*The philosopher's have, in many ways, tried to interpret the World. The point, however, is to change it.*

— Karl Marx
11th thesis on Feurbach

# Chapter 9

# Example Applications

This chapter describes some sample applications built with the differential approach and the tools developed in the previous chapters. The purpose of discussing these applications is threefold:

- to show how the interaction techniques developed with the differential approach can be fit in the context of a realistic application.
- to show that the differential approach is viable.
- to show some of the range of the tools, particularly the Bramble toolkit, giving evidence of how the differential approach can be used to make such tools more general.

All of the example applications discussed, except for the Briar drawing program, were constructed using the Bramble toolkit.

## 9.1 A Drawing Program

Although constraint-based techniques have been used since Sketchpad [Sut63], the earliest drawing program, they have not been generally successful. Many difficult issues have limited the success of constraint-based systems for drawing. Not only must a system be able to solve constraint satisfaction problems, but it must make it easy for users to specify, debug, and edit constrained models. The *Briar*[1] drawing program attempts to address all of these issues. The issues of constraint-based drawing that Briar addresses apply more generally to interfaces built with the differential approach. Briar is discussed in detail in [GW94] and [Gle92a], and illustrated in Figure 9.1.

A major goal in the development of Briar was to build a system with constraints that provided users with the fluent interface that they have come to expect from direct

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

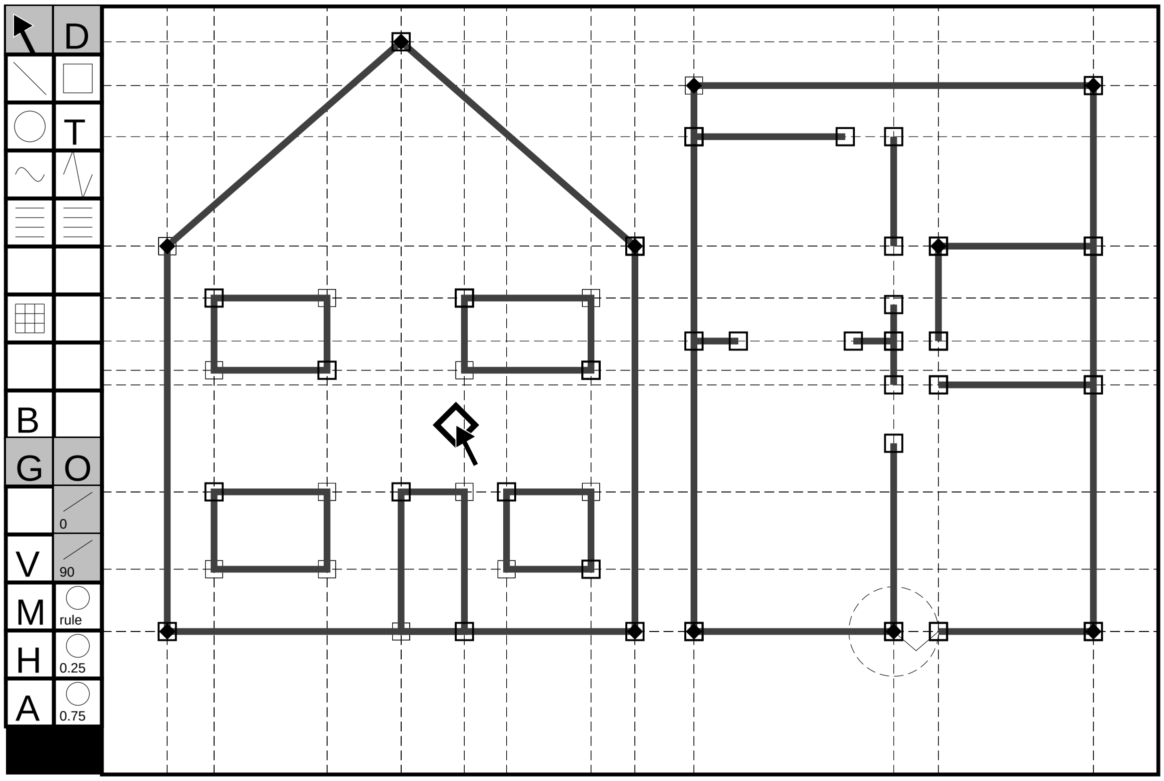


**Figure 9.1:** Briar editing a constrained drawing.

manipulation drawing programs. The techniques in Briar aim to add at least some of the advantages of constraints without detracting from what has made direct manipulation drawing programs so successful. In short, they aim to make Briar a direct manipulation drawing program augmented with constraints, not a drawing program with a primarily constraint-based interface. The interface feels similar to other direct manipulation drawing programs which provide snapping except that once snapped, things can stick together.

The basic idea behind Briar is to enhance an existing, successful, direct-manipulation drawing technique with constraints. Briar separates the task of initially establishing relationships in drawings from that of maintaining them during subsequent editing. It uses snap-dragging, a successful non-constraint-based technique introduced in Gargoyle [BS86], for initially establishing relationships in drawings. By augmenting snap-dragging, Briar obtains the constraint specification with little or no additional effort from the user. The methods of the differential approach allow the constrained drawings to be be manipulated directly; as the user drags an object, constraint techniques adjust other objects to maintain relationships.

Combining snap-dragging with constraint techniques significantly changes the nature of the constraints. Unlike most previous constraint-based approaches that rely on solving methods to initially satisfy the relationships, constraint methods in Briar are used only to maintain relationships during dragging. This permits using the differential approach, as objects move only during continuous-motion dragging. The primary benefits of using constraints only to maintain constraints during dragging parallel those that motivated the differential approach in Section 1.2: we can avoid solving non-linear equations from arbitrary starting points; we need not select configurations in under-constrained cases, and we can aid the user in the problem of understanding state transitions by always providing continuous motion.

In Section 8.2.2 discusses many of the challenges in creating constraint-based drawing editors. The differential approach addresses some of these. Briar includes techniques designed to address many others including how to specify constraints, how to edit them, and how to display them to the user. These issues arise in a wide range of applications. This section describes Briar in detail to illustrate some possible solutions to these problems.

### 9.1.1 Augmented Drawing Tools

Briar's approach only uses constraint techniques after relationships are already established. To establish the relationships initially, we use techniques such as grids, gravity, and snap-dragging [BS86] that have been employed by non-constraint based drawing programs. To avoid giving the user the extra work of specifying both the constraints and an initial solution, Briar provides *augmented snap-dragging*. Augmented snap-dragging is a variant of snap-dragging that has been extended to specify persistent constraints as well as positions. The basic idea is that the cursor placement operations of snap-dragging contain information about why an object was positioned where it was, and therefore they can also provide a constraint specification in addition to positional information. The technique can also aid traditional constraint-based drawing programs which require good starting points to prevent long jumps that are hard for solvers to make and users to understand.

