Graphical Interaction Techniques in Constraint-Based Geometric Modeling

Steve Sistare

Harvard University

Abstract

I present a three-dimensional geometry editor called *Converge* which takes advantage of several new interaction techniques in constraint-based geometric modeling to achieve a greater ease of use than that found in comparable systems. Constraints are presented as graphical symbols that are incorporated into the geometric design, and both constraints and geometry may be manipulated with the same graphical gestures through the direct-manipulation paradigm. A novel feature of *Converge* is that constraints may be created implicitly as a side effect of creating geometry, giving the user the benefit of using constraints without explicitly having to create them. Lastly, constraints are used in a novel way to facilitate interactive creation and positioning of geometry in three space, despite the limitations imposed by commonly available two-dimensional display and input devices. A video demonstrating *Converge* will be presented.

Keywords: direct manipulation, drafting aids, constraint, geometric modeling, interactive techniques.

1 Introduction

The design of geometric models can be a painstaking and time-consuming task. While drafting aids are available in many CAD packages, they are often primitive or deficient in the means they offer for placing geometry in a design or for subsequently modifying the geometry. Modification in particular can tax a user's patience when it requires many deletion, creation, or perturbation operations to effect a conceptually simple change in the design.

One area of research that attempts to address these deficiencies involves the use of constraints on the geometry as a means of both specifying and controlling its form [2][3][4][8]. The additional information about the structure of the geometry that is provided by the constraints allows the user to specify modifications at a higher conceptual level, which the system can then translate into the lower-level changes needed to achieve the desired modification. The form in which the constraint information is demanded from the user determines the ease of use of any geometric system that is based on constraints, and is one of the basic problems to be addressed in the design of such a system.

The *Converge*[1] editor achieves ease of use through a variety of means. It allows the manipulation of both constraints and geometry using the direct-manipulation paradigm, which is well established as being of central importance in many easy-to-use systems. Whenever geometry is modified using direct manipulation, a solver is called which updates the geometry in accordance with the existing constraints. The graphical presentation of constraints makes them easier to understand and use, and the fact that they may be created implicitly eases the burden on the user. Other features, to be explored in the remainder of this paper, further increase the usefulness of constraints.

The solver will not be discussed here except to say that it has been designed to solve robustly and efficiently the unstructured network of constraints and geometry that is produced by *Converge*. The basic algorithm is a combination of the Newton-Raphson method and the conjugate-gradient method that can solve for n constrained variables in $O(n^2)$ time. However, this algorithm is combined with a new traversal algorithm that simultaneously partitions and solves many networks of constraints and variables in close to linear time. See [7] for a detailed presentation of the solver.

2 Graphical Presentation of Constraints

A key distinction that separates *Converge* from similar systems is its incorporation of graphical constraint icons into the geometry. The user can at once see the geometry and the constraints on it in a unified framework. This may be contrasted with Nelson's *Juno* system [4] and Gross's *Co* system [3] in which a textual programming language description of constraints is used,

Author's present address: Thinking Machines Corp., 245 First St., Cambridge, MA 02142; sistare@think.com

[1] Constraint solver for geometry

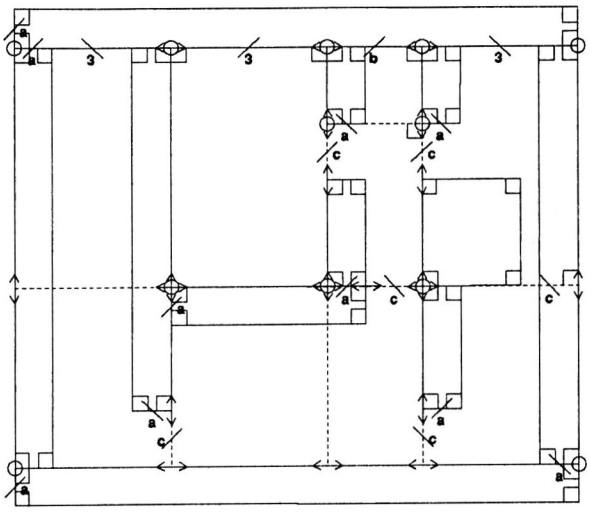

Figure 1: Architectural floor plan produced with *Converge*.

