

2.1. Removing Ambiguity

If multiple objects coincide when snapping, it might not be clear which object to snap to. If we are only using snapping for positioning, the ambiguity is irrelevant: only the target location is important. With persistent relations this distinction becomes significant. If the two coincident points are later separated, the correct attachment relationship must be maintained.

Feedback, along with Snap-Dragging's cycling mechanism, solves the problem of ambiguity. Feedback mechanisms clearly show the user which object is being snapped to and what relationships are being established. If these are incorrect, the user can click the cycle button and the system will snap to the next object within gravity range.

A related problem is that the user might construct a model in a manner which does not convey the desired constraints. As an example, consider the equilateral triangle construction (Figure 3) which is used to demonstrate Snap-Dragging in [Bier and Stone 1986]. In this example, alignment circles of $3/4$ inch are used to create an equilateral triangle. The user has specified a triangle

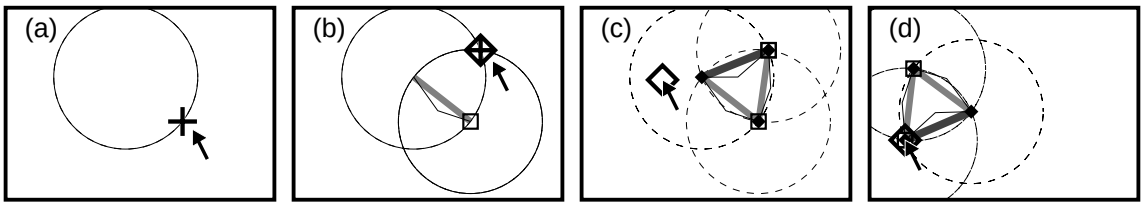

Figure 3: Alignment circles help create an equilateral triangle. Briar places 3/4 inch alignment circles around points of interest. Snapping to the circle created with the initial mouse point sets the length of the first line (A). Snapping to the intersection of the circles created around the ends of the newly created line segment constrains the point to be 3/4 inches from each endpoint (b). Snapping the final segment in place leaves three lines constrained to be connected with their endpoints 3/4 inches apart (c). The system can maintain these constraints as as the user drags pieces of the triangle (d).

with all sides equal to $3/4$ inch, not a triangle whose equal sides can be scaled as well as rotated and translated. The program only knows what the user has specified and, therefore, cannot guess another option.

Briar does not attempt to guess the user's intent. Instead, it tries to keep the automatic generation process predictable, to use feedback to remind the user exactly what the system has been told, and to provide tools which convey the desired relations directly. Since our goal is to extract constraints without extra work, it is wrong to require users to expend extra effort to use constructions which create the correct constraints. Therefore, we must make it as easy as possible for the user to convey what is really intended.

The scalable equilateral triangle problem is addressed using a mechanism from Snap-Dragging. Rather than specifying that the alignment circles, and therefore the resulting triangle, have size $3/4$ inch, a measurement tool is employed. The first line segment is drawn and then measured. Circles are then created with radius equal to the length of the line segment. In *Briar,* , this construction requires only one more mouse click than to construct the fixed sized triangle.

As we find relationships that are needed in drawings but are difficult to express directly with current tools, we can develop new tools to expand the set that is easily specified. Expanding the vocabulary by adding a wider assortment of tools only makes modeling easier to a point, after which the larger number of tools becomes unmanageable. User experience with Gargoyle, as described in [Bier 1989], shows that the small of Snap-Dragging tools are sufficient to create a wide array of drawings.

Another complication in augmenting Snap-Dragging is that if the constraints already imply that two objects are related, there is no reason to snap them together. Doing so can force the user to cycle or can create a redundant constraint. For example, if the user is dragging a point that is already connected to a second point, there is no reason to snap to the second point. Filtering the set of objects that can be snapped to eliminates this nuisance.

