Inferring Constraints from Multiple Snapshots

David Kurlander
Steven Feiner

Department of Computer Science
450 Computer Science Building
Columbia University
New York, NY 10027

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques—interaction techniques; I.2.6 [Artificial Intelligence] Learning—concept learning; D.2.2 [Software Engineering]: Tools and Techniques—user interfaces

Additional Key Words and Phrases: constraints, empirical learning, graphical editing

Abstract

Many graphics tasks, such as the manipulation of graphical objects, and the construction of user-interface widgets, can be facilitated by geometric constraints. However, the difficulty of specifying constraints by traditional methods forms a barrier to their widespread use. In order to make constraints easier to declare, we have developed a method of specifying constraints implicitly, through multiple examples. Snapshots are taken of an initial scene configuration, and one or more additional snapshots are taken after the scene has been edited into other valid configurations. The constraints that are satisfied in all the snapshots are then applied to the scene objects. We discuss an efficient algorithm for inferring constraints from multiple snapshots. The algorithm has been incorporated into the Chimera editor, and several examples of its use are discussed.

Technical Report CUCS 008-91. Submitted to *ACM Transactions on Graphics*.

1.0 Introduction

Geometric constraints are used extensively in computer graphics in the specification of relationships between graphical objects [Sutherland63a][Borning 79][Myers 88][Olsen90]. They are useful during object construction to position components relative to one another precisely, as well as during subsequent manipulation of the components. Several graphical techniques, such as grids, snap-dragging [Bier86] and automatic beautification [Pavlidis85] were developed to make the initial construction phase easier, since specifying constraints explicitly can be a complex task. However, when objects are to be manipulated frequently, permanent constraints have an advantage over these other techniques in that they need not be reapplied. Permanent constraints can be particularly useful when subsequent editing of a scene is required, in constructing parameterized shapes that can be added to a library, in specifying how the components of a window should change when the window is resized, or in building user-interface widgets by demonstration.

We introduce a technique for inferring geometric constraints from multiple examples, replacing the traditional constraint specification process with another that is often simpler and more intuitive. Initially, the designer draws a configuration in which all constraints are satisfied, and presses a button to take a snapshot. A large number of possible constraints are inferred automatically. Subsequently, if the scene is modified and other snapshots taken, previously inferred constraints are generalized or eliminated so that each snapshot is a valid solution of the constraint system. For example, we can define two objects to be squares, constrained to maintain the same proportional sizes, by taking a snapshot of two squares, scaling them by equal amounts, and taking another snapshot. Then, if the length of one of the square's sides is changed, the lengths of its other sides and the sides of the second square are updated automatically. The designer need not have a mental model of all the constraints that must hold.

Furthermore, the designer may make snapshots at any time. If after one or more snapshots a set of graphical objects do not transform as expected or if the constraint solver cannot reconcile all inferred constraints simultaneously, the graphical primitives can be manipulated into a new configuration with constraints turned off, and a new snapshot taken. The incorrectly inferred constraint set is automatically modified so that the new snapshot is a valid constraint solution.

There are a number of problems with traditional constraint specification that this new technique attempts to address:

■ *Often many constraints must be specified.*

Complex geometric scenes contain many degrees of freedom, and often most of these need to be constrained.

■ *Geometric constraints can be difficult to determine or understand.*

Using constraints requires geometric skills that many people using graphical editors do not have. Declaring constraints is foreign to the act of drawing, and is an unnatural part of the graphical editing process.

■ *Debugging, editing, and refining constraint networks are complex tasks.*

When incorrect or contradictory constraints are specified, the designer needs to debug the constraint network, which can be a cumbersome process. To support the debugging task, a visual representation is usually provided for constraints. WYSIWYG editors need a special mode for displaying constraints, and when constraints and graphical objects are presented together, the scene becomes cluttered if more than a few constraints are displayed simultaneously.

Many approaches have been taken to solve these limitations. The first problem was addressed by Lee, who built a system to construct a set of constraint equations automatically for a database of geometric shapes [Lee83]. In doing so, he worked with a restricted class of mutually orthogonal constraints, and required that the geometric shapes be aligned with the coordinate axes. Lee's problem domain and assumptions restricted the set of constraints such that there was never any ambiguity about which to select. In our domain the initial ambiguity is unavoidable, and we rely on multiple examples to converge to the desired constraint set.

One of the innovations of Myers's Peridot [Myers86][Myers88] is a component that infers constraints automatically as objects are added to the scene. A rule base determines which relationships are sought, and when a match is found the user is asked to confirm or deny the constraint explicitly. This reduces much of the difficulty inherent in choosing constraints—the designer is prompted with likely choices. Peridot's geometric inferencing component is limited to objects that can be represented geometrically as boxes aligned with the coordinate axes. As more degrees of freedom are allowed, however, it becomes increasingly difficult to infer geometric relationships from a single example.

Maulsby's Metamouse [Maulsby89] induces graphical procedures by example, and infers constraints to be solved at every program step. To make the task more tractable, he considers only touch constraints in the vicinity of an iconic turtle that the user teaches to perform the desired task. These constraints are treated as post-conditions for learned procedural steps, and not as permanent scene constraints. Complex relationships between scene objects can be expressed through procedural constructions, but the relationships between objects in these constructions tend to be unidirectional, and procedures for every dependency need to be demonstrated.

The difficulty inherent in understanding interactions among multiple constraints and debugging large constraint networks has been addressed by the snap-dragging interaction technique [Bier86][Bier88] and by an automatic illustration beautifier [Pavlidis85]. In snap-dragging, individual constraint solutions are isolated temporally from one another, so that their interaction cannot confuse the artist. The automatic beautifier infers a set of constraints sufficient to neaten a drawing, but the constraints are solved once and discarded—they are isolated temporally from subsequent user-interaction. In the approach described here, constraints can interfere with one another when a new solution is computed. However, the conflicting constraints can be removed by taking additional snapshots.

A number of systems provide visual representations of constraints to facilitate debugging. Sutherland's SketchPad [Sutherland63a][Sutherland63b] connected constrained vertices together with lines accompanied by a symbol indicating the constraint. One of Borning's ThingLab implementations also allowed new *types* of constraints to be defined and viewed graphically [Borning86]. Nelson's Juno editor [Nelson85] provided a program view of constraints. Peridot communicated constraints as English language fragments during confirmation, and Metamouse used buttons for confirming and prioritizing constraints. Our technique never requires that the users work with individual, low-level constraints. In both the specification and debugging stages, they can think entirely in terms of acceptable configurations of the illustration. The inferred constraints can be tested by manipulating scene objects, and the constraint set refined through additional snapshots. For those that prefer a more direct interface for verifying the inferred constraint set, we provide a browser that displays constraints in a SketchPad-like fashion. Because our technique is particularly useful in heavily constrained systems, we allow constraints in the browser to be filtered by type or object reference.

Our technique is an application of learning from multiple examples, also known as *empirical learning*. Several empirical learning systems are discussed in [Cohen82]. In contrast, generalizing from a single example is called *explanation-based learning* and is surveyed in [Ellman89]. Explanation-based learning requires a potentially large amount of domain knowledge to determine why one explanation is particularly likely. As we illustrate in subsequent examples, there are often few or no contextual clues in a static picture indicating that one set of constraints is more likely than the next, so we felt the empirical approach was warranted. Empirical learning algorithms have been extensively studied by the AI community, but we developed our own to take advantage of certain features of the problem domain and to make learning from multiple examples a feasible approach to geometric constraint specification.

We have implemented this technique as part of Chimera, a multi-modal editor with support for editing graphics, interfaces, and text [Kurlander90]. Constraints can be inferred on both graphical and interface primitives. Chimera is being developed as a testbed for experimenting with techniques for graphical editing *by example* [Kurlander91].

In Section 2, we illustrate the user's view of constraint specification with a number of examples. We provide a detailed description of our algorithm in Section 3. In Section 4 we discuss implementation details. Finally we mention limitations of the approach, present our conclusions, and discuss future work in Section 5.

2.0 Examples

In this section we show three examples of how constraints are inferred from multiple examples within the graphics and interface editing modes of Chimera. To facilitate the initial construction of the scene, Chimera provides both grids and snap-dragging alignment lines. All figures in this paper were generated directly from Chimera's PostScript output.