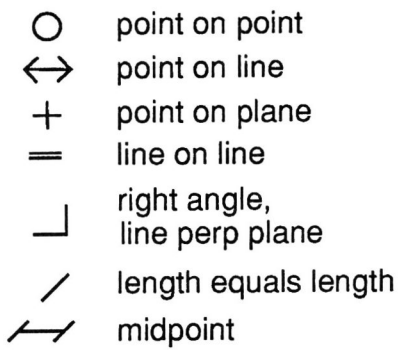

Figure 2: Constraint icons.

and with Borning's *Thinglab* [1], in which a network diagram is used to convey the constraint description. In each of these cases, the constraint description is presented disjointly from the geometry. By unifying the presentation of constraints and geometry in a graphical framework, *Converge* allows the user to more readily understand the constraints, their relationships to each other and the geometry, and the structure of the overall constraint network. A further advantage to unification is that the same graphical gestures may be used to create, manipulate, and destroy both constraints and geometry, making the system easier to use.

The use of graphical constraints is illustrated in Figure 1, which shows an architectural floor plan designed using *Converge*. This design has over one hundred constraints that enforce congruence, coincidence, and orthogonality between various geometric entities in the design. On a color monitor, the constraint icons would be displayed with a different hue than the geometry, but we may discern them in black and white with the aid of the key shown in Figure 2. Several of these icons, the slash and the right-angle symbol, were taken from mathematics, while the rest were designed to graphically show the degrees of freedom available in each constraint. For more information on the design process that was used to derive these icons, see [7].

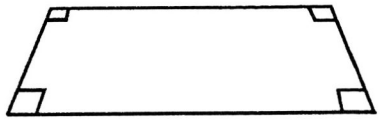

Figure 3: Square seen in perspective projection.

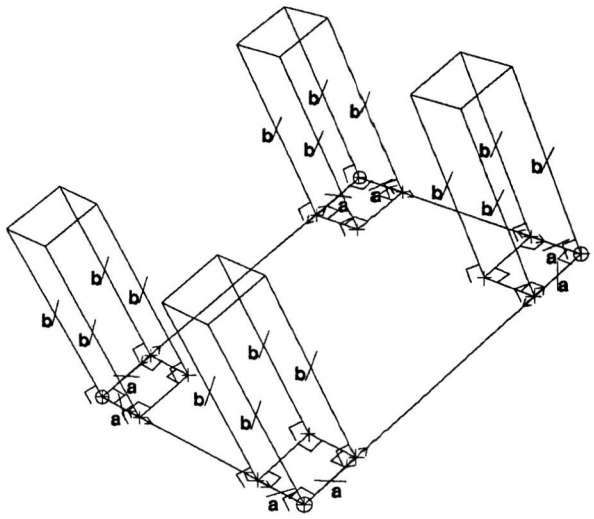

Figure 4: Table designed using *Converge*. Legs are pointing up.

Certain characteristics of the constraint icons deserve mention. One is that they are neither three-dimensional nor two-dimensional objects, but share characteristics of both. They are planar objects, and as such need to be displayed parallel to the screen, which means they are computed in screen space. However, they are associated with three-dimensional geometry that has been projected onto the screen and has been subjected to perspective foreshortening which shrinks the geometry with increasing distance from the viewer. To avoid visual contradictions, the icons must be similarly scaled. Figure 3 shows a tilted square in which this scaling is apparent.

One problem in displaying constraints graphically with the geometry is that the user may be subjected to

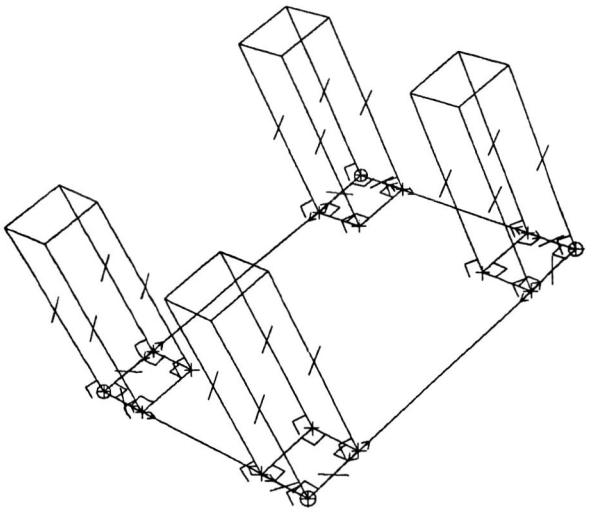

Figure 5: Filtered view of previous table.