Ours is not the first attempt to spare users from additional effort required to explicitly specify constraints. Previous systems have attempted to infer relationships after drawing operations by looking at the resulting drawing, as in automatic beautification [PW85], sequences of drawings, as in Chimera's snapshot mechanism [KF93], or at a trace of user actions, as in Metamouse [MKW89]. Because this information typically does not specify the relationships unambiguously, these systems relied on heuristics or asked the user to resolve the ambiguity, as in Peridot [MB86] and Druid [SKN90]. Our approach provides positioning methods which unambiguously specify constraints, eliminating the need for inferences. We simply augment drawing tools to specify constraints as well.

In Briar, augmented snap-dragging is the only method for specifying constraints. It provides a uniform method for creating a variety of constraints, such as controlling distances, positions and orientations. Other systems which infer constraints from snapping either have a limited vocabulary of constraints, such as the Manhattan gridding rules of the interface builder of [HY91], or use other methods to specify the complete set of constraints, as in Intellidraw [Ald92] and DesignView [Com92]. Rockit [KLW92] also infers gridding constraints from drawing actions, but does not avoid ambiguity, actually averaging multiple possibilities. Chimera [Kur93] has both snap-dragging and constraints, but does not integrate the two.

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

The uniform grid is the most common function for mapping pointer location to cursor position. It displaces the cursor to points on an equally spaced rectangular grid. "Gravity" is another cursor positioning function. When the pointer is brought sufficiently close to an interesting element in the scene, the cursor snaps to it. The idea of gravity has existed for a long time, having been demonstrated as early as Sketchpad [Sut63]. snap-dragging [Bie89, BS86] enhances the usefulness of gravity. The cursor snaps not only to the edges of objects, but also to interesting points in the scene such

[2]This differs from the original snap-dragging terminology [Bie89] where the position of the hardware pointing device is known as the cursor and the software cursor is known as the caret.

as intersections and vertices of objects. The ability to snap to intersections enables the use of traditional drafting compass-and-straight edge constructions.

Since cursor placement operations contain information about why an object was positioned where it was, they can also provide a constraint specification. Suppose the user, while dragging an object, moves the pointer near another object so that the cursor, and the point being dragged, snap to the second object. This may have been an accident, but the user might have been trying to achieve this relationship. We provide the user with the option of making the relationship persistent, so if it was intentional it can be preserved during subsequent editing. We call the extension of snapping to specify a relationship in addition to a position *augmented snapping.*

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

Snap-Dragging provides two basic operations for positioning points in two dimensions: snapping the cursor to a point, such as a vertex, and snapping the cursor to an object's edge or curve. These operations correspond directly to the constraints "points–coincident" and "point–on–object" respectively. The two constraint types work with each object type that Briar supports. The set of objects presently includes lines, circles, ellipses, and rectangles.

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

The two simple snapping operations combined with alignment objects allow a user to establish a wide variety of relationships. The simple mapping from snaps to constraints extends to a variety of constraints. For instance, snapping to an intersection is a conjunction of the simpler constraints. By using the simple constraints with alignment objects, the relationships specified by snapping to these objects can be made persistent. For example distance-from-point constraints are created by snapping to an alignment circle. The Gargoyle editor [Bie89] shows how snap-dragging can be used to create

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

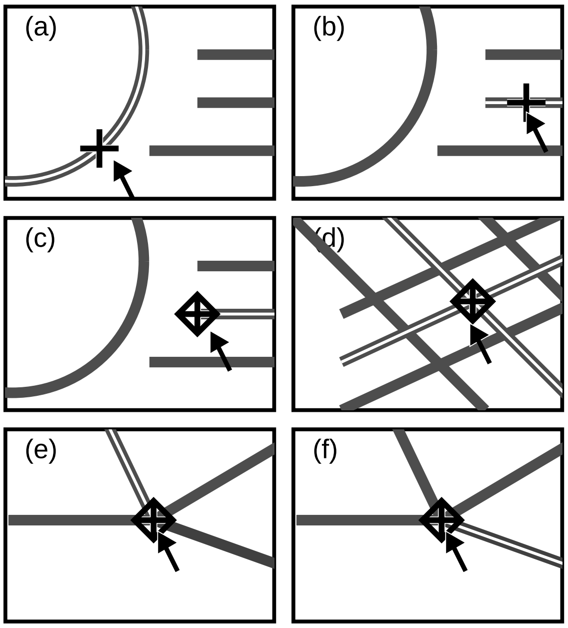


**Figure 9.2:** Feedback mechanisms display exactly what is snapped to. The cursor changes shape depending on whether it is snapped to a curve or edge (a,b), or a point such as an endpoint or intersection (c,d). The object snapped to is brightly lit. If several objects are close together, the one desired can be selected by cycling (e,f). Color is used for feedback when available.

most of the relationships which are needed in drawings. Augmented snap-dragging extends this to inferring a similarly complete set of constraints.

Hidden state is a fundamental problem in the interface between man and machine; therefore, feedback is an important aspect of any user interface [Nor90]. To make augmented snap-dragging work, feedback is crucial. Our feedback mechanisms (Figure 9.2) ensure that the snapping state is not hidden from the user, by highlighting the object snapped to and changing the cursor's shape to show if it is snapped, and whether it is snapped to a point or an edge. Good feedback makes snapping easier to use because the user never needs to guess what relationships the system is establishing, or which relationships can be made persistent when the snapping operation is complete. The other benefits of snapping feedback [Hud90] also apply as snapping shows the user what can be snapped to, not just what is snapped to.