2.1 Rhombus and Line

Suppose that we would like to add permanent constraints to the rhombus in Figure 1a, so that during subsequent graphical editing it will remain a rhombus, its horizontally aligned vertices will be fixed in space, and the nearby line will remain horizontal, of fixed length, to the right and at the same height of the bottom vertex of the rhombus. After the initial scene is constructed in Figure 1a, the user presses the *snapshot* button in the editor's control panel. Next, the user translates the top and bottom vertices of the rhombus to make it taller, and translates the horizontal line to the same *Y* coordinate as the rhombus's bottom vertex, but to a different *X* so that its position will not

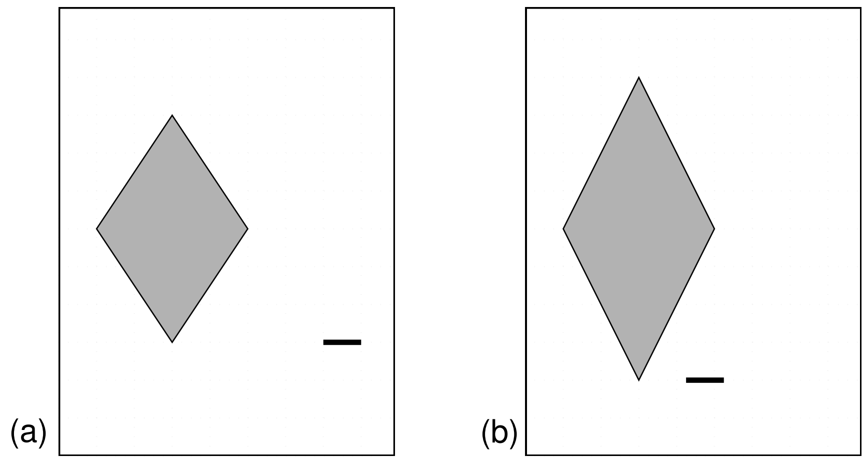

FIGURE 1. Two snapshots of a rhombus and line

be absolutely constrained in *X* with respect to the bottom vertex. The user presses the snapshot button once more. Figure 1b shows the second snapshot.

Initially constraints were turned off. Now when the user turns them on from the control panel and edits the scene, the constraints inferred from the snapshot are maintained by the editor. In Figure 2a, the user has selected the horizontal line and moved it upwards. The top and bottom vertices of the rhombus automatically move so that the demonstrated constraints are maintained. When, in Figure 2b, the top joint of the rhombus is selected and translated to a higher grid location, the bottom rhombus vertex and the horizontal line both move appropriately.

The abovementioned constraints were all specified implicitly, without the user having to express their intent in low-level geometric terms. Inferring this information from a single example would be problematic, since it is not clear how to distinguish between those parameters that should be fixed (such as two of the rhombus's vertices, and the length and slope of the horizontal line) and those that should be allowed to vary (such as the length of the rhombus's sides, and the locations of the horizontal line's vertices).

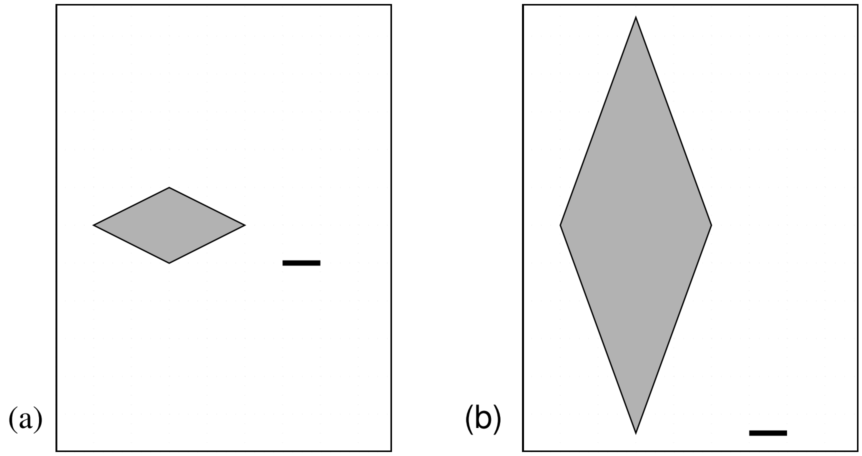

FIGURE 2. Two constrained solutions to the snapshots in Figure 1.

For this example, our system requires about a third the number of user operations as a typical interface for traditional constraint specification. Here, snapshot specification requires eight operations: two vertices and one segment are independently selected and translated, and two snapshots are taken. In contrast, the declarative specification could easily involve 24 operations in a typical interface. Equating the distance between two pairs of vertices requires five operations in all: four vertex selections, and one constraint operation. This step is repeated three times to equate the length of three of the rhombus segments to the fourth, requiring 15 operations. Two of the rhombus's vertices must be selected and fixed in space, at a cost of three more operations. In three additional operations, the two vertices of the horizontal line are constrained to be horizontal and of fixed length, and the constraint between one of these vertices and the bottom of the rhombus requires two selections, and another constraint operation. Though the number of operations necessary to perform a particular task relates only indirectly to its difficulty, we believe that the snapshot technique often compares favorably to declarative specification in terms of ease of use.

2.2 Resizing a Window

Constraints are useful in constructing user-interfaces because they allow the attributes of one interface object to be defined in terms of the attributes of others. For example, when a window is resized, the position and size of the contents may change. Figure 3a shows a window that we have constructed in Chimera's interface editing facility, containing an application canvas (the darkly-shaded rectangle), a scrollbar, and three buttons that invoke menus. After positioning these widgets within the parent window, the user presses the snapshot button. The components of the window are then shifted into another configuration, shown in Figure3b, and a second snapshot is taken. Precise positioning in these snapshots was achieved by using a combination of grids and snap-dragging.

The user intends that the buttons be a fixed distance above the bottom of the window, that the left side of the Basics button be a constant distance from the window's left, that the right side of the File Ops button be a constant distance from the window's right, and that the Transformations button be evenly spaced between the inner sides of the two other buttons. The scrollbar's dimensions are intended to be fixed by the top and right sides of the window, the top of the buttons, and by its constant width. The application canvas should be fixed relative to the left and top of the window, the top of the buttons, and the left side of the scrollbar. Now, when we turn on constraints and select the upper right corner of the window (while the lower left corner is fixed), the window and its contents reshape as shown in Figure 3c.

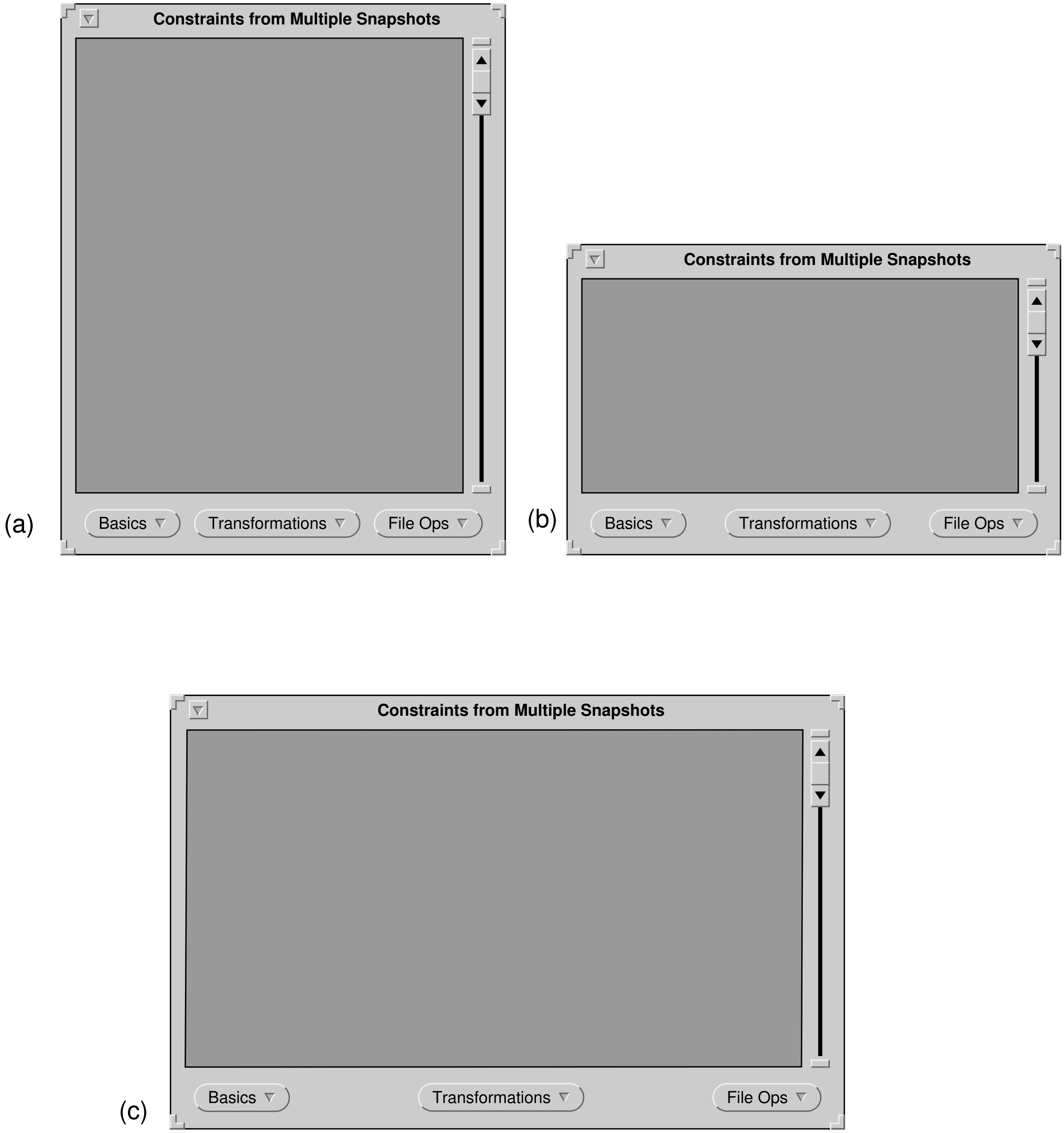

FIGURE 3. Specifying window resizing constraints. (a) and (b) are the two snapshots, (c) was produced by dragging the upper right corner of the window.