information overload. There is more information present in a drawing annotated with constraints than in one that shows only geometry, and furthermore, the typical user is not accustomed to dealing with constraints in any form whatsoever. Thus, *Converge* provides operations for filtering constraint information to the point where the user is comfortable dealing with the resulting display. Constraint icons may be made invisible individually, by type, or by the geometry they use. All icons may also be uniformly scaled to give them more or less prominence with respect to the geometry. Lastly, the level of detail that is displayed for each constraint may be reduced, decreasing clutter but increasing ambiguity. The design for a table (with legs pointing up) may be seen fully detailed in Figure 4 and filtered in Figure 5. The latter figure is more appealing to work with, even though some information is lost. The user can either remember the elided details having seen the first figure, or he can disregard them until they are relevant, at which time the detailed view can be reinstated.

The constraint icons in Figure 2 were designed to be purposely ambiguous for these same reasons. Specifically, there is ambiguity as to exactly which geometric entities are used by each constraint. The ambiguities may not be obvious, however, because humans are quite good at resolving ambiguity using context. This is in fact the justification for allowing such ambiguity! However, the user may ask *Converge* to highlight the entities used by a selected constraint when confusion arises.

3 The Editor

The *Converge* editor provides most of the basic features for manipulating geometry that are found in standard geometry editors such as MacDraw on the Macintosh and Idraw under the X windows system. In addition, *Converge* provides operations to manage constraints and provide control over the constraint solver. Supported geometric primitives include points, lines, and polygons. A variety of predefined constraint types exist that act on these primitives, some of which were listed in Figure 2. A new primitive called a *scalar* is also available, which is simply a named variable. The floor plan in Figure 1 contains three scalars named a, b, and c, which are used in length-equals-length constraints to control the width of the walls, doorways, and one room. A separate menu is dedicated to the scalars in a design, listing their names and current values.

Menus related to constraints include the constraint menu, which lists the available constraints; the filter menu, which contains options for filtering constraints as described in the previous section; and the locus menu, which implements locus specification, to be described in Section 5. Other menus are available for editing, file system access, and three-dimensional viewing specification.

Converge allows both explicit and implicit control over the solver. By default, the solver is not immediately called after the addition of each constraint due to the potentially slow response time for a large design. The user may thus batch new constraints together and click on the solve menu option to satisfy the constraints all at once. It is possible that the design may be unsolvable due to some of the constraints being contradictory, in which case the design is left as it was before the solver was called. In this case, the user may select the undo constraint option, which may be repeatedly called to delete the last constraint entered, and retry the solver after some number of constraints have been withdrawn. Conversely, the solver may succeed, but return with an undesirable result, perhaps due to the design being underconstrained. A menu option is available to restore the geometry to its previous state in this case. Lastly, design primitives may be frozen in their present state with the freeze option, which prevents selected primitives from being modified by the solver.

In addition to being called explicitly, the solver is called implicitly whenever the design is modified. This occurs when the user drags existing points or lines with the mouse, or changes the value of a scalar by clicking on its value box in the scalar menu. For example, we can drag the internal horizontal partition of the floor plan in Figure 1 and get the result shown in Figure 6, and then grab the left-most wall and drag it to the left to yield Figure 7.[2] Note that the plan's first, second, and fourth sections from the left have expanded equally, while the third section has remained of constant width. We can also shrink the walls by changing the scalar parameter a and get the result in Figure 8. Similarly, we can modify the width of the legs of the table in Figure 4 by modifying a, yielding Figure 9. All of these manipulations are

[2]The constraints in these figures have been filtered out for clarity.