However, recognizing that a relationship is implied by a set of geometric constraints can require difficult geometric proofs as some complicated mechanism might imply additional relationships. From our experience, it appears that handling the simple cases—not snapping to the object being

dragged, checking for existing connection constraints, applying basic geometric identities, etc.—filters out the vast majority of redundant snaps. Avoiding redundant snaps is an optimization, it is not critical to the functioning of snapping. However, it does make it the case that the constraint techniques are robust in handling redundant constraints.

3. Maintaining Constraints

Inside a constraint-based system, geometric relationships are represented as mathematical equations. The problem of finding a configuration which meets a set of constraints therefore requires solving a system of equations. Since geometric relationships often involve non-linear equations, the systems of equations are non-linear. The general problem of solving arbitrary systems of non-linear equations is extremely difficult. In fact, there are arguments that no good general techniques for the problem can exist [Press et al. 1986].

When initial solutions are provided, the task of constraint techniques changes; instead of establishing the relationships, constraint-based techniques are used to maintain them. The constraint solver must keep the configuration of the drawing consistent with the constraints as the drawing is edited.

In order to provide direct manipulation of drawings, the constraints must be solved continuously during dragging. As one part is moved, others must be adjusted to keep the constraints satisfied. Fast computers and good algorithms allow update rates which give the appearance of continuous motion. This rapid feedback is essential. Although the trajectory the model follows is not part of the resulting drawing, this animation makes it possible for users to employ their perceptual skills to connect states of the drawing with many things changing between them [Baecker and Small 1990, Robertson, Mackinlay and Card 1991].

With constrained direct manipulation, objects are dragged the same way as with standard direct manipulation, except that relationships can be maintained among them. We choose to force dragging to be subjected to the persistent constraints, that is that direct manipulation should not break a persistent constraint. This allows the drawing process to be incremental: each new relationship added to a drawing does not disturb previously established ones. The existence of a direct manipulation facility means that all parts of the model do not need to be specified by constraints. If it is difficult to devise a way to describe an aspect of a drawing with constraints, direct manipulation can be used instead.

3.1. Differential Methods

The solving techniques employed by a system should be of no concern to the user, provided that they are sufficiently fast, general, and stable. Solving can be accomplished using a standard constraint-solving approach: the model is repeatedly perturbed slightly, then resolved. However, since the solver is only used to maintain constraints, rather than to establish them, we have the opportunity to use techniques designed for the more specific task. Such solvers can better handle the non-linear constraints and provide better performance. We have used differential constraint

methods which are such a technique.

Differential constraint methods are detailed in [Gleicher and Witkin 1991b] and [Gleicher and Witkin 1992], we will briefly review them here. Direct manipulation requires objects to move with smooth motion. Differential methods take advantage of this. Rather than attempting to directly control the configurations of the objects, the methods operate by controlling how the objects move, specifying the time derivatives of their configuration.

Given a desired motion, for example that a point tracks the mouse, and a set of constraints whose values are to be maintained, differential methods solve for the time derivatives of the configuration that achieve these. Non-linear constraints on the configuration are enforced by maintaining linear equations that specify the time derivatives of the constraints. Differential methods repeatedly solve these systems of linear equations to compute the motion of the objects, and update the configurations of the objects accordingly.

The linear equations that determine the time derivatives of the configuration will typically not uniquely determine a solution. When the constraints do not fully specify the configuration, the linear equations that maintain them do not uniquely determine the time derivative of the configuration. To select among the possible ways for the configuration to change, differential methods minimize a quadratic function of the derivative. The quadratic objective function used in *Briar* is to minimize the magnitude of the derivative of the configuration; that is, to change the configuration as little as possible in order to maintain the constraints.

To compute the time derivative of the configuration, the differential method must solve an optimization problem with a quadratic objective function subject to a set of linear constraints. This is a well studied class of problems, for which many good methods have been developed (see [Fletcher 1987] or [Gill et al. 1981] for a survey). In *Briar,* a conjugate-gradient method, adapted from the one presented in [Press et al. 1986], is used. Issues in setting up the efficiently setting up and solving these optimizations are discussed in [Gleicher and Witkin 1993].