2.3 Constraining a Luxo™ Lamp

This final example applies to both graphical editing and user-interface construction. We would like to constrain a 2D illustration of a Luxo lamp, so that it behaves like a Luxo lamp. In particular, we want the various pieces to remain connected, the base to be fixed at its initial location, and the arms of the lamp to remain a constant length. Other constraints are important as well, but instead of determining which are significant ourselves, we would prefer to edit the lamp into a number of valid configurations and take snapshots. To control the direction of the lamp's beam, we have built a simple dial widget out of a circle and line, and we specify the behavior of the dial relative to the Luxo lamp by demonstration as well. Figures 4a and 4b show the initial two snapshots of valid configurations of our illustration. Note that the constraints inferred from these two snapshots are independent of the particular editing operations chosen, as explained in Section 3.

After taking the first two snapshots, we turn on constraints and try to manipulate the Luxo lamp, but the constraint solver indicates that it cannot solve the system. The source of the problem is an *incidental constraint,* that is, a constraint that was evident in the first two illustrations, but was not an intended relationship. When incidental constraints interfere with a desired configuration they can be removed by manipulating the scene into the new configuration with constraints turned off, and taking an additional snapshot. There is no need to individually examine inferred constraints to find and cull unwanted relationships. An additional snapshot is shown in Figure 4c.

Now the various components of the lamp move as we had intended. In Figure 5, we manipulate the lamp into three configurations by moving its top joint and adjusting the beam direction dial. Note that these two controls are not independent—when the dial is rotated, the arms of the lamp can move during the solution of the constraint equations, since it does not uniquely determine a lamp configuration. Here it would be useful to allow a traditional declarative constraint to be placed on the joint if we want to change only the beam direction while keeping the arm fixed.

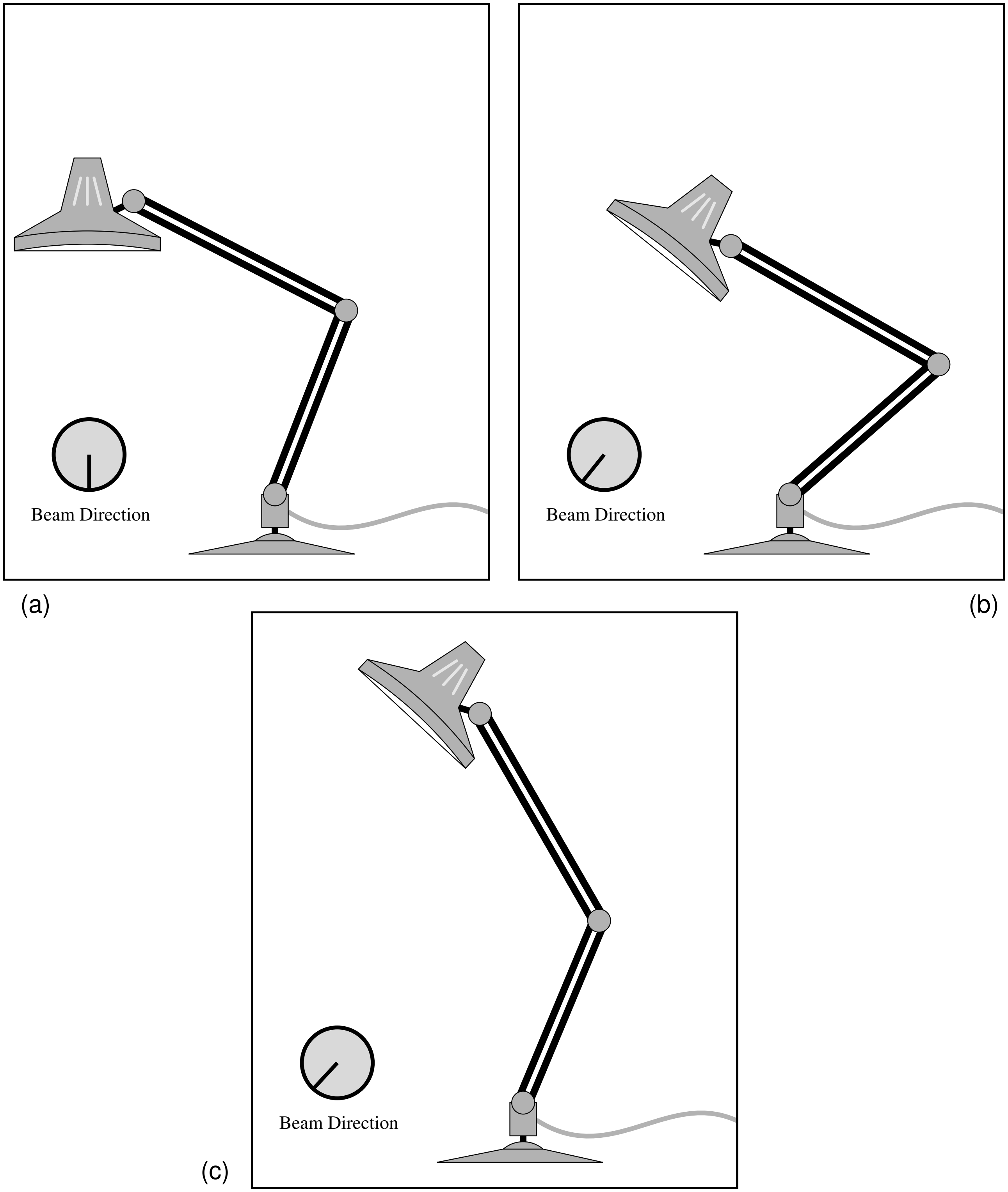

FIGURE 4. Teaching Luxo constraints. Three snapshots of valid configurations, provided as input.

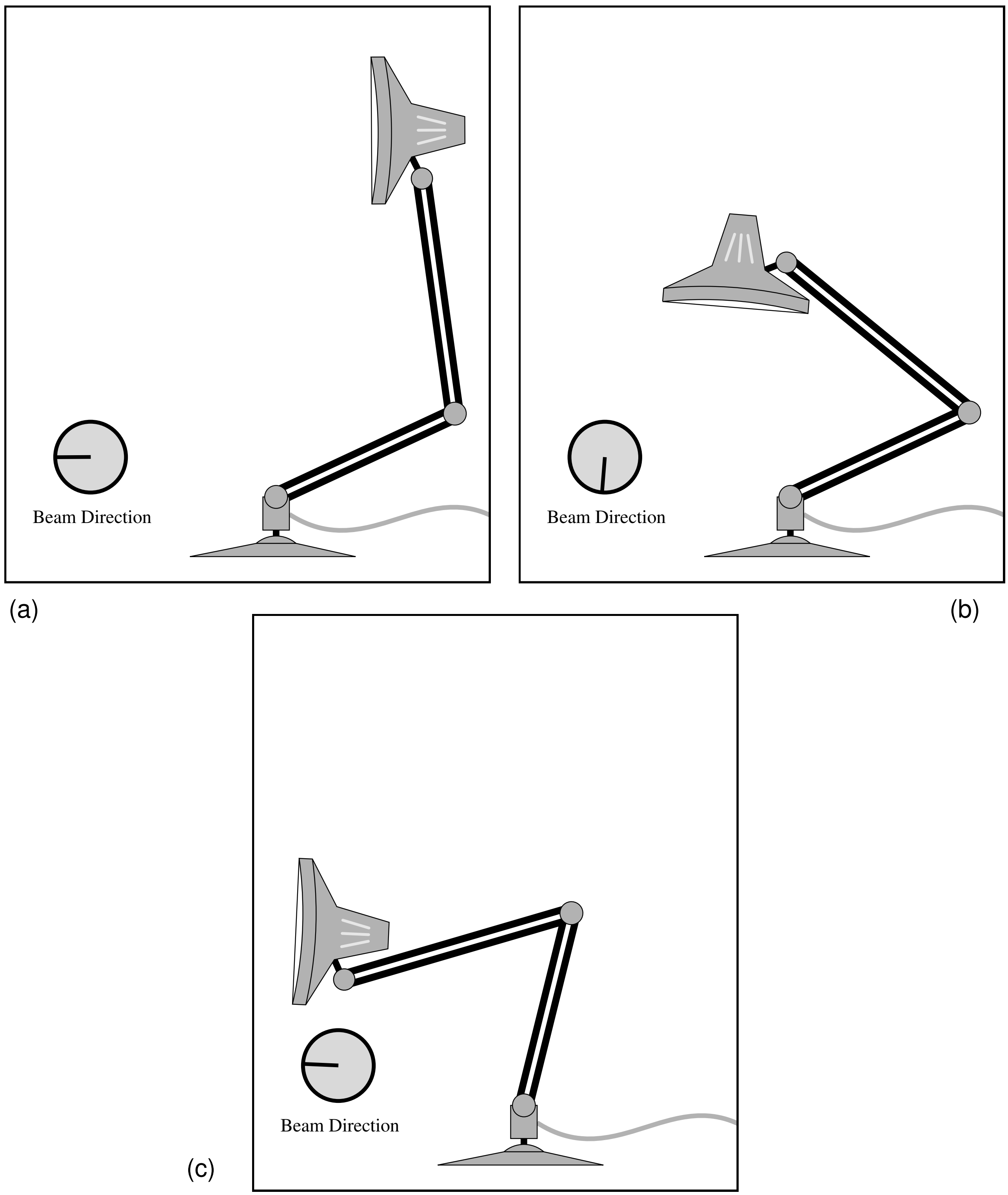

FIGURE 5. Luxo on his own. Configurations created by manipulating uppermost joint and Beam Direction dial.