There are many advantages to augmented snapping as a front-end to a constraint system. Augmented snapping is opportunistic, creating constraints where it can. Constraints are specified with little additional user effort beyond what is required to initially establish them. The constraint creation process is quite transparent so it does not interfere with the user's drawing process. Since snap-dragging is used for all drawing operations, including dragging and creating new objects, constraint creation is always available. Augmented snapping still provides the user with the snapping interface

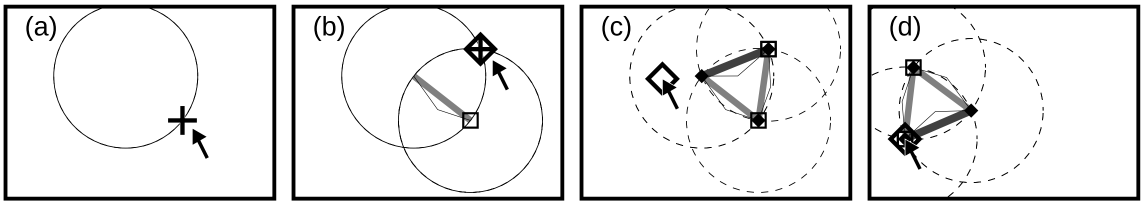


**Figure 9.3:** Alignment circles help create an equilateral triangle. Briar places 3/4 inch alignment circles around points of interest. Snapping to the circle created with the initial mouse point sets the length of the first line (A). Snapping to the intersection of the circles created around the ends of the newly created line segment constrains the point to be 3/4 inches from each endpoint (b). Snapping the final segment in place leaves three lines constrained to be connected with their endpoints 3/4 inches apart (c). The system can maintain these constraints as as the user drags pieces of the triangle (d).

which has proved successful in systems without constraints, so that constraints need not be used if they are not convenient for a particular drawing operation. Briar augments snap-dragging, which by itself is an extremely powerful drawing tool.

### Removing Ambiguity

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

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

A related problem is that the user might construct a model in a manner which does not convey the desired constraints. As an example, consider the equilateral triangle construction (Figure 9.3) which is used to demonstrate snap-dragging [BS86]. In this example, alignment circles of $3/4$ inch are used to create an equilateral triangle. The user has specified a triangle with all sides equal to $3/4$ inch, not a triangle whose equal sides can be scaled as well as rotated and translated. The program only knows what the user has specified and, therefore, cannot guess another option.

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

Therefore, we must make it as easy as possible for the user to convey what is really intended.

The scalable equilateral triangle problem is solved using a mechanism from snap-dragging. Rather than specifying that the alignment circles, and therefore the resulting triangle, have size $3/4$ inch, a length measurement tool is employed. The first line segment is drawn and then measured. Circles are then created with radius equal to the length of the line segment. In Briar, this construction requires only one more mouse click than to construct the fixed sized triangle. Angle and slope measurement tools could also be provided to express other relationships.

As we find relationships that are needed in drawings but are difficult to express directly with current tools, we can develop new tools to expand the set of relationships that are easily specified. Expanding the vocabulary by adding a wider assortment of tools only makes modeling easier to a point, after which the larger number of tools becomes unmanageable. User experience with Gargoyle [Bie89] shows that the standard set of snap-dragging tools are sufficient to create a wide array of drawings.

Another complication in augmenting snap-dragging is that if the constraints already imply that two objects are related, there is no reason to snap them together. Suppose the user is dragging a point. Gravity snaps the cursor to nearby objects. If another object is connected to the point, it will always be nearby and it might be snapped to. Since it is already connected, this would create a redundant constraint, or require to the user to cycle in order to snap to another object. Filtering the set of objects that can be snapped to eliminates this nuisance.

However, recognizing that a relationship is implied by a set of geometric constraints can require difficult geometric proofs as some complicated mechanism might imply additional relationships. From our experience, it appears that handling the simple cases—not snapping to the object being dragged, checking for existing connection constraints, applying basic geometric identities, etc.—filters out the vast majority of redundant snaps. Avoiding redundant snaps is an optimization; it is not critical to the correct functioning of snapping. However, not filtering all redundant snaps means that the solving techniques must be robust in handling redundant constraints.

### 9.1.2 Constrained Direct Manipulation

Briar uses augmented snap-dragging to initially establish constraints and then uses the methods of the differential approach to maintain these constraints during subsequent editing.

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

Dragging parts of drawings allows the user to experiment with the constrained model. This interactive animation is a useful tool for understanding and debugging constraints. It also opens up the possibility of *dynamic drawings:* models which are created to be dragged and played with. Such drawings contain unconstrained degrees of freedom, so they exhibit motion when their parts are dragged. The use of constraint-based approaches to animate drawings dates back to Sketchpad [Sut63] and special purpose techniques for interactively simulating mechanisms have been developed [Kra90, RK77, End90]. Differential constraint systems are also well suited for such mechanical simulation and animation tasks as well as interaction. Section 9.2 describes a special purpose application for mechanism sketching.

The facility with which we handle constraints opens the possibility of using constraints to aid in manipulation. Most constraints are used to represent structure in the drawing. However, temporary constraints that are easy to create and destroy are useful for making drawings easier to control. We call such constraints, which are meant to be short-lived, "lightweight" constraints.

Dragging is one example of a lightweight constraint. It is achieved by temporarily constraining the point being dragged to follow the mouse. Another useful lightweight constraint is the *tack.* A tack hold a particular point in place. It acts as an extra hand, making it easy to stretch or rotate an object. Nailing a point at a particular point in space is a common facility in constraint-based systems. Making it easy to place and remove the nails easily enables new uses for them. For example, tacks can perform the tasks that anchors do in traditional snap-dragging [Bie89], specifying the center of rotation and scaling.

### 9.1.3 Displaying And Editing Constraints

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

Textual languages for describing constraints, such as that employed in Juno [Nel85], have the advantage that they are editable. Unfortunately, they are distinct from the drawing and can be difficult to connect. Schematic representations of constraints, such as that presented by Borning [Bor86], are similar in that the constraint display is separate from the drawing.

Visual representations, such as in Converge [Sis91], CoDraw [Gro89], and Viking

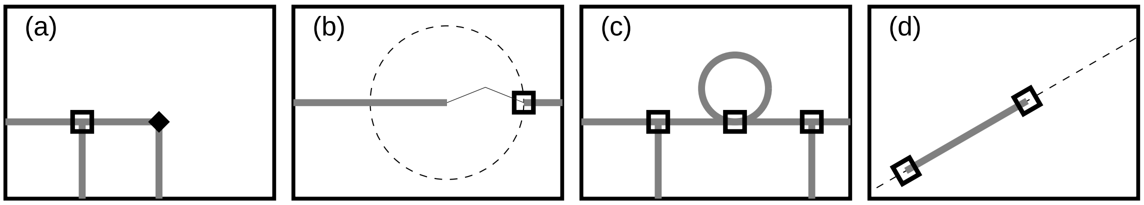


**Figure 9.4:** Snap-Dragging provides a basis for visual representation of constraints. The two basic constraints, point-on-object and point-to-point, are drawn as an empty square and a filled diamond (a) respectively. Other constraints are represented using the basic ones and alignment objects, for example, distance-between points (b), points-aligned (c), and orientation (d). Thin wedges as seen in (b), emphasize distance constraints. Color is used for constraint display when available.