3.2. Using Constrained Dragging

Dragging parts of drawings allows the user to experiment with the constrained model. This interactive animation is a useful tool for understanding and debugging constraints. It also opens up the possibility of *dynamic drawings:* models which are created to be dragged and played with. Such drawings contain unconstrained degrees of freedom so they have behavior when their parts are dragged. For example, the engine in Figure 4 “works” – a user can pull on parts and operate the engine. The use of constraint-based approaches to animate drawings dates back to Sketchpad [Sutherland 1963] and approaches for interactive mechanism simulation, such as [Kramer 1990], [Rubel and Kaufman 1977], and [Enderton 1990], have been developed. However, differential constraint systems are well suited for such mechanical simulation and animation tasks as well as interaction.

The facility with which we handle constraints opens the possibility of using constraints to aid in manipulation. Most constraints are used to represent structure in the drawing. However, temporary constraints which are easy to create and destroy are useful to make drawings easier to control. We

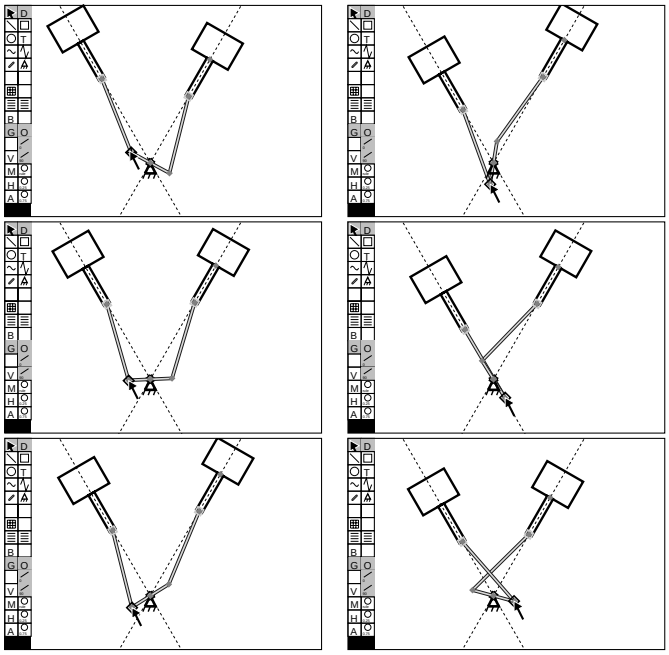

Figure 4: A constrained model of an engine. As the model is dragged, the constraints are enforced, allowing the user to experiment with the kinematic behavior of the object.

call such constraints, which are meant to be short-lived, "lightweight" constraints.

Dragging is one example of a lightweight constraint. It is achieved by temporarily constraining the point being dragged to follow the mouse. Another useful lightweight constraint is the *tack.* Tacks hold a particular point in place. They act as an extra hand, making it easy to stretch or rotate an object. Nailing a point at a particular point in space is a common facility in constraint-based systems. Making it easy to place and remove the nails easily enables new uses for them. For example, tacks can perform the tasks that anchors do in traditional Snap-Dragging [Bier 1989], specifying the center of rotation and scaling.

4. Displaying And Editing Constraints

A constrained drawing has more state that must be displayed to the user than a non-constrained one does. A system must convey to the user not only the geometry of the model, but also the constraints. The user must be able to edit this structural information as well as the geometry. Previous constraint-based systems have used three types of techniques for displaying constraints to users: textual languages, diagrammatic representations, and graphical cues drawn directly on the model.

Textual languages for describing constraints, such as that employed in Juno [Nelson 1985], have the advantage that they are editable. Unfortunately, they are distinct from the drawing and can be difficult to connect. Schematic representations of constraints, such as that presented in [Borning 1986], are similar in that the constraint display is separate from the drawing.