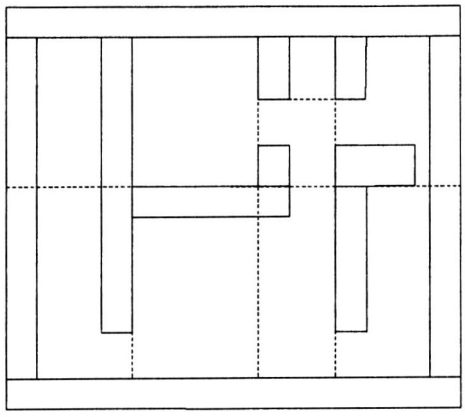

Figure 6: Floor plan after moving interior partition.

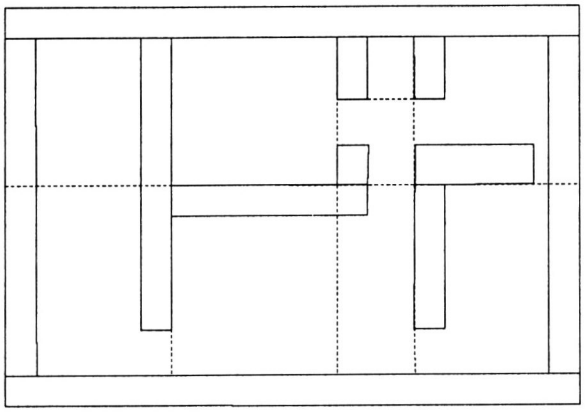

Figure 7: Floor plan has been stretched by dragging the outer wall.

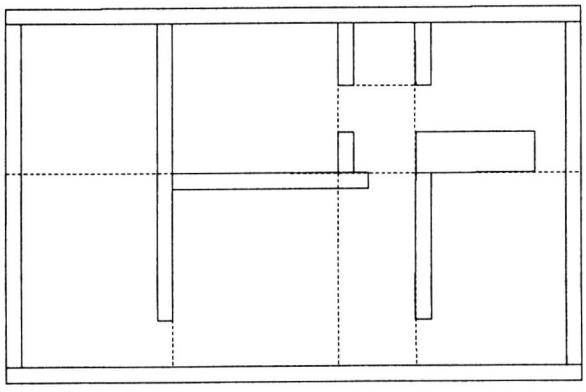

Figure 8: Width of walls has been modified.

performed by the solver in under one second on a Sun 4.

Thus far, I have presented the basic methods for interacting with the *Converge* editor. In the next few sections, I will expand upon these methods and present additional interaction techniques, many of which are unique to *Converge*.

4 Constraint-Based Interaction Techniques

Users of constraint-based geometry editors must work slightly harder than users of non-constraint based systems while entering their designs, as the former must input constraints as well as geometry. The extra effort required may discourage them from trying constraint-based systems. For systems such as *Converge* to succeed and be accepted, users must be convinced that constraints are worth using. More specifically, the payoff that comes from using constraints must outweigh the effort required to enter them, and the ratio of payoff to effort defines the utility of using such systems. We can increase the utility by either reducing the effort required or increasing the payoff, and I will describe methods that accomplish both objectives.

5 Implicit Constraint Creation

The obvious way to create a new constraint is to explicitly select the constraint type and the (previously created) geometry that will participate in the constraint. This can be done fairly rapidly with my system due to the interactive and graphical nature of my interface; indeed, more rapidly and easily than with any of the constraint-based systems described in [2], [3], and [4]. When one chooses a constraint type, constraint-creation mode is turned on, and all subsequent geometric selection operations are used to create constraints of that type. To create right angles, one clicks on the right-angle menu item and proceeds to select pairs of lines by pointing to them with the mouse and clicking. Each pair will be constrained to form a right angle. The user may switch to another constraint type at any time or may cancel constraint-creation mode.

Explicit constraint creation is the most straightforward and versatile way to create constraints. It requires that the geometry has been previously created and that explicit operations subsequently be performed to add the constraints. Constraints may also be created implicitly, however, by integrating the creation of constraints and geometry in such a way that they are created as a consequence of creating geometry. This is an attempt to increase the usability of the system by minimizing the amount of time that the user must spend dealing with constraints.