[Pug92], superimpose constraints directly on the drawing. The connections between the relationships that will be maintained and the objects they affect are shown to the user. This is particularly dramatic when the model is moving subject to the constraints, such as when it is being dragged.

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

### A Visual Representation of Constraints

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

Augmented snap-dragging specifies all relationships by one of two basic types of snapping operations. The graphical depictions of these two types of snaps becomes a way to depict the constraints the snaps create. Alignment objects are also part of relationships and therefore they must also be made persistent, unlike traditional snap-dragging. The snapping symbols and alignment objects provide a graphical representation for a wide variety of constraints. Some examples are shown in Figure 9.4.

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

### Editing Constraints

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

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

Another technique for deleting constraints without pointing at them is to allow the user to temporarily disable constraint maintenance. In this mode, the user can manipulate the objects as if they did not have any constraints on them. When constraint enforcement is re-enabled, constraints which the user has broken are discarded. Feedback as to which constraints are broken and the ability to use snapping operations to reassemble things help make this a useful technique for editing constraints. A similar mechanism for destroying unwanted constraints is provided by Chimera [KF93], where the user can create a new "snapshot" of the drawing which shows the constraint broken.

## 9.1.4 Lessons from Briar

Briar was constructed as a testbed for ideas and techniques for constraint-based drawing. It was developed in the autumn and winter of 1990, before most of the work in this thesis. Briar was built with a predecessor to Snap-Together Mathematics. It employed a conjugate-gradient solver directly to handle under- and over-constrained cases, as described in Section 3.6, and a "Briar-style" two pass solver for soft controls, as described in Section 3.5.1.

Briar is an important demonstration of the differential approach because:

- it provides an example that shows the approach is viable;
- it shows that many of the difficult issues in using the approach can be addressed;
- the techniques that it introduced, such as augmented snapping, can be applied to other applications;

- it provided me with experience of what services constraint-based graphical editors required, which influenced the design of subsequent toolkits;
- it showed how Snap-Together Mathematics could be incorporated into an application, which led to alterations in the design of Snap-Together Mathematics;
- it inspired the Bramble toolkit because building a graphical editor without significant support to leverage off of was too much work;
- it emphasized many of the features required for the mathematical techniques, such as continuous motion and redundancy handling;
- it provided an interface that permitted creating larger, constrained models that taxed the performance of implementations, providing inspiration and test cases for exploring performance enhancements.

## 9.2 A Planar Mechanisms Sketcher

Creating drawings of mechanisms is a common use of a constraint-based drawing program. Not only do the constraints help accurately place objects to create the mechanism, but they also keep the mechanism together as it is manipulated. This permits the user to play with the mechanism to experiment with its behavior. Because mechanism creation was such a popular use of Briar, the first Bramble application was a special purpose drawing program designed especially for drawing and experimenting with planar linkage mechanisms. The *MechToy* application is illustrated in Figure 9.5.

Mechtoy defines a number of special purpose graphical object types. Linkage rods are line segments drawn to appear more like what is commonly used to draw mechanisms. The length of the rods are constrained like the rigid elements that they model. The point equality constraints used to connect pieces are drawn as hexagonal bolts to look more like a mechanical object. The ground points that provide fixed locations for pivot points are drawn using the standard symbol from mechanics texts.

Special adjuster objects permit altering the geometry of mechanisms. When an adjuster is dropped onto a linkage rod, the length of the rod is unfrozen. Deleting the adjuster re-freezes the rod's length.

Several MechToy object types have points that are nailed in place except when they are grabbed. For example, ground points are normally locked in place, permitting attached objects to pivot around them. However, if the user grabs the ground point, it is temporarily freed so it can be moved during dragging.

Special variants of objects permit creating a variety of mechanical contraptions. Sliders are line segments that have their endpoints nailed in place, and a floating bead that is limited to sliding along the line's length. Motors are special variants of line

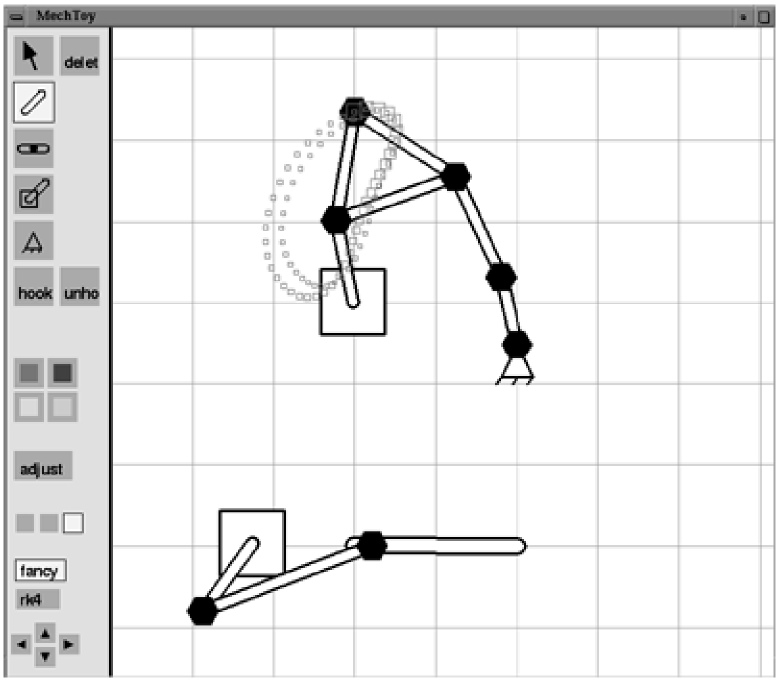


**Figure 9.5:** The *Mechtoy* application is specifically designed for drawing and animating planar linkage mechanisms. This picture shows a mechanism being animated, with smoke trails tracing the motion.

segments that have one endpoint nailed in place. When a button is pressed, a soft control is placed on the motor's orientation that causes it to rotate. Motors permit mechanisms to be animated.

Smoke trails are objects that can be drag-and-dropped on points in the mechanism. They remember a history of the point's position, and display it in a non-obtrusive fashion as a decaying trail of dots, as seen in Figure 9.5.

Mechtoy uses augmented snapping to infer constraints on objects as they are created. If an object is created in a position snapped to another, the two are bolted together. This permits mechanisms to be sketched very rapidly. Mechanism can easily be disassembled by deleting parts, such as the bolts between parts.