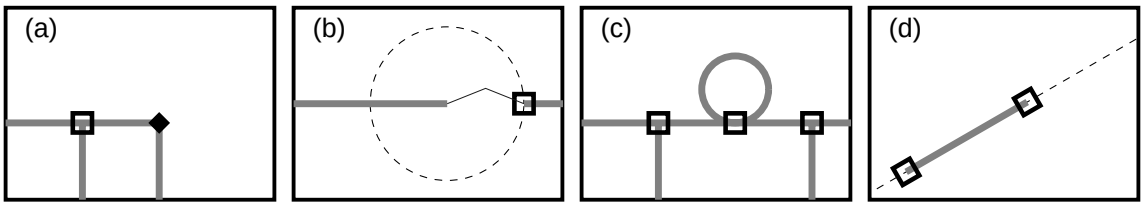

Figure 5: Snap-Dragging provides a basis for visual representation of constraints. The two basic constraints, point-on-object and point-to-point, are drawn as an empty square and a filled diamond (a) respectively. Other constraints are represented using the basic ones and alignment objects, for example, distance-between points (b), points-aligned (c), and orientation (d). Thin wedges as seen in (b), emphasize distance constraints. Color is used for constraint display when available.

Visual representations, such as in Converge [Sistare 1991], CoDraw [Gross 1989], and Viking [Pugh 1992], superimpose constraints directly on the drawing. The connection between the relationships that will be maintained and the objects they affect is shown to the user. This is particularly dramatic when the model is moving subject to the constraints, such as when it is being dragged.

Constraint representations which are superimposed on drawings do have drawbacks. Each type of constraint must be displayable in a manner which makes clear both what the constraint is and what it effects. This can be particularly difficult in systems with a large palette of constraints. Visual representations can also be difficult to edit. The tendency of constraints to cluster is one source of this difficulty.

4.1. A Visual Representation of Constraints

Augmented Snap-Dragging provides a graphical method for specifying constraints. One of its features is that it provides a uniform method for describing a wide variety of relationships. It can also be used as a representation for displaying constraints, providing an equally uniform visual language for displaying constraints that is the same as that used to specify them.

Augmented snap-dragging specifies all relationships by one of two basic types of snapping operations. By graphically depicting these two types of snaps, they become a way to depict the constraints they create. Alignment objects are also part of relationships and therefore they must also be made persistent, unlike in traditional snap-dragging. The snapping symbols and alignment objects provide a graphical representation for a wide variety of constraints. Some examples are shown in Figure 5.

One factor which complicates visual display and editing of constraints is the tendency of constraints to cluster, often being in exactly the same place. The semantics of constraints can be tuned to make this problem easier. For example, rather than using binary relations to connect points to each other, linked points are placed into an equivalence class. If a set of points has been made equivalent, just the equivalence class needs to be shown, not the potentially large number of equality relationships all piled on top of each other.

4.2. Editing Constraints

Ease of editing constraints is important; relations should be as easy to break as to make. Users may change their minds as to what relationships should be in the model, or may simply have made a mistake in specifying the constraints. Using a visual representation creates only part of the difficulty in deleting constraints: before being able to point at a constraint, the user must know which constraint or constraints to delete. To address this, we use methods for deleting constraints which allow users to remove constraints by referring to the desired effects, not to the constraints themselves. In fact, *Briar* provides no mechanisms for users to point directly to constraints.

A direct method of removing constraints by referring to the objects they influence is to "rip" them. When an object is grabbed for dragging, holding down a modifier key causes the grabbing operation to "grab hard," removing any constraints on the point grabbed. This method is often undesirable as it may remove too many constraints.