3.0 Algorithm

In this section we discuss the set of constraints that our system infers. Then we present an efficient algorithm for inferring these constraints, and demonstrate the algorithm on a simple example. Next we analyze the complexity of the algorithm, and discuss how parameters can be inferred.

3.1 The Constraint Set

All objects in the Chimera editor are defined geometrically in terms of vertices, and constraints fix the relationships between these vertices. Based upon a finite set of example scenes, an infinite number of arbitrary constraints can be inferred. Hence we have chosen to infer a fixed set of geometric relationships that have proven particularly useful in graphical editors.

Our system infers both absolute and relative geometric constraints. *Absolute constraints* fix geometric relations to constant values. *Relative constraints* associate geometric relations with one another. For example, an absolute constraint might fix a vertex to be at a particular location, or a distance to be a constant scalar. A relative constraint might fix two distances or slopes to be the same. The following chart lists the constraints inferred by Chimera. Each relative constraint on the right corresponds to an absolute constraint on the left.

Absolute Constraints	**Relative Constraints**
Fixed vertex location	Coincident vertices
Distance between two vertices	Relative distance between two pairs of vertices
Distance between parallel lines	Relative distance between two pairs of parallel lines
Slope between two vertices	Relative slope between two pairs of two vertices
Angle between three vertices	Equal angles defined by two pairs of three vertices

The relative slope constraint fixes one slope to be a constant offset from another (when represented in terms of degrees, not y/x ratio). Each of the relative distance constraints fixes two distances to be proportional to one another. Since two of the above constraints are subsumed by others, there is no need to explicitly support them in either the inferencing component or the constraint solver: the coincident vertices constraint is subsumed by the absolute distance constraint between vertices, and the absolute angle constraint is subsumed by the relative slope constraint. Parallel and orthogonal vector relationships are largely captured by the relative slope constraint (which, for example, in the former case would not only fix the vectors between two pairs of two vertices as parallel, but would also constrain their relative directions). Similarly, the absolute angle relation captures collinearity, with an additional ordering on the vertices.

The algorithm discussed in this section finds all of the abovementioned constraints that hold over a sequence of snapshots. Many higher-level constraints can be formed by the composition of these lower-level constraints, and thus are also inferred by the algorithm. For example, the constraint that one box be centered within another is captured by two relative distance constraints between parallel lines.

3.2 Algorithm Description

In the rest of this section, we describe the algorithm that infers these constraints. An overview of its steps is given in Table 1. It may be helpful to refer back to this table during the subsequent discussion.

```
IF first snapshot THEN
    add vertices to initial transformational group
ELSE BEGIN
    split transformational groups to form new child groups
    identify intra-group constraints of child transformational groups
    identify inter-group group-to-group constraints due to splitting transformational groups
    identify inter-group vertex-to-vertex constraints
    break previously instantiated constraints that have been violated
    form delta-value groups from broken absolute constraints
    make a copy of the constraints with redundancies filtered out for the solver
END;
```

TABLE 1. Steps of the inferencing algorithm

With the first snapshot, the scene is entirely constrained, and each subsequent snapshot acts to reduce or generalize the constraints in the system. If we were to represent explicitly each constraint that could hold at any one time, the space and time costs would be prohibitive. Instead, we economically represent similar constraints over sets of vertices as *groups*. For example, after the initial snapshot, all vertices are constrained to a set location, and the distance and slope between each pair of vertices is fixed, as is the angle between each set of three vertices. Although we could instantiate each of these constraints explicitly, it is far more efficient to represent the vertices as a group with a tag indicating the relationships that hold among all of its members. As will be discussed later, groups can also accelerate the process of determining which constraints hold over a series of snapshots, and can ultimately reduce the number of constraint equations that are passed to the solver.

3.2.1 Transformational Groups

The most important type of group in our inferencing mechanism is the *transformational group*. A transformational group contains a set of vertices that have always been transformed together since the first snapshot. At the first snapshot, the algorithm places all the vertices into a fixed location transformational group, since their positions are initially constrained to be fixed. As vertices are transformed, our undo mechanism keeps track of the sets of vertices selected and the transformation applied, and this information is used by the inferencing mechanism to fragment existing transformational groups into smaller ones. The transformations that can be applied in our system currently include translations, rotations, and isometric scales, although we plan to extend our algorithm to work with any affine transformation.

3.2.2 Intra-Group Constraints

We can very efficiently determine *intra-group constraints*, that is, constraints that hold within a given transformational group. Figure 6, shows various affine transformations and the geometric relationships that they preserve.[1] The transformation listed in the top half of each box maintains the relationships listed in the lower half of the box, and those relationships in the boxes above it. For example, if a transformational group has only been scaled and translated, the slopes, angles,

1. Note that rotation in this diagram refers to isotropic rotation, and the vector relationship is the combination of slope and distance constraints.

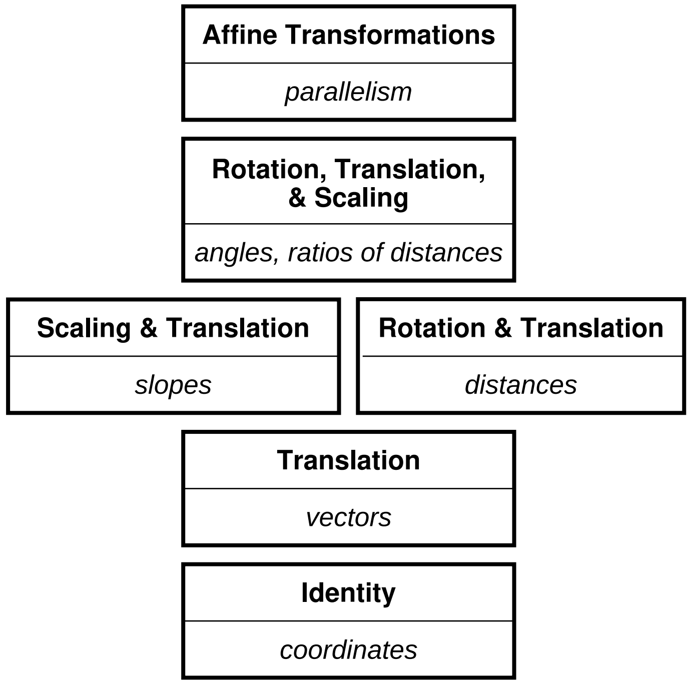

FIGURE 6. Transformations and the geometric relationships that they maintain. Reprinted with permission from [Bier86].

ratios of distances, and parallel relationships are all maintained. By tracking the transformations that have been applied to a transformational group, we determine without examining its individual vertices which constraints must hold within the group.

We next determine which constraints *cannot* hold within the transformational group. Again, this is easily done by examining the transformations that have been applied to the group. If a group has been translated, all of its fixed location constraints are broken. Fixed location constraints are also broken among vertices during rotations and scales if the vertices are not at the center of the transformation. Scales break all fixed distance relationships within a transformational group, and rotations break all fixed slope relationships within a transformational group.

After determining which relationships *must* hold within a group, and which *cannot* hold, we must consider the relationships that *might* hold. For each of these constraint relationships, we must examine the vertices in the group, looking for invariant relationships. Fortunately this expensive task need not be done for the most common transformations, translations, rotations, and isotropic scales, since all relationships in our constraint set can immediately be classified as either definitely present or definitely not present. The situation is more complex for the other less common affine transformations, and the vertices must be examined explicitly.

For every snapshot after the initial one, the first step is to fragment existing transformational groups into new ones, accounting for the transformations that have occurred since the last snapshot. The new child group has all the constraints of its parent, except those broken by the transformations performed since the last snapshot. Since we are only interested in effective transformations at the snapshot granularity level, we factor the composition of transformations applied since the last snapshot into scale, rotation, and translation components, and use these, as described above, in determining which intra-group constraints were broken. This allows us to ignore transformations that have been undone by subsequent operations between the two snapshots. For example, if a set of vertices is translated away from its original location, and then back again between snapshots, then those translations are effectively ignored.

To illustrate transformational groups, and several other algorithmic details discussed later, consider the two simple snapshots given in Figure 7. Two boxes were captured in the first snapshot (Figure 7a). Initially, all vertices were in the same transformational group, and constrained to have fixed locations. After this, but prior to the next snapshot (Figure 7b) the boxes were scaled together, and the right box was translated back to a position one large grid unit from the left box. The second snapshot split the original transformational group into two children, each containing the vertices of one of the boxes. The second snapshot broke the fixed location constraints for all vertices except the bottom left of the left rectangle, since this vertex's effective

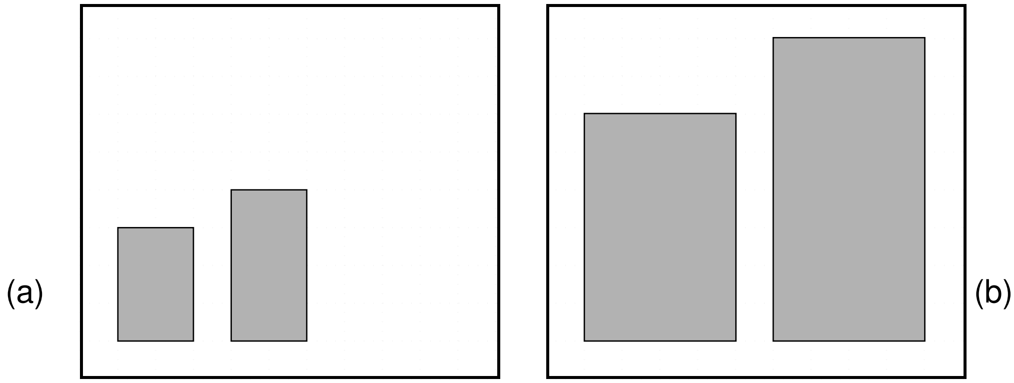

FIGURE 7. Two snapshots of a simple scene

transformation had no translational component, and its location was at the center of the scale. The intra-group fixed distance constraints were broken for each group because there was a net scale, but isometric scales maintain proportional distances, so an implicit relative distance constraint was added to each group. It is important to note that transformational groups are dependent upon the transformations performed, but they have no impact on the constraint set that will eventually be inferred. They accelerate the search process by pruning the search space.

3.2.3 Inter-Group Constraints

The next step is to compute *inter-group constraints*—constraints between different transformational groups or their vertices. These constraints are generated in several ways. They can be formed from a relative intra-group constraint when a transformational groups is split by a transformation that preserves the relation. Consider a transformational group with a relative slope constraint among all of its vertices. If the group is split in two by a translation or scale, then we must add a relative slope constraint between the two groups, relating the slopes contained within one group to the slopes within the other. Similarly, if a transformational group has a relative distance constraint among all of its vertices, and the group is split by a translation or rotation, then we need to add a relative distance constraint between the two groups, specifying that the distances within one group will remain proportional to the distances in the other.

We have just described *group-to-group* inter-group constraints—constraints that make entire groups rotate or scale with another. There are also *vertex-to-vertex* inter-group constraints, which express relationships between a small number of vertices. Finding these is the most costly step in our algorithm, but the cost is reduced by the observation that we only need to compute inter-group constraints between a child transformational group, its parent, and its siblings (other child groups of the same parent spawned during the same snapshot). Inter-group constraints between the child and other groups were already formed when their ancestors were split.

For small sets of vertices chosen from the newly created child group and its parent or siblings, we look for relationships that have not changed and generate absolute constraints for these when found. For example, we compute the slope and distance between such pairs of vertices at the current snapshot, and the previous snapshot. If either of these values are unchanged, we create an absolute constraint between the two vertices. Similarly, for each pair of lines constrained to have the same slope, that were contained in a single transformational group during the last snapshot, but are now split among groups, we identify absolute parallel distance constraints.

In our rectangle example, an inter-group vertex-to-vertex constraint inferred at the second snapshot declares that the inner segments be one large grid unit apart. This constraint was implicit after the first snapshot, when both rectangles were members of the same transformational group, but must be made explicit after the second snapshot since the relationship still holds after the transformational group was split.

3.2.4 Delta-Value Groups

Existing constraints between groups or vertices transformed since the last snapshot are now considered, and those that no longer hold are broken. Broken *relative* constraints, constraints relating geometric measures (such as slope) of more than one object, are split if possible into constraints that are still satisfied among fewer objects. Absolute constraints that have been broken during the current snapshot are matched, as is now described, to form new relative constraints.

We have already described how absolute inter-group constraints are found by locating relationships that do not change. One type of relative inter-group constraint is found by locating relationships that change together. If two pairs of vertices are constrained to have constant slopes, then there is no need for a relative constraint between the two, since the individual values are fixed. However, if these slopes now change by the same amount, it becomes necessary to create a relative constraint between them. Collections of relations that were absolutely constrained in a previous snapshot, but have broken by similar amounts in the current snapshot, are bundled together into *delta-value groups*.

Delta-value groups are simply relative constraints between arbitrary numbers of relations, and like transformational groups, they allow us to represent similar constraints among many objects compactly. For example, a delta-value group might constrain n distances to be proportional. However, when passing the delta-value group to the solver, it need only be expanded to $n-1$ binary constraints when solving the system (relating the first element to each subsequent element) rather than n^2 constraints relating each pair of elements.

Every absolute constraint broken in the current snapshot must be considered for inclusion in a delta-value group. There are three steps in our algorithm where broken absolute constraints are identified:

- During the fragmentation of transformational groups
- During the identification of vertex-to-vertex inter-group constraints
- During the breaking of constraints instantiated during previous snapshots

We place together in delta-value groups distance relations that change by the same proportion, and angle and distance relations that change by the same number of degrees. Since typically many absolute constraints break during the same snapshot, it is important to find matches efficiently. We employ hashing to match constraints that break by similar amounts, so this step is performed in linear time with respect to the number of broken constraints identified.

Returning to the example of Figure 7, the two rectangles are in separate transformational groups after the second snapshot. This snapshot broke absolute distance constraints for both of these groups, since they were scaled differently than their parent, which had an implicit absolute distance constraint among all of its vertices. Both of these absolute distance constraints broke by a factor of two. As a result, they were added to the same delta-value group, maintaining that distances in the two groups be proportional.

3.2.5 Redundant Constraints

We have now computed all of the constraints that are invariant among snapshots. When objects are transformed with constraints turned on, the inferred constraints are passed to our solver. Typically our constraint set contains a large number of redundant constraints—constraints derivable from others through geometric tautologies. To accelerate the process of finding a solution to the constraint set, we try to remove redundant constraints. There are two ways that this can be done, both of which involve looking for simple geometric relationships. The first looks for these relationships as a post-process, after the inferencing has been performed, and filters out extra constraints known to hold in those circumstances. This works well for those relationships that generate a constant number of redundant constraints. However, certain relationships yield a polynomial number of such constraints, and it is more efficient never to generate them.

These redundancies can be avoided by building additional kinds of groups during the inferencing process. As discussed earlier, transformational groups and delta-value groups allow large numbers of graphical relationships to be represented tersely. By identifying relationships that lead to redundant constraints, and classifying them as special groups, we can pass only the essential constraints off to the constraint solver. Figure 8 illustrates two relationships that are particularly useful to express as groups since they are common and yield many redundant constraints if fully expanded. In Figure 8a, vertices p and q are constrained to be coincident. If each other vertex r_i in the figure were part of a separate transformational group, our algorithm would instantiate the constraints distance(p, r_i) = distance(q, r_i), and slope(p, r_i) = slope(q, r_i) for all r_i. These redundant constraints can be avoided by building *coincident vertex groups* for sets of vertices currently constrained to be coincident. These groups are used in lieu of their actual vertices while computing inter-group vertex-to-vertex constraints. If vertices in the group are not coincident in a subsequent snapshot, the group is broken, and the formerly redundant constraints which still hold are instantiated.

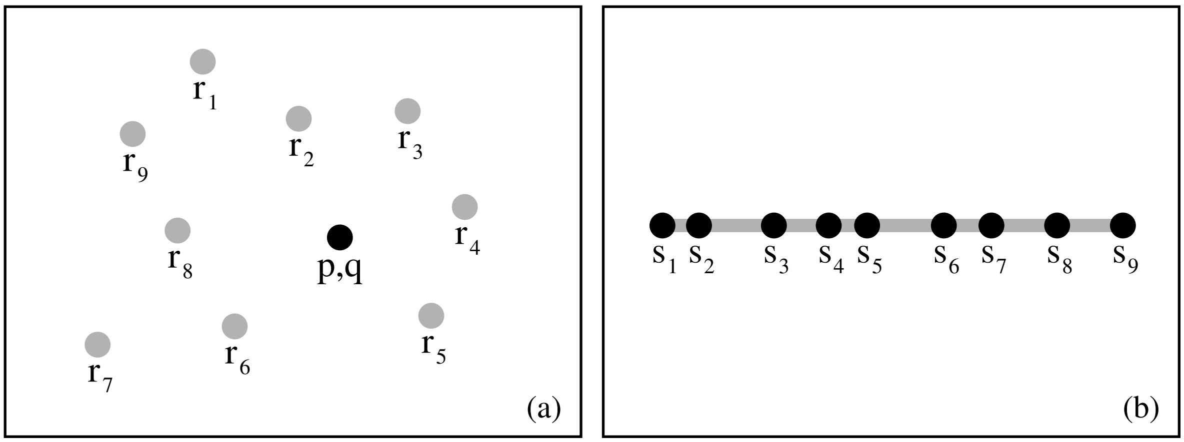

FIGURE 8. Two geometric relationships that lead to redundant constraints.

Another common geometric relationship leading to redundant constraints is shown in Figure 8b. Here, snapshots have resulted in a set of collinear vertices s_i, such that each vertex is in a separate transformational group, and the slope between each pair of vertices is fixed. Here, only n-1 constraints are necessary to represent the slope constraints between the n vertices, but the algorithm identifies constant slope constraints between each pair of vertices, s_i and s_j such that $i < j$. By identifying this relationship as a group during the inferencing process, we can avoid generating these redundant constraints.

Currently we look for only a few classes of redundant constraints, and often a large number eludes us. We are working on improving this component of our system.

3.2.6 Solving the Constraint System

When constraints are turned on and constrained objects transformed, we compute the effects on other objects in the scene. The constraint system can be viewed as a graph, with the nodes being vertices of scene objects, and the arcs being constraints between the vertices. Changes to one disconnected subgraph cannot affect another, since there are no constraints linking them. We find the disconnected subgraphs containing the vertices actively being transformed, by performing a simple graph traversal beginning at these vertices. Constraints which are not a part of any of these subgraphs cannot affect our solution and can be safely ignored. Also, since the constraints of different subgraphs are mutually independent, they are solved independently, thereby reducing the cost of the solution.

We also reduce the solution cost by using a simple generalization of the technique many constraint-based systems use to solve for rigid bodies efficiently. If a set of vertices are part of a transformational group, they are constrained to transform together under a restricted class of transformations, and often we can use this information to avoid passing certain constraints and vertices to the solver. A transformational group that has only been translated has absolute slope and distance constraints between each pair of vertices, and these same constraints insure that all vertices in the group will translate together. If some vertices in the transformational group participate only in these constraints, then instead of passing them to the solver, we can explicitly apply to these vertices the translation that the solver finds for other vertices in the group. Similar approaches can be taken for isometric scales, rotations, and compositions of these transformation classes.

As an example of this, consider Figure 9. The snapshots in Figure 9a and 9b constrain the hand to scale so that the lower left vertex of the wrist is fixed, and the right-most vertex of the index finger aligns with the arrow. All vertices of the hand are part of the same scale transformational group, and those of the arrow are part of the same translation transformational group. Figure 9c shows all

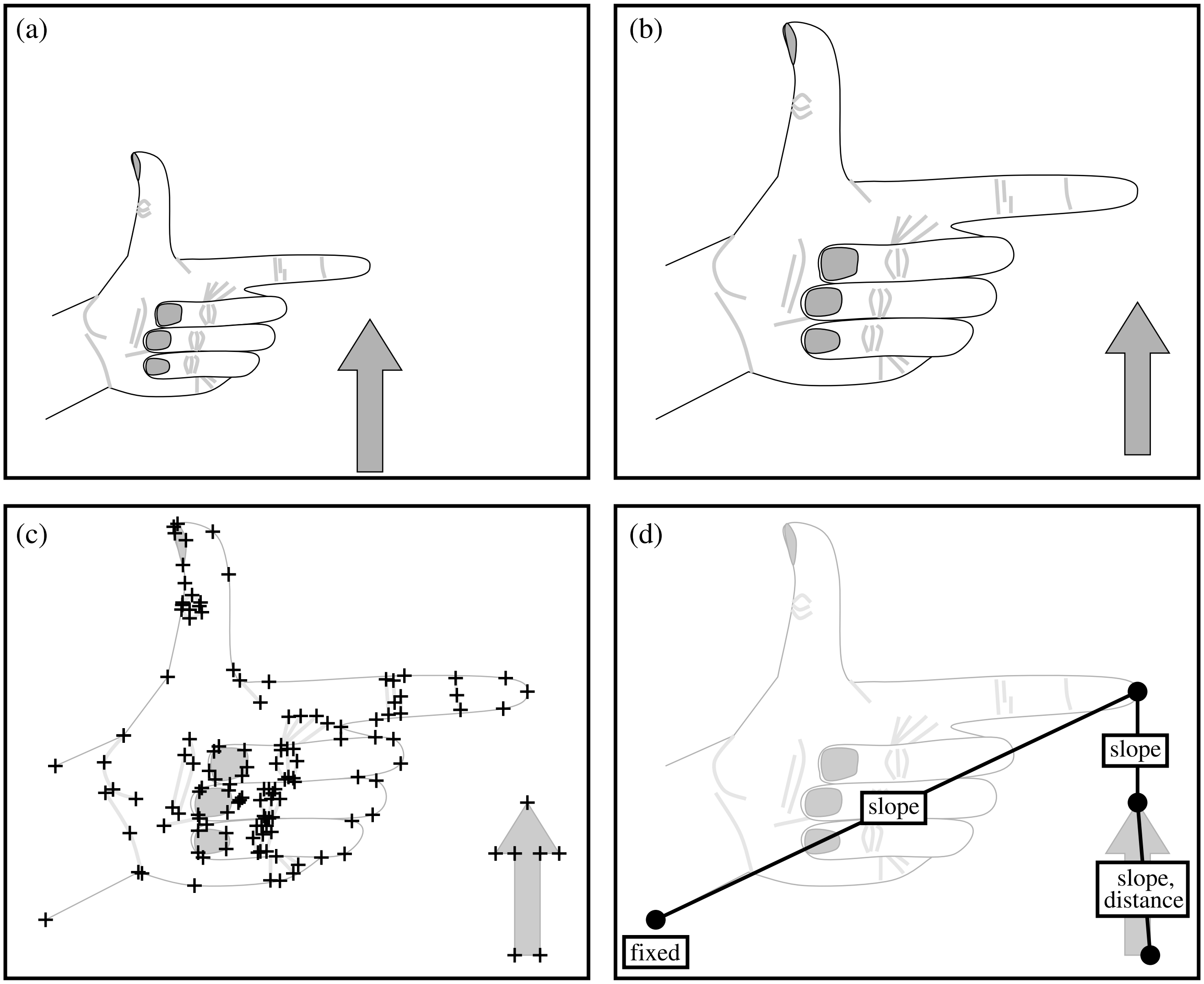

FIGURE 9. Efficient constraint formulations for transformational groups. Snapshots (a) and (b) constrain the scene. A naive approach solves constraints for all vertices marked in (c). A more efficient method solves only constraints shown in (d).

the vertices in the system that participate in the constraint solution. However, only a few of these 134 vertices must be passed to the solver.

In Figure 9d, we choose to translate the lower right vertex of the arrow. We begin traversing the constraint graph at this vertex to determine which constraints and vertices must be passed to the solver. This vertex will be passed to the solver, since it is being manipulated directly by the user.

The top vertex of the arrow must also be passed to the solver, since it participates in an inter-group slope constraint. These two vertices are bound together by absolute slope and distance constraints because the arrow is a single translation transformational group. The displacement of all the other vertices in this group will be determined by calculating the displacement vector that the solver finds for these points.

Similarly, the vertex at the tip of the index finger is passed to the solver, since it participates in an inter-group slope constraint with the point of the arrow. The lower left vertex of the wrist must also be passed to the solver, since it has a fixed location constraint. These two vertices of the hand are connected by an absolute slope constraint, because they are part of the same scale transformational group. The positions of all the other vertices in the hand are easily determined by the scale transformation that maps these two vertices to their new positions.

3.3 Algorithmic Complexity

In this section we analyze the complexity of generating a snapshot by considering the cost of each step of the algorithm. Forming the initial transformational group takes $O(n)$ steps, where n is the number of vertices in the scene. Fragmenting existing groups into new groups is $O(tn)$, where t is the number of transformations performed between snapshots. In typical editing, t is much smaller than n. The cost of forming new inter-group constraints from absolute intra-group constraints during the splitting process is linear with the number of splits being performed, and this is no worse than $O(n)$.

Typically the most computationally expensive step is finding inter-group constraints between sets of vertices. For both absolute slope and distance relationships, between arbitrary pairs of vertices, the cost of this is $O(n^2)$. Finding absolute angle constraints between arbitrary sets of three vertices would be $O(n^3)$, however angle constraints are rarely meaningful unless the vertices are connected, so we only consider such angles. Since vertices in our system connect no more than two lines, the task of searching for absolute angle constraints of connected vertices is O(n). We also look for absolute distance constraints between pairs of parallel lines, which is an O(n^2) task.

The process of weakening or breaking existing constraints is linear with respect to the number of constraints in the system, which is no more than O(n^2). Forming delta-value groups is linear with respect to the number of newly broken absolute constraint relationships, which from the above discussion is also O(n^2). The step of traversing the constraint graph, finding relevant constraints, is linear with respect to the number of constraints in the system. The entire inferencing algorithm, as discussed so far, is O(n^2) with respect to the number of vertices in the scene. The only step that we have yet to analyze is that of removing redundant constraints. This step might be a bottleneck if we were to filter out all redundant constraints in the scene, but our experience suggests that the majority of redundant constraints can be filtered out within the O(n^2) bounds of the rest of the algorithm.

3.4 Parameterizing an Illustration

Often it is convenient to be able to parameterize graphical illustrations. A slight modification to the algorithm described above allows simple relationships between scene objects and parameters to be inferred during the snapshot process. We provide an Arguments window in which scalar values can be typed as the illustration is edited into new configurations. These values are interpreted by the algorithm as though they were distances, slopes, and angles between vertices. If one of the changing geometric relationships in the scene matches a changing numeric argument, a relative constraint is created between the two values.

In Figure 10 we have drawn a scrollbar in the Chimera editor, and we would like to equate the percentage typed in the Argument 1 field of the Arguments window to the height of the scrollbar's slider. We constrain the scene by providing the two snapshots depicted in Figures 10a and 10b, but in addition to presenting two valid versions of the scene's geometry, we type corresponding

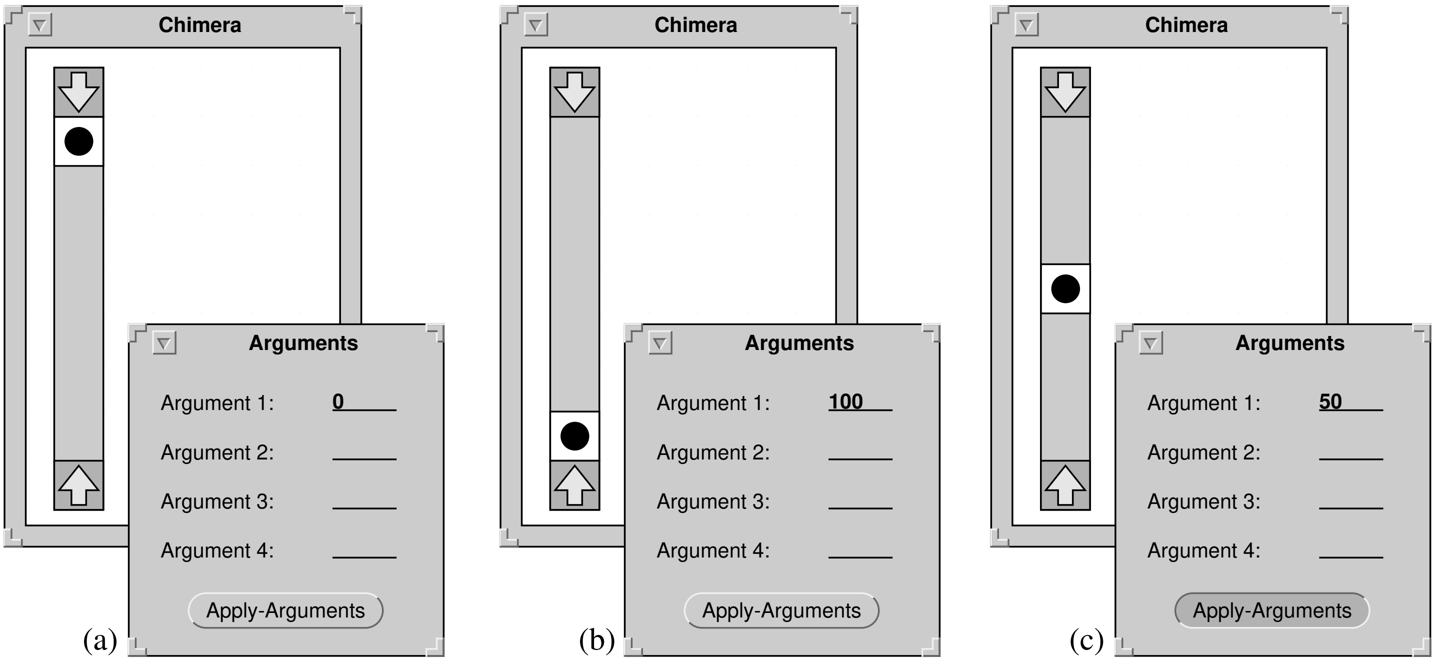

FIGURE 10. Dimensioning the height of a scrollbar. Initially two snapshots, (a) and (b), are specified. A new value for Argument 1 is entered in (c), and the scrollbar adjusts automatically.

values in the Argument 1 text field. As shown in Figure 10c, after turning constraints on, we can adjust the scrollbar's slider by editing the value in this same text field, and pressing the Apply-Arguments button. Alternatively, we can adjust the slider, and the value of Argument 1 changes accordingly.

Myers presents a similar example of parameterizing scrollbar behavior in [Myers88]. His method linearly interpolates between two different constrained configurations, which is a more powerful abstraction, particularly for defining the behavior of widgets. For example, in Peridot the slider height can be parameterized with respect to the bottom and top of the scrollbar. This cannot currently be done in our system. In our example, Argument 1 is interpreted as proportional to the distance between two parallel lines—the top of the slider box and the bottom of the box containing the upper scroll arrow. So if the scrollbar is resized, the percentage parameter will no longer range from 0 to 100. Peridot's constraints were chosen for the domain of widget construction, and

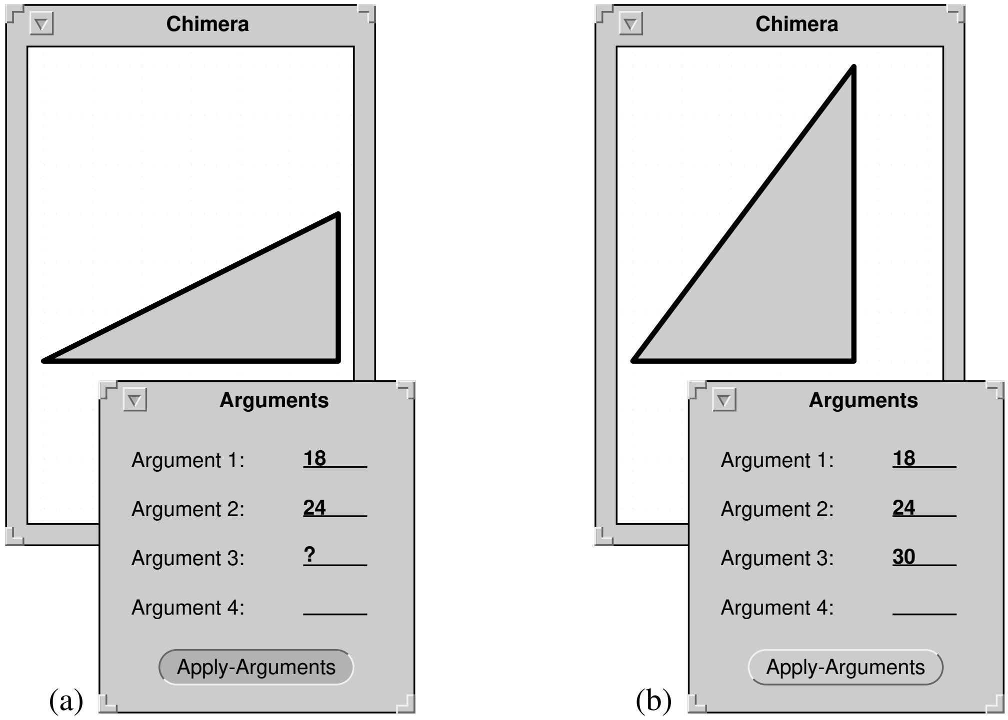

FIGURE 11. Specifying a subset of the parameters. Only the first two parameters are specified in (a). The triangle resizes, and a value is computed for the third parameter in (b).

are specialized for this type of task. Our system provides a lower-level constraint set for the construction of general illustrations. The type of parameterization that our system provides is useful for many basic illustration tasks, such as the dimensioning of distances, slopes, and angles.

Since parameters of the illustration can be mutually dependent, the values of a subset may determine the rest. Sometimes the user may care to set only a few of the available parameters. For these reasons, we allow parameters to be either be either specified or unspecified. Specified parameters are constrained to their current value during the constraint solution, but unspecified parameters are allowed to vary. Figure 11a shows a Chimera editor scene containing a single triangle. Two previous snapshots (which are not shown) have constrained it to be a right triangle, with a fixed lower left corner, and horizontal base. They also have constrained Argument 1 to be proportional to the length of the base, Argument 2 to be proportional to the length of the vertical segment, and Argument 3 to be proportional to the hypotenuse's length. In Figure 11a, we type the desired lengths of each of the sides but the hypotenuse into the Arguments window. The question mark entered for Argument 3 requests that it be chosen by the constraint solver. After the Apply-Arguments button is pressed, the triangle resizes subject to its constraints, and Argument 3 is filled in with a suitable value.

4.0 Implementation

The Chimera editor is implemented mainly in Lucid Common LISP and CLOS (the Common Lisp Object system), with some C code as well. Our constraint solver is implemented in C, but the inferencing mechanism is in LISP. The code runs on Sun workstations under OpenWindows.

We use Levenberg-Marquadt iteration [Press88] to solve the constraint systems. This method uses gradient descent when far from a solution, but switches to the inverse Hessian method to converge quadratically when a solution is near. Levenberg-Marquadt is a least-squares method. Each constraint is implemented as an error function, and the algorithm finds the best solution to a set of error functions according to a least-squares evaluation, provided it does not fall into a local minimum. The functions are not limited to be linear, or even algebraic. If the constraint solver cannot find an acceptable solution the user is notified of this, and they then have the option of undoing the operation, or trying to coax the system out of a local minimum by further manipulating the graphical objects. We would eventually like to add multiple constraint solvers, so that when one fails to find a solution, another can be invoked.

Part of the Levenberg-Marquadt method requires solving a system of equations to determine how the current solution estimate should change. If the error functions and their partial derivatives are not mutually independent (which is the case with redundant constraints), the system cannot be solved using Gaussian elimination. Instead, we use singular value decomposition [Press88] to find a solution at this step.

In looking for absolute and relative relations in the scene, it is important to build a tolerance into the matching process. We use a small, fixed, empirically-derived tolerance, just large enough to account for floating point inaccuracies during the construction and editing of the scene. If the tolerance were large, the number of incidental constraints would increase. Our small tolerance requires that the snapshots be drawn accurately, so we provide both grids and snap-dragging for this purpose.

Both the inferencing algorithm and constraint solver typically run at interactive speeds, on a 1.4 MFLOP Sun SparcStation 1+, for systems of the size presented in the paper. The slowest snapshot (that of Figure 4b) took about 3 seconds. Constraint solutions were obtained in under a second in all cases but the window resizing example. This took somewhat longer because a large number of redundant constraints were passed to the solver by the inferencer. Further work on the inferencer should reduce the number of redundant constraints, and speed up constraint solutions.

5.0 Conclusions and Future Work

Snapshots appear to be a very intuitive way of specifying constraints, and often allow complex constraint systems to be specified with relatively few operations. However, we will not know until performing user-trials whether, and under what conditions, people prefer the technique to traditional declarative specification. Our personal experience with snapshot constraint specification in Chimera suggests that it is not a panacea. There are certain tasks for which it appears to be a simpler, more natural method of constraint specification, but for others, traditional declarative specification remains easier. There are a number of problems using snapshots that the traditional method does not have:

- Certain pictures can be difficult to edit into new configurations. In some of these cases it may be easier to specify constraints explicitly.
- Incidental, unintended relationships often occur in large scenes, necessitating extra snapshots.
- Redundant constraints are commonly passed to the solver, increasing solution costs.

When it is easier to specify constraints declaratively than by example, then the declarative technique should be used. We have built a traditional declarative constraint interface for our editor that is useful in these cases, and will allow us to better compare the two methods.

Incidental constraints can be reduced by restricting the classes of constraints that can be inferred. In our initial implementation, we inferred inter-group relative distance and slope constraints between any two pairs of vertices. This resulted in too many incidental constraints, so we restricted these constraints to pairs of two connected vertices (although there need not be a connection between the pairs). We still infer absolute distance and slope constraints between any two vertices, and intra-group relative distance and slope constraints between any two pairs of vertices. We are looking for additional restrictions that will not significantly impair the utility of the system.

Another way to reduce incidental constraints is to have the user select a set of objects prior to the beginning of a snapshot sequence, and have the inferencer look for constraints only among these objects. Partitioning the scene has the additional benefit of accelerating snapshots. In traditional constraint specification, constraints are also often added in partitions, to speed up solution. Currently we do not allow inferencing to be restricted to a subset of the scene, but this option is important for large scenes, and we plan to include it in the future.

One approach to reducing redundant constraints might involve using algorithms similar to those Chyz developed for maintaining complete and consistent constraint systems [Chyz85]. When a new constraint is added to the network, his algorithms determine which constraint must be eliminated to avoid overconstraining the system. These methods may allow us to reduce the set of constraints passed to the solver. However, we would not filter out most redundancies from our master constraint set, since after subsequent snapshots they may no longer be redundant.

There are a number of other interesting topics for future work. We would like to extend our system to handle constraints between non-geometric quantities, such as color or font. Animating the constrained systems would provide an intuitive display of the set of constraints inferred, in the same visual language as the snapshot specification. We would like to provide an audit trail of snapshots by incorporating them into our graphical edit history representation [Kurlander90]. This will allow individual snapshots to be eliminated and the constraint network recalculated.

It would be helpful to infer a few additional geometric relationships, such as distance between a vertex and a line, or the angle between two arbitrary lines. These constraints could be easily added to the inferencing algorithm. Currently we infer constraints only among vertices in the initial drawing. There are cases when we would also like to infer relationships among implied objects, such as the center of a rotation, or the bounding box of an object. We also plan to allow constrained shapes inferred by our technique to be parameterized in more complex ways, and included as part of macros.

Acknowledgments

We thank Terry Boult for valuable advice on numerical techniques, and Larry Koved and Dan Ling for several helpful discussions. Eric Bier suggested very good background material. Eric Bier, Michael Elhadad, and Ken Perlin provided useful comments on earlier drafts. Initial development of Chimera was facilitated by an equipment grant from Hewlett-Packard. David Kurlander was funded during this research by a grant from IBM.

References

[Bier86] Bier, Eric A., and Stone, Maureen C. Snap-Dragging. Proceedings of SIGGRAPH '86 (Dallas, Texas, August 18-22, 1986) In *Computer Graphics 20,* 4 (August 1986). 233-240.

[Bier88] Bier, Eric A. Snap-Dragging: Interactive Geometric Design in Two and Three Dimensions. Ph.D. Thesis. U.C. Berkeley. EECS Department. April 1988.

[Borning79] Borning, Alan. ThingLab: A Constraint-Oriented Simulation Laboratory. Xerox PARC Tech Report, SSL-79-3. Revised version of Stanford PhD thesis. July 1979.

[Borning86] Borning, Alan. Graphically Defining New Building Blocks in ThingLab. *Human Computer Interaction 2,* 4. 1986. 269-295. Reprinted in *Visual Programming Environments: Paradigms and Systems*. Ephraim Glinert, ed. IEEE Computer Society Press, Los Alamitos, CA. 1990. 450-469.

[Chyz85] Chyz, George W. Constraint Management for Constructive Geometry. Master's Thesis. MIT. Mechanical Engineering. June 1985.

[Cohen82] Cohen, Paul R., and Feigenbaum, Edward A. *The Handbook of Artificial Intelligence*. vol. 3. Kaufmann, Inc., Los Altos, CA. 1982.

[Ellman89] Ellman, Thomas. Explanation-Based Learning: A Survey of Programs and Perspectives. *ACM Computing Surveys 21,* 2. June 1989. 163-221.

[Kurlander90] Kurlander, David and Feiner, Steven. A Visual Language for Browsing, Undoing, and Redoing Graphical Interface Commands. In *Visual Languages and Visual Programming*, Shi-Kuo Chang, ed. Plenum Press, New York. 1990. 257-275.

[Kurlander91] Kurlander, David. Example-Based Techniques for Graphical Editing. Ph.D. Thesis. Columbia University. Computer Science. In preparation.

[Lee83] Lee, Kunwoo. Shape Optimization of Assemblies Using Geometric Properties. Ph.D. Thesis. MIT. Mechanical Engineering. December 1983.

[Maulsby89] Maulsby, David L., Witten, Ian H., and Kittlitz, Kenneth A. Metamouse: Specifying Graphical Procedures by Example. Proceedings of SIGGRAPH '89 (Boston, MA, July 31-August 4, 1989) In *Computer Graphics 23,* 4 (July 1989). 127-136.

[Myers86] Myers, Brad A., and Buxton, William. Creating Highly Interactive and Graphical User Interfaces by Demonstration. Proceedings of SIGGRAPH '86 (Dallas, Texas, August 18-22, 1986) In *Computer Graphics 20,* 4 (August 1986). 249-268.

[Myers88] Myers, Brad A. *Creating User Interfaces by Demonstration.* Academic Press, Boston, 1988.

[Nelson85] Nelson, Greg. Juno, A Constraint-Based Graphics System. Proceedings of SIGGRAPH '85 (San Francisco, CA, July 22-26, 1985) In *Computer Graphics 19,* 3 (July 1985). 235-243.

[Olsen90] Olsen, Dan R., Jr., and Allan, Kirk. Creating Interactive Techniques by Symbolically Solving Geometric Constraints. Proceedings of