## 9.3 A Box-and-Arrow Diagram Editor

Another major use of a constraint-based drawing program is to create diagrams with arrows connecting the pieces. Constraints are particularly useful in such an application because they permit the structure of the drawing to be retained as the pieces are moved. This can save a considerable amount of work of reestablishing connections after each edit.

Diagrams are an important enough class of drawings that a market for specific tools for creating diagrams exists. Such tools can maintain connections without a general purpose solving mechanism because of the simplicity of the specific constraint problem. Simple dependencies suffice for keeping boxes and arrows attached. One of the many successful commercial examples is Visio[Sha93]. Visio is a particularly interesting example because it permits users to define new types of objects and connections using a spreadsheet interface.

A simple diagram editor called *Boxer* has been built with Bramble and is shown in Figure 9.6. Boxer provides a few basic object types, each providing connectors at points around their periphery. Boxer objects automatically display text inside themselves, although text-editing is not supported[4]. The objects are constrained to be axis aligned. They can be created and placed using a drag-and-drop interface from the palette provided at the top of the window. The drag-and-drop interface permits users to select icons from the palette and drag them onto the drawing area. The technique, which is also used in Visio, avoids separate modes for object creation.

Connecting lines are specified by providing two points on boxer objects. Arrows can be placed at either end. The connecting lines and arrows look at the normals of the points they are attached to in order to attach at a proper angle. The connecting arrows are drawn as Bezier segments to have a smooth appearance yet approach their destinations with the correct orientations.

[4]Text handling facilities are conspicuously absent from Bramble.

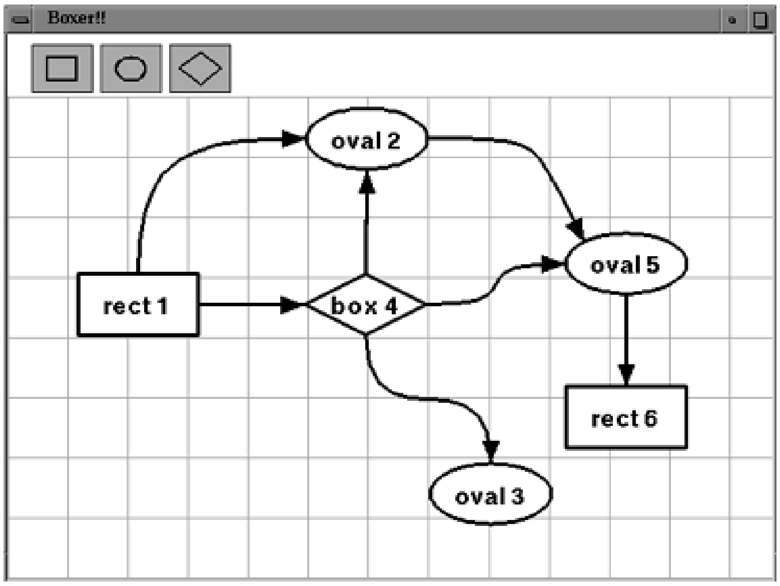


**Figure 9.6:** The *Boxer* diagram editor is designed for creating box and arrow diagrams.

Boxer was the first Bramble application to employ non-interpenetration constraints, as described in Section 8.4.2. Boxer objects are constrained not to overlap. As an object is dragged, it pushes other objects out of the way, just as physical objects would. The collision mechanism treats all boxer objects as rectangles. The simple collisions have been useful for a variety of purposes, including moving stacks of objects and clearing space around an object by pushing neighboring objects away.

Boxer is designed to be extended easily. It defines a Whisper function called `install-shape` that installs a new shape in the interface, adding a drag-and-drop icon to the top area of the screen. The `install-shape` function takes the output provided by `define-shape` (Section A.2.1), or the functions to create instances and draw icons can be created by hand.

## 9.4 A Curve Modeller

The *NewFF* program shown in Figure 9.7 is an application designed to permit experimenting with 2D parametric curves and constraints. It is named after Andy Witkin's original parametric curve manipulation program [Wit89b]. Like Witkin's original FF system, and several generations of successors described in [WGW90], [GW91b], and [GW91a], NewFF provides a variety of curve and constraint types, and offers easy addition of new types.

As discussed in Section 8.1.1, all that is required to add a new type of parametric curve is the parametric function used to draw the curve. NewFF is distinguished from

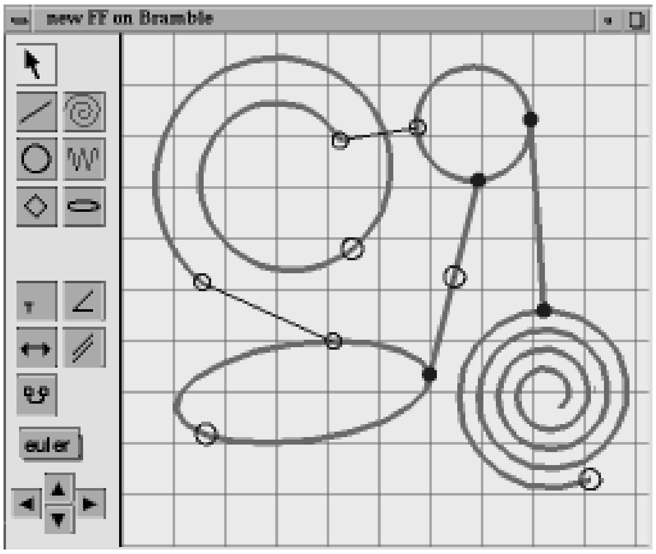


**Figure 9.7:** The *newFF* curve modeller allows experimenting with a variety of types of parametric curves and constraints.

its ancestors by permitting new types to be defined dynamically while the application is running. The parametric function is given as a Whisper expression, and a new curve type is defined, including an icon for creating instances. A small amount of auxiliary information is also required to specify the sampling of the curve, and initial values for the parameters. Because the dynamic addition of curve types interprets the function using Whisper, performance (especially for redraw) is poor. Therefore, new curve types are best added at compile time. A Mathematica program automatically generates C++ code for new Bramble object types from parametric functions.

The standard creation mechanism in the curve modeller simply creates an instance of the object with its default initial values for the parameters. The initial values can be defined in terms of an initial point value. When NewFF creates an object instance, it sets these parameters equal to the position of the mouse. Certain objects in NewFF have more sophisticated creation mechanisms. For example, line segments and circles are created with rubber banding and augmented snapping to infer connection constraints on the object.