How may we achieve such an integration? Sutherland's *Sketchpad* implicitly created coincidence constraints whenever a point was created close to a line or a circle. By contrast, *Converge* relies on a previously established constraint context called a *locus* that dictates

how newly-created geometry will be used to form new constraints.

5.1 Locus Specification

The locus is a means of describing constraints that is less restrictive than simply listing types such as point-on-line. A locus specification consists of a geometric primitive such as a point, line, or plane, together with a constraining relationship, which is *not* the same as a constraint type. Conceivable relationships include *on*, *orthogonal*, and *parallel*. The user specifies a locus by first selecting a geometric primitive from those that already exist in the design and then selecting a relationship from a menu of possibilities that currently include *on* and *orthogonal*.

While a locus remains in effect, there exists an implicit constraint between the locus's geometry and any as yet uncreated geometry. The implicit constraint is made explicit when new geometry is created, at which time a new constraint is created whose type is defined by the concatenation of the locus geometry, the locus relationship, and the new geometry.

To illustrate the creation of constraints through locus specification, we construct a small fragment of the floor plan shown earlier. We first draw a horizontal line and select it as the current locus, using the *on* relationship. We then create several points along the line, and as each one is created, a point-on-line constraint is automatically supplied by the system, resulting in Figure 10. Next, we change the locus relationship to *orthogonal*, without having to reselect the line – it remains the locus by default. Lastly, we create lines anchored at the existing vertices. The system automatically supplies the right-angle constraints, and we have constructed the design in Figure 11 without explicitly creating any constraints. Most designs can be constructed using a combination of implicit and explicit constraint creation. Approximately half of the constraints in the table that was shown in Figure 4 were created using locus specification.

5.2 Copy Operations

In standard geometry editors, the copy operation provides a quick and easy way to create geometry without adding it piecemeal. The copy operation extends naturally to the constraint domain and provides yet another way to implicitly create constraints.

It is nonsensical to copy constraints in isolation; rather, they must be part of a geometric copy operation. To initiate a copy operation, the user first selects the geometry to be copied. The constraints that may be copied are then limited to those that constrain the selected geometry. These constraints may be divided into the *internal* constraints, which apply to selected geometry only, and *external* constraints, which apply to selected geometry and some geometry that is not being copied. The user next selects the constraints to be copied, either by picking them individually (for complete generality),

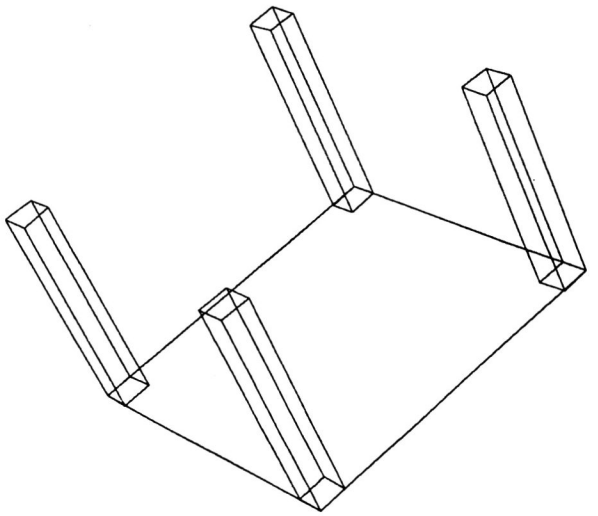

Figure 9: Table with thinner legs.

Figure 10: The line is chosen as the locus, and point-on-line constraints are automatically added by the system when points are created.

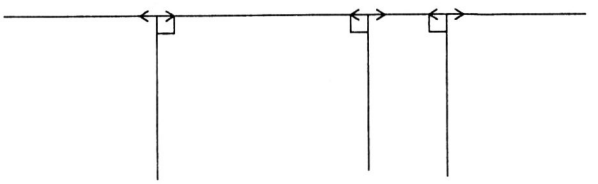

Figure 11: Design fragment that may be produced entirely through the use of locus specification.