Another technique for deleting constraints without pointing at them is to allow the user to temporarily disable constraint maintenance. In this mode, the user can manipulate the objects as if they did not have any constraints on them. When constraint enforcement is re-enabled, constraints which the user has broken are discarded. Feedback as to which constraints are broken and the ability to use snapping operations to reassemble things help make this a useful technique for editing the constraints. A similar mechanism for destroying unwanted constraints is provided by [Kurlander and Feiner 1993], where the user can create a new "snapshot" of the drawing which shows the constraint broken

5. Briar's Implementation

The *Briar* drawing program serves as a testbed for the ideas and techniques discussed in this paper. *Briar* is written in C++ and runs on Silicon Graphics Iris workstations. It is built on top of an early version of our general purpose mathematical toolbox, described in [Gleicher and Witkin 1993]. *Briar* was developed in the autumn and winter of 1990, and was first reported in a technical report [Gleicher and Witkin 1991a].

The constraint techniques used in *Briar* were described in Section 3.1. *Briar* exploits sparsity in its algorithms to provide needed performance and better scalability. Because of the dynamic nature of drawings, pre-analysis techniques, such as those discussed in [Surles 1992], are not appropriate. To further enhance performance and scalability, *Briar* also employs partitioning techniques to automatically disable parts of the drawing which cannot change.

In addition to augmented snap-dragging, differential constraint maintenance, and graphical display and editing of constraints, *Briar* has many standard drawing program features, such as grids, hierarchical grouping, and the ability to make pictures for our text formatter. Special features for sketching and experimenting with planar linkages are also provided, as seen in Figure 4.

5.1. Limitations

Experimenting with *Briar* has afforded an opportunity to explore the limits of the approach described in the paper. These limitations fall into three categories: artifacts of the *Briar* prototype, limitations in the approach, and general issues in constraint-based approaches.

Briar was developed as a research prototype, and therefore lacks many of the features required of real drawing programs, such as good text handling and free-form curves. Similarly, *Briar's* interface is not as polished as those of a commercial system. This makes comparisons difficult, but also leaves open the question as to whether *Briar's* commands can be presented to the user in a user-friendly fashion. Drawings are limited are limited in color and linestyle, so constraint display is easily distinguishable from the drawing.

Only a subset of Gargoyle's alignment objects, as described in [Bier and Stone 1986], are implemented in *Briar.*This makes it impossible to express some constraints. For example, to make two line segments parallel a complete Snap-Dragging implementation would provide distance lines, which are not implemented in *Briar* . Users' experience with Gargoyle, discussed in [Bier 1989], seems to indicate that the tools provided Snap-Dragging are sufficient for creating the relationships needed in drawings. There are some potential issues in implementing the complete set in *Briar* , for example, some form of feedback to distinguish the different types of alignment lines might be necessary when they are made persistent.

As in any constraint-based system, scalability is a major concern with the techniques described in this paper. Our reliance on sufficient performance to create continuous motion can become a limitation when models become larger. However, in light of ever increasing processor and drawing performance, cognitive complexity may prove to be the limiting factor in constraint-based systems. As drawings become larger and more complex, constrained behavior becomes even more difficult to understand. Attempts to use feedback to alleviate this complexity also face an additional challenge as the display can become cluttered. If the interface and performance issues can be addressed, constraints can offer a significant advantage for large drawings. Effort saved by not having to manually re-establish the geometric relations in a drawing after an editing operation can become significant when the number of relationships grows large.

Another scalability issue in *Briar* is clutter. As the drawing becomes more complicated, there will be more elements on the display. This can make snapping less convenient, as the may have to continually cycle to select among snap targets which are close to one another, and constraint display less informative, as objects obscure each another. Such problems can be partially combatted by selective display, and better use of alternate visual cues such as dimming.

6. Conclusions

In this paper, we have discussed some of the techniques used in the *Briar* drawing program and how these techniques help address the issues in employing constraints in drawing programs. A visual record of *Briar* can be found in our video[Gleicher 1992].

The techniques described in this paper extend beyond the task of two dimensional drawing. One interesting domain to which they apply is three dimensional modeling and manipulation. Both Snap-Dragging [Bier 1990] and constraints have been demonstrated in 3D. However, the richer set of relationships, the greater challenges in creating non-intrusive feedback, and the increased model complexities are some of the more difficult issues in realizing interactive 3D constraint systems.