The curve modeler allows a variety of constraints to be placed on objects. All constraint types attach to points, and therefore can be connected to any object. Available constraints include connection, distance, collinearity, and parallel-ness of four points.

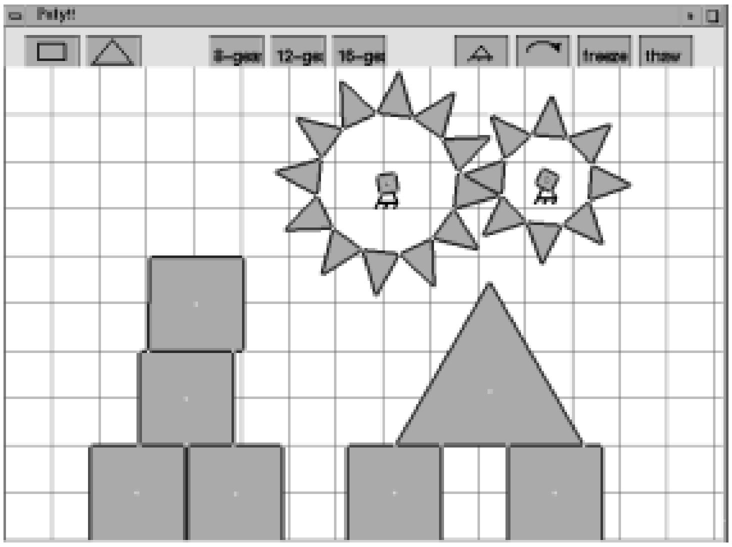


**Figure 9.8:** The *Poly* application for experimenting with planar collision simulation.

## 9.5 A Collision Simulator

A simple application to experiment with the planar convex polygon collision constraints of Section 8.4.2 has been built with Bramble and is shown in Figure 9.8. The *Poly* application permits several types of polygonal objects, such as blocks and sawtooth gears, to be created with a drag and drop interface. All objects automatically are constrained not to overlap other objects and to remain above the floor. Gravity, implemented by a soft controlling pulling downwards, can optionally be placed on objects.

A major use of Poly is to create mechanisms, so some features to facilitate this are provided. Using the drag and drop interface, the user can nail points in place, freeze objects, or place a torque source at the center of the object (e.g. turn it into a motor). All of the Bramble functionality, including the objects and constraints from MechToy, can be used in creating Poly models.

## 9.6 3D Construction Toys

Simulating Tinkertoys[5] was one of the original motivating applications for the creation of the constraint technology that the differential approach is built on. The idea is to create a graphical modeling environment where pieces can be connected together to build more complicated objects that have interesting behaviors. Tinkertoys, with pieces that can be plugged into one another to create mechanical connection, was an direct

[5]Tinkertoy is a registered trademark of PlaySchool, Inc.

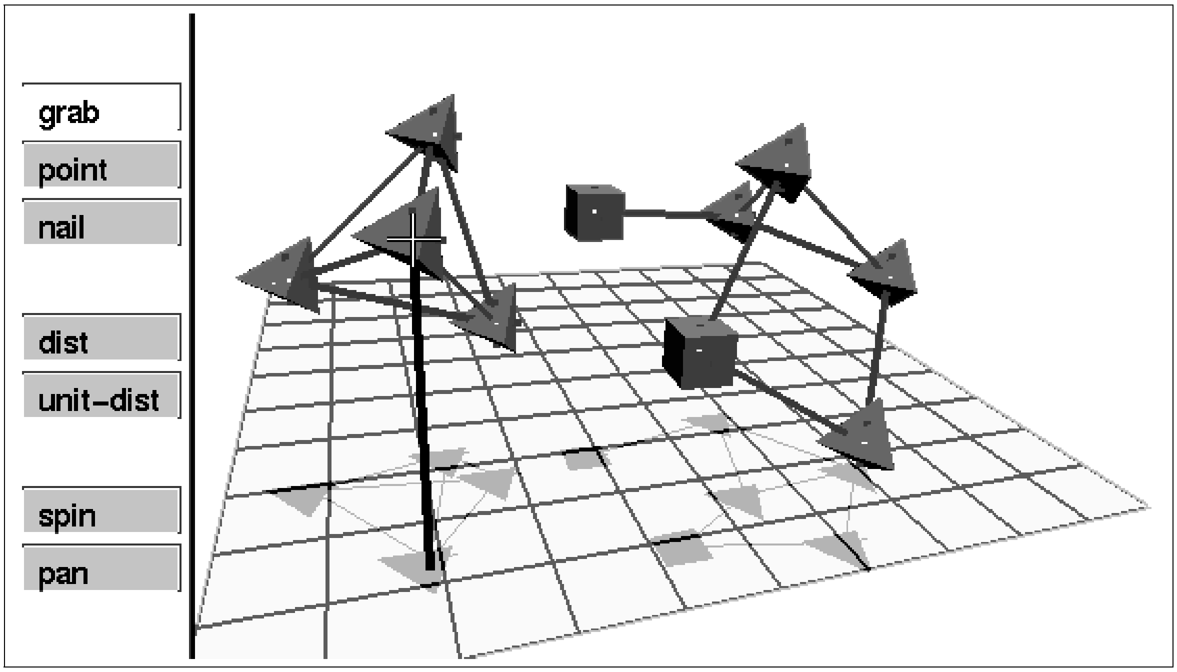


**Figure 9.9:** The *Ptinker* application permits the user to build simple models from point objects and constraints.

metaphor for constraint-based modeling. The freedom and flexibility needed to play with Tinkertoy models had served as a touchstone for evaluating 3D interfaces.

One of the earliest applications built with the original snap-together math implementation was a simple Tinkertoys like system which provided a simpler set of objects that could be attached with point-to-point or distance constraints. Although this application was called Tinkertoys at the time[WGW90], it models a simpler set of constructions. A version constructed with Bramble is shown in Figure 9.9. *Tinker* or Point Tinkertoys recreates the functionality of the original, allowing a user to create simple objects, connect them with distance constraints, and manipulate them with the mouse-pole.

Point objects in PTinker are denoted by small tetrahedra. Points that have fixed positions in space, called "Ethernails,"are denoted by cubes. Distance constraints are shown as lines between point objects. The interface permits specifying distance constraints by selecting two points to connect. When connected, the distance can be constrained either to be a unit distance or to simply maintain the distance between the points.