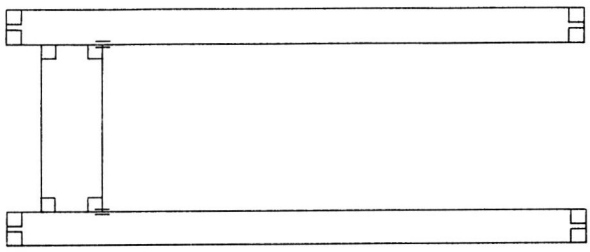

Figure 12: An unfinished design that will be developed in the next few figures using copy operations.

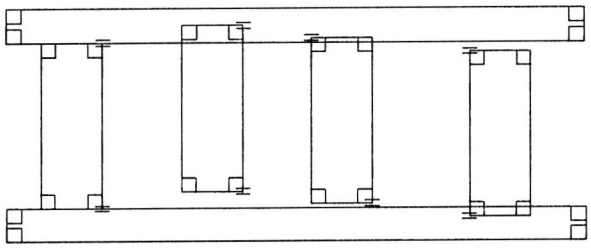

Figure 13: Copies with internal and external constraints are made and translated.

or by asking the editor to identify and select the internal or external constraints. Copying a geometric object and its internal constraints yields an identical but isolated copy of its geometry, and copying an object with its internal and external constraints yields a duplicate with the same geometric context as its parent.

An example serves to illustrate these distinctions. Consider the design shown in Figure 12. If we select the perimeter of the vertical rectangle, ask the system to select the internal constraints, and perform a copy operation, we will get a new rectangle that is free to translate about the screen. However, if we ask the system to select internal and external constraints, then we get a copy that is constrained to lie between the upper and lower rails. The state of being selected is automatically transferred to the new primitives after a copy operation, so we may repeatedly click on the copy menu item to generate several replicas of the original rectangle. We next translate the copies to their approximate destinations as shown in Figure 13. Lastly, we invoke the solver to satisfy the constraints, which gives us Figure 14.

6 Constraints and Interactive Manipulations

We have seen that the use of constraints in a design affords an ease of modification not found in non-constraint-oriented systems, and this forms the foundation of their utility. However, there are further applications of constraints that increase their basic utility and make constraint-based systems look even more attractive. This section details several such applications, all of which involve using constraints to guide interactive manipulations of geometry by the user.

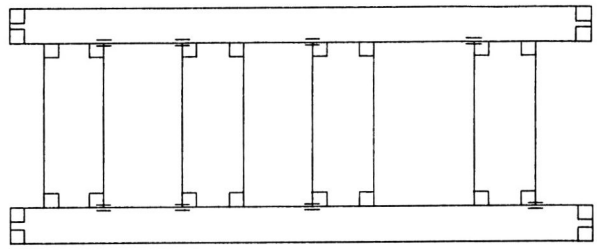

Figure 14: The solver is called to satisfy the copied constraints.

6.1 Locus Constraints

Locus specification was presented in the previous section as a means of automatically creating constraints when new geometry is created. However, it can also be used to give the user greater interactive control over the placement of the same new geometry. To achieve this control, the implicit constraint that is defined by the locus is applied to the new geometric primitive during its initial placement. For example, if a line locus with relationship *on* is in effect and the user creates a point, the point will move back and forth along the line as the mouse is moved in approximately the same direction. When the user clicks the mouse, the point is permanently incorporated into the design at its current position, and the implicit constraint is made explicit. One nice feature of this operation is that the constraint is already satisfied at the time it is created. This is not true of constraints that are explicitly created by the user; rather, the solver must be called to satisfy them.

The solver is used to enforce the constraint during the placement operation. In the example above, for every small motion of the mouse, the change in mouse position is added to the current position of the point. In general, this will move the point away from the line. The new coordinates are passed to the solver, which modifies them slightly to bring the point back to the line. As long as there is a component of mouse movement in the direction of the line, the point will experience some motion along it. The solver works quickly under these circumstances, because the perturbations to the point are small, and the constraint system it must solve consists of only one constraint and three variables.[3] Thus, the user enjoys

[3] On a Sun 4, the solver can process dozens of these requests per second.

rapid, real-time response to mouse movements, which results in smooth motion of the point along the line.