Presently, we are encapsulating the techniques in this paper in a general purpose toolkit for building two and three dimensional interactive graphical applications [Gleicher 1993]. Such an approach permits exploration of the techniques independently of the application and testing techniques in a variety of settings. It also makes significantly easier to build special purpose applications which have a complete set of interface functionality.

Building a constraint-based drawing program is difficult because of the large number of issues which must be addressed. The approach presented in this paper and implemented in *Briar* addresses many of them. Separating the tasks of establishing and maintaining constraints allows us to use non-constraint-based techniques to avoid many potential pitfalls. Augmented drawing tools make it easy for users to specify both constraints and initial configurations which meet them. The use of dragging permits users to explore configurations of the model. Visual methods allow users to see and edit the constraints. We are working on exploring the application of these ideas in both two dimensional and three dimensional applications.

Acknowledgments

This research was funded in part by Apple Computer, a fellowship from the Schlumberger Foundation, and an equipment grant from Silicon Graphics. We would also like to thank our colleagues who have commented on various versions of this paper.

References

[Aldus 1992] Aldus Corporation (1992) Intellidraw. Computer Program.

[Baecker and Small 1990] Baecker R, Small I (1990) Animation at the interface. In: Laurel B, editor, *The Art of Human Computer Interface Design*. Addison-Wesly, pp. 251–268.

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

[Bier 1989] Bier E (1989) Snap-dragging: Interactive geometric design in two and three dimensions. Technical Report EDL-89-2, Xerox Palo Alto Research Center.

[Bier 1990] Bier E (1990) Snap-dragging in three dimensions. *Computer Graphics*, 24(2):193–204. Proceedings 1990 Symposium on Interactive 3D Graphics.

[Borning 1986] Borning A (1986) Defining constraints graphically. In: *Proceedings CHI 86*, pp. 137–143.

[Computervision 1992] Computervision Corporation (1992) DesignView. Computer Program.

[Enderton 1990] Enderton E (1990) Interactive type synthesis of mechanisms. Master's thesis, University of California, Berkeley. Also appears as Report No. UCB/CSD 90/570.

[Fletcher 1987] Fletcher R (1987) *Practical Methods of Optimization*. John Wiley and Sons.

[Gill et al. 1981] Gill P, Murray W, Wright M (1981) *Practical Optimization*. Academic Press, New York, NY.

[Gleicher and Witkin 1991a] Gleicher M, Witkin A (1991) Creating and manipulating constrained models. Technical Report CMU-CS-91-125, School of Computer Science, Carnegie Mellon University.

[Gleicher and Witkin 1991b] Gleicher M, Witkin A (1991) Differential manipulation. *Graphics Interface*, pp. 61–67.

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

[Gleicher and Witkin 1993] Gleicher M, Witkin A (1993) Supporting numerical computations in interactive contexts. In: Calvert T, editor, *Graphics Interface*, pp. 138–145.

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

[Gleicher 1993] Gleicher M (1993) A graphics toolkit based on differential constraints. In: Pausch R, editor, *Proceedings UIST '93*.

[Gosling 1983] Gosling J (1983) *Algebraic Constraints*. PhD thesis, Carnegie Mellon University.

[Gross 1989] Gross M (1989) Relational modeling: A basis for computer-assisted design. In: McCullough M, Mitchell WJ, Purcell P, editors, *The Electronic Design Studio (Proc. CAAD Futrues '89)*. MIT Press, pp. 123–146.

[Hudson and Yeatts 1991] Hudson SE, Yeatts AK (1991) Smoothly integrating rule-based techniques into a direct manipulation user interface builder. In: *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pp. 145–153.

[Hudson 1990] Hudson SE (1990) Adaptive semantic snapping – a technique for semantic feedback at the lexical level. In: *Proceedings CHI '90*, pp. 65–70.