The *Toys* application, depicted in Figure 9.10, is a more ambitious simulation of tinkertoys. Toys includes pieces designed to mimic their counterparts in a real Tinkertoys set. The sizes and colors of the objects are reproduced in the simulation. Connecting pieces by plugging rods into holes on the yellow connector objects is modelled using

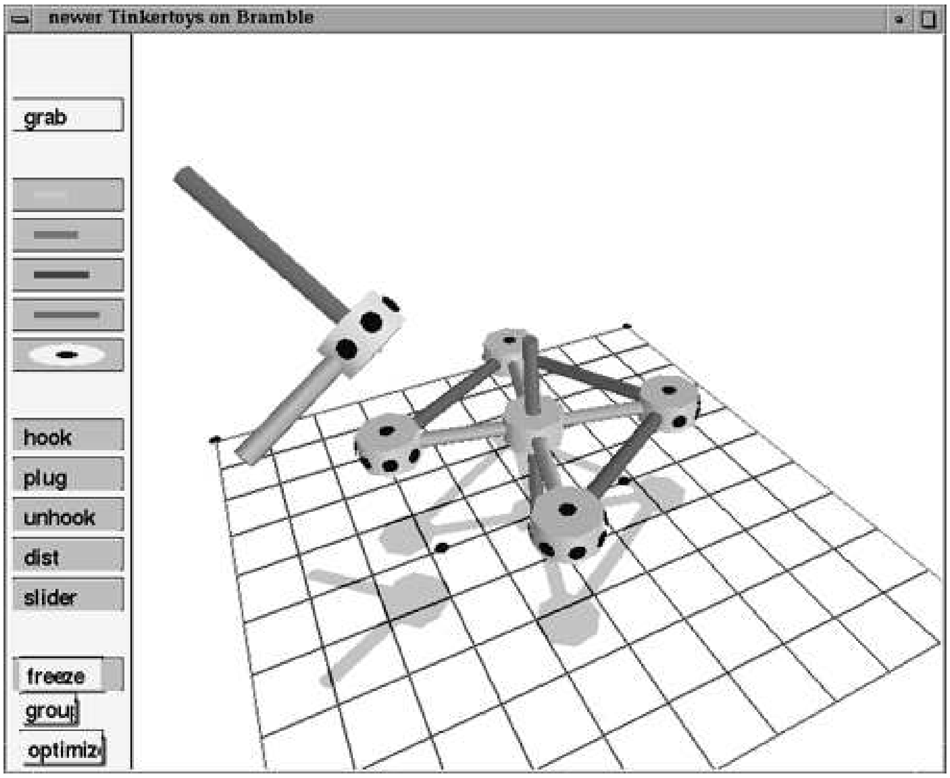


**Figure 9.10:** The Tinkertoys simulation permits users to create objects out of pieces modelled after the real tinkertoys.

constraints.

In order to test if the abstractions of Bramble were sufficient to create the range of behavior desired in a complicated application, no new functionality in the toolkit was added. Toys is written entirely in Whisper. The application provides a complete 3D system, with a tuned interface. This was important to show that such an application could be constructed with the tools, but also to provide a realistic testbed for interface ideas. The interface for Tinkertoy simulation provides a variety of interface challenges. The application must provide for the fluent manipulation of 3D objects, easy specification of constraints, and quick control of the view. Because the application is designed to be fun, it would be unacceptable if the interface was not fluent and expressive. Personally, I think the Toys application is a success — *I* enjoy playing with it.

### 9.6.1 Tinkertoy Pieces

Toys models most of the objects in a Tinkertoy set. The dimensions and colors of the real parts are reproduced. The rods and connector spools are made from standard cylinder primitives. The dots for the holes on connector spools are created by using a Bramble function that marks a `DistinguishedPoint` with a circle in its tangent plane. The connector spools use a `DistinguishedPoint` on their surface for drawing, but each hole also has a corresponding point inside the cylinder that is used for connection. This is required to model the fact that the rod pushes into the hole.

Connections between pieces are specified by picking an end of a rod and a hole. The pieces then "self-assemble" to establish the connection. The self-assembly is phased into several stages. First, the pieces orient themselves, then they fly together, and finally, the piece pushes into the hole (rods actually do go inside the holes). The phasing was created for purely aesthetic reasons, and is implemented by demons that wait until one phase is complete before beginning the next. To make the connection operation seem more natural, Toys attempts to move only one object in order to establish the constraints. Heuristics are used to decide which object to move. The system prefers to move pieces that have not been connected yet, and prefers to freeze connector spools instead of rods. If both the rod and its target have other connections, neither is frozen.

The Tinkertoy interface permits objects to be grabbed and dragged using the mouse-pole, and the view to be controlled using a virtual trackball by grabbing the ground-plane. Rods are plugged into the holes on the connector pieces. Once connected, the rod can rotate is the hole.

To aid in picking, Tinkertoys provides semantic snapping so that only valid objects can be picked when various operations are to be performed. For example, when plugging a rod into a hole, only unattached rod ends and empty holes are snapped to. For unplugging, only rod ends that are plugged into holes are seen by the snap server.

### 9.6.2 Performance of Tinkertoys

Simulating Tinkertoys provides difficult performance goals. Because of the complex behavior of the 3D models, rapid refresh rates are important. However, tinkertoy models have large numbers of constraints. Each rod into hole connection requires 5 constraints. The total number of constraints to simulate can grow large very quickly. For the Toys application I set a specific performance goal: on the machine in our lab (an SGI Indigo 2 Extreme), the program should achieve acceptable performance on objects that were as complicated as those that could be built with the small set of real Tinkertoys. The small jar of Tinkertoys contains roughly a dozen rods and a half-dozen connectors.

Meeting the performance goals for Toys pushed the limits of Bramble. The objects in Tinkertoys were all standard cylinders, quaternion transformations, and frame-alignment constraints. Because these are often used objects, they were carefully optimized. However, users of Toys quickly assembled models that were large enough that that $O(n^2)$ linear system solving dominated the performance. Therefore, a number of implicit constraint methods were created to enhance the performance of Toys. All of the methods use only standard Bramble features.

The implicit constraint methods in Toys exploit the fact that connected pieces form rigid bodies. Technically, a rod that plugged into a connector's hole has a degree of freedom to rotate around its axis. However, since the rod is symmetric, this degree of freedom is invisible to the user. Toys can, therefore, treat the two pieces as a single rigid body. Rather than having two objects and 5 constraints, the system need only simulate a single rigid object. If the other end of the rod is plugged into another connector, the degree of freedom can be manipulated by the user. Therefore, Toys can use implicit constraint techniques for at most one connection per rod. Another case where the symmetry of a rod permits implicit constraints is when a rod is connected to the ground. In such a case, the rod can be frozen.