The point-on-line scenario outlined above becomes even more interesting when we consider that in a three-dimensional object space, the line need not lie in the plane of the screen, in which case motion along the line also implies motion towards and away from the viewer. Locus constraints thus give us the ability to precisely position a point in three space using a mouse with only two degrees of freedom! Other combinations of locus and geometry yield a similarly useful result. If a polygon locus with relationship "on" is chosen, subsequently created points will track the polygon in three space as they are moved by the mouse during initial placement. Selection of a line or polygon locus with relationship "orthogonal" will affect placement of new lines. The first endpoint of a new line may be maneuvered freely by the mouse until the user clicks to set the point in its place. As the second endpoint is dragged with the mouse, the line between the endpoints will telescope in and out in a fixed direction that is perpendicular to the locus polygon or line.

The interactive three-dimensional positioning capability offered by locus constraints is one of the unique features of *Converge*.

6.2 Constrained Transformation

Aggregations of geometric primitives may be translated and rotated in *Converge* just as they may be in any geometry editor. In addition, however, *Converge* offers a limited form of constrained translation, in which an object may be translated as a rigid body subject to simple constraints. Specifically, the user may request that constraints on one vertex of the object be maintained as the object is translated. For example, an object may be translated such that one of its vertices remains coincident with a stationary plane. This gives the user a degree of control over the translation operation similar to that provided by a locus constraint, and the solution process for the constrained point is in fact identical. The object is rigidly translated in lockstep with the point by taking the difference between the position of the point from one step to the next and adding it to every vertex in the object. Thus, as the mouse is moved, the object moves in three space, maintaining contact with the plane at one vertex. Other constraints that are appropriate for use in this manner include point-on-line, line-on-line, right-angle, and several others.

6.3 Constrained Perturbation

The perturbations that we applied to the floor plan developed in section 3 were *unconstrained* in that we were free to drag a wall or vertex in any direction whatsoever without resistance. This of course violated many of the constraints in the design, which had to be restored by calling the solver when the dragging operation was complete. However, a violated design can have many possible solutions, and it is not always clear which one is desired by the user. For this reason, several forms of *constrained* perturbation are also available in which some of the constraints are continuously maintained while the geometry is being manipulated. Constrained transformation and locus specification are special cases of constrained perturbation in which exactly one constraint is interactively maintained.

Ideally, we would like to be able to maintain *all* the constraints in a design while continuously dragging geometry with the mouse. In any design of significant size, however, the solver is simply too slow to allow interactive manipulation in this fashion. Someday, this will be an excellent way to use up the hundreds of mips that will sit on our desktops. For the present, however, we must look for alternatives that approximate this ideal behavior. The obvious alternative is to maintain a subset of the constraints while the manipulation is in progress, and call the solver to satisfy the rest of the constraints when the manipulation terminates. This can give a good approximation of how the design should look while it is being perturbed if the constraints are chosen appropriately.

One possibility is to choose those constraints that use the vertex or line being perturbed. If we allow only the single vertex or line to move, and hold the rest of the geometry fixed during the dragging operation, then this subset provides good results in some but not all cases. If one side of a rectangle were dragged, for example, all four right-angle constraints of the rectangle would be considered, as each endpoint of the side is used in the right angle at its vertex and at two other vertices. Thus, the rectangle would scale nicely. If we tried to drag a single vertex of the rectangle, however, the other three vertices would remain fixed, and the three right-angle constraints acting on the vertex would lock it in place. This situation may be rectified by defining a variant of the drag operation that can modify the geometry used by the constraints as well as the vertex being dragged. Of course, this adds more free variables to the constraint system that must be solved and degrades interactive response, but still gives good results for some geometries.

For complete generality, the constraints that control a perturbation may also be individually selected by the user. As before, the dragged vertex or line may be modified solely, or all geometry of the constraint set may be modified.

6.4 Automatic Positioning

The previous sections detail ways in which constraints are used to guide the interactive manipulations that the user must perform to move an object into its place in a larger design. However, constraints may also be used to accomplish this task without the need for *any* manipulation on the part of the user. By creating constraints between an object and the larger design and calling the solver, the object may be pulled into place.