[Karsenty et al. 1992] Karsenty S, Landay JA, Weikart C (1992) Inferring graphical constraints with Rockit. In: *HCI'92 Conference on People and Computers VII*. British Computer Society, pp. 137–153.

[Kramer 1990] Kramer GA (1990) Solving geometric constraint systems. In: *Proceedings AAAI-90*, pp. 708–714.

[Kurlander and Feiner 1993] Kurlander D, Feiner S (1993) Inferring constraints from multiple snapshots. *ACM Transactions on Computer Graphics*, 12(4).

[Kurlander 1993] Kurlander D (1993) *Graphical Editing by Example*. PhD thesis, Columbia University.

[Li 1988] Li J (1988) Using algebraic constraints in interactive text and graphics editing. In: Duce PA, Jancene P, editors, *Proceedings Eurographics*.

[Lin et al. 1981] Lin VC, Gossard DC, Light RA (1981) Variational geometry in C.A.D. *Computer Graphics*, 15(3):171–177. Proceedings SIGGRAPH '81.

[Maulsby et al. 1989] Maulsby DL, Kittlinz KA, Witten IH (1989) Metamouse: Specifying graphical procedures by example. *Computer Graphics*, 23(3):127–136. Proceedings SIGGRAPH '89.

[Myers and Buxton 1986] Myers BA, Buxton W (1986) Creating highly-interactive and graphical user interfaces by demonstration. *Computer Graphics*, 20(4):249–258. Proceedings SIGGRAPH '86.

[Nelson 1985] Nelson G (1985) Juno, a constraint based graphics system. *Computer Graphics*, 19(3):235–243. Proceedings SIGGRAPH '85.

[Norman 1990] Norman D (1990) *The Design of Everyday Things*. Doubleday.

[Pavlidis and Wyk 1985] Pavlidis T, Wyk CV (1985) An automatic beautifier for drawings and illustrations. *Computer Graphics*, 19(3):225–234. Proceedings SIGGRAPH '85.

[Piela et al. 1990] Piela P, Epperly T, Westerberg K, Westerberg A (1990) 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.

[Press et al. 1986] Press W, Flannery B, Teukolsky S, Vetterling W (1986) *Numerical Recipes in C*. Cambridge University Press, Cambridge, England.

[Pugh 1992] Pugh D (1992) Designing solid objects with interactive sketch interpretation. In: *Proceedings of the 1992 Symposium on Interactive Computer Graphics*, pp. 117–126.

[Robertson, Mackinlay and Card 1991] Robertson G, Mackinlay J, Card S (1991) Cone trees: Animated 3d visualizations of hierarchical information. In: *Proceedings CHI '91*, pp. 189–194.

[Rubel and Kaufman 1977] Rubel AJ, Kaufman RE (1977) Kinsyn iii: A new human-engineered systems for interactive computer aided design of planar linkages. *Transactions of the ASME: Journal of Engineeringf for Industry*, pp. 440–448.

[Singh et al. 1990] Singh G, Kok CH, Ngan TY (1990) Druid: A system for demonstrational rapid user interface development. In: *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 167–177.

[Sistare 1991] Sistare S (1991) Interaction techniques in constraint-based geometric modeling. In: *Proceedings Graphics Interface '91*, pp. 85–92.

[Surles 1992] Surles MC (1992) An algorithm for linear complexity for interactive, physically-based modelling of large proteins. *Computer Graphics*, 26(2):221–230. Proceedings SIGGRAPH '92.

[Sutherland 1963] Sutherland I (1963) *Sketchpad: A Man Machine Graphical Communication System*. PhD thesis, Massachusetts Institute of Technology.

[Van Wyk 1982] Van Wyk CJ (1982) A high level language for specifying pictures. *ACM Transactions on Graphics*, 1(2):163–182.

[White 1988] White RM (1988) Applying direct manipulation to geometric construction systems. In: Magnenant-Thalmann M, Thalmann D, editors, *New Trends in Computer Graphics: Proceedings of CG International '88*. Springer-Verlag.