When a connection between two parts is accomplished by an implicit constraint, the connection must work the same way that it would if normal techniques were used. All connections must appear the same to the user; they are specified and deleted in the same ways. This creates several complications. For example, when a connection is deleted, there are no constraints to delete if the constraints are implicit. Instead, disconnection must ungroup the two objects. This is implemented by having phantom connection objects that represent implicit constraints. The phantom objects specify a deleter hook that performs the ungrouping. The phantom objects are also convenient for implementing save and load.

Creating a connection with an implicit constraint also poses a challenge as the constraint must self-assemble first. This is accomplished by initially creating the connection with regular constraints. A demon waits for the connection to be made. When the two pieces are connected, the demon converts the connection to use implicit constraints. If the demon fails by timing out, it does not convert the connection. In fact, it takes a precaution against the constraint never being satisfied by adding additional damping.

To demonstrate the benefits of the implicit constraint methods in Toys, we consider building an example object. The merry-go-round object, shown in Figure 9.11, is a model that is shown in the instructions to the real Tinkertoy set. It is made of four small rods, four medium rods, a long rod, four connector spools, and a slider connector. These pieces have a total of 93 variables. Toys running on an SGI Indigo 2 Extreme takes approximately 50 milliseconds to redraw the display with these pieces, whether they are attached or not.

Without the automatic implicit constraint techniques, the merry-go-round requires 85 constraints[6]. Dragging this merry-go-round with the mousepole takes approximately 320 milliseconds per 4th order Runge-Kutta step. This provides a frame rate of little more than two frames per second. This performance is unacceptable. However, using the implicit constraint methods, the merry-go-round only requires 40 constraints, and has a much smaller number of variables. The frame rate for this model is over 6 frames per second, as each Runge-Kutta step requires only approximately 100 millisec-

[6] The slider is implicitly connected to the vertical pole.

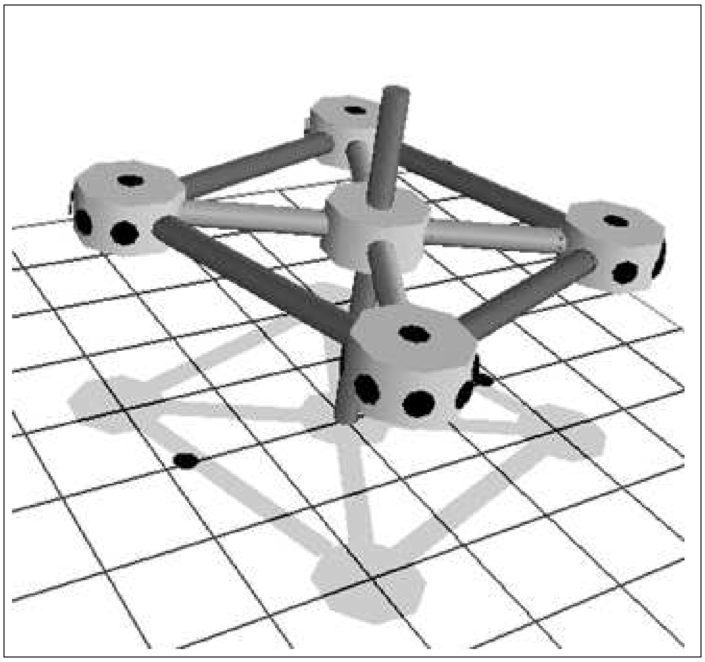

**Figure 9.11:** A Merry-Go-Round constructed in the toys application from 9 rods, 4 connector spools, and a slider connector.

onds. While interaction is a little sluggish, it seems acceptable.

The merry-go-round illustrates the limits of implicit constraint techniques as well. The merry-go-round actually only has two degrees of freedom: it can slide up and down the pole, or spin around the pole. All of the parts except for the pole form a single rigid body. The merry-go-round could be simulated by grouping these pieces as one unit, rather than using constraints. Identifying that more complex objects are rigid is difficult, and is left for future work.

## 9.7 Scene Composition

A scene composition system permits a user to position objects in a 3D space, assign properties to them, light them, position the camera, and send the model off to a renderer to have an image created. Scene composition is a good testbed for 3D methods because it requires controlling objects, lights, and cameras, and it avoids many of the issues of how models are to be edited and represented.

The Bramble scene composition application is called *Showoff*. The name comes from the fact that it was originally intended as a viewer for objects in a standard[7] file

[7] I don't know how much of a standard it is, but a large number of sample models in this format can be found in public archives.

format called OFF. Showoff is not an application in the traditional sense, but rather, it is a framework for building demonstrations of interaction techniques for scene composition problems. Unlike other applications, Showoff does not create windows or fill the scene with graphical objects. Instead, it defines Whisper functions that make it easy for scripts to do these tasks. For instance, it defines a function that automatically creates a framed view, complete with a standard camera and buttons along the side to set various viewing modes. Showoff creates commands for finding and manipulating object aspects.

Showoff's main role is for testing and demonstrating attributes and interaction techniques for 3D applications. The typical showoff script creates some objects to provide a sample scene, sets up some initial controls and constraints, and begins the interactive loop. Showoff binds keys and menus to commands for many standard interaction techniques. For example, mouse buttons can be used to grab a point with the mousepole or through-the-lens with either soft or hard constraints.

Showoff has been the primary vehicle for experimentation with controls for positioning objects, lights and cameras. A wide variety of controls are provided for the user, or the script, to mix and match. Figures 8.3, 8.5, 8.10, and 8.8 are all scenes from Showoff demonstrations.

Showoff provides few features aside from those used to manipulate scene objects. For example, the interface does not help the user create objects, they must be created by writing Whisper code. However, Showoff does contain an extensive set of commands that allow the differential approach to be applied to the problem of scene composition directly. Showoff presents the user with a large range of controls, including most of the attributes of Section 8.1, that can be applied as needed. Controls are combined by locking: any control can have its value nailed in place. The idea is that the user will employ a set of constraints and controls that conveniently described the image to be created.

Showoff does not address the difficult problem of determining how to present a wide range of controls to a user. A sufficient number of controls are available in Showoff that accessing them is a problem. Showoff uses a haphazard combination of menus, keys, and buttons to specify operations. The interface is "designed" to maximize the number of controls made available to the user, even at the expense of usability.