The walls of the architectural floor plan shown earlier

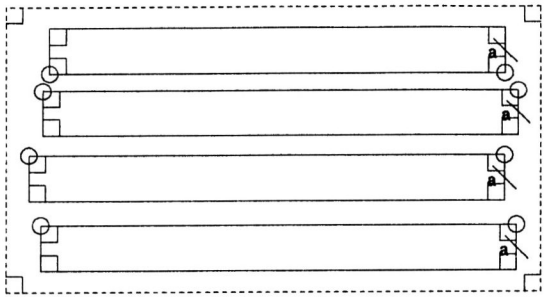

Figure 15: The walls will be positioned using constraints.

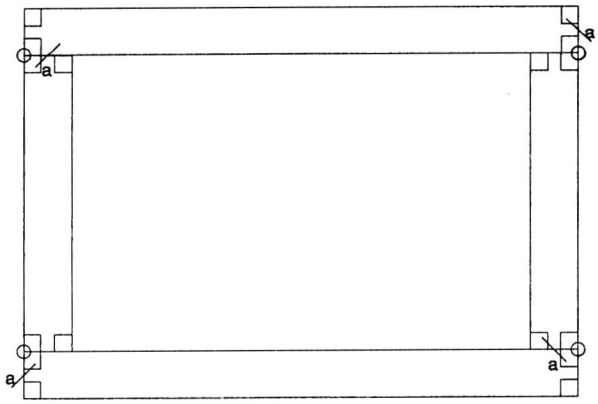

Figure 16: Previous figure after calling the solver.

may be positioned in such a manner. Figure 15 shows an early stage in the design of the floor plan in which the outline of the plan is present along with several randomly placed walls. Point-on-point constraints have been added between the corners of the walls and the outline, but the solver has not yet been called. Figure 16 shows that the desired result has been achieved after calling the solver.

7 Summary

Converge employs a number of new constraint-based interaction techniques that make constraint-based editing easier. Constraints may be created implicitly using locus specification and copy operations, relieving the user from the need to explicitly add constraints to a design. The techniques of constrained perturbation and constrained transformation provide the user with a means of *interactively* modifying parts of a design under the control of constraints that can be selected by the user. This allows local modifications to be made to a design without solving the entire design. In addition, locus specification, constrained perturbation, and constrained transformation offer a a new solution to the problem of positioning geometry in three space; they reduce the available degrees of freedom to one or two, allowing the positioning to be done with a standard mouse.

The *Converge* editor is the first constraint-based system to effectively incorporate constraints graphically in a geometric design. This feature, together with the use of the direct-manipulation paradigm and new interaction techniques, yields an editor with a look and feel that is substantially different from that of existing constraint-based systems. The result is a system that is easier to use and understand.

Acknowledgments

This research was supported in part by contracts from the Lockheed Corporation.

References

[1] Borning, A., "Defining Constraints Graphically", *SIGCHI Bulletin*, 17(4), April 1986, pp. 137-143.

[2] Borning, A., "Constraint Hierarchies", *SIGPLAN Notices*, 22(12), October 1987, pp. 48-60.

[3] Gross, M., Ervin, S., Anderson, J., Fleisher, A., "Constraints: Knowledge Representation in Design", *Design Studies*, 9(3), July 1988.

[4] Nelson, G., "Juno, A Constraint-Based Graphics System", *Computer Graphics*, 19(3), July 1985, pp. 235-243.

[5] Olsen, D., and Allan, K., "Creating Interactive Techniques by Symbolically Solving Geometric Constraints", *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, October 1990, pp. 102-107.

[6] Shu, Nan C., *Visual Programming*, (New York: Van Nostrand Reinhold Company), 1988.

[7] Sistare, S., *A Graphical Editor for Three-Dimensional Constraint-Based Geometric Modeling*, Ph.D. dissertation, Harvard University, 1990. Available as technical report TR-06-91, Center for Research in Computing Technology, Harvard University.

[8] Sutherland, I. "Sketchpad: A Man-Machine Graphical Communication System", *Proceedings of the Spring Joint Computer Conference*, IFIPS, 1963, pp. 329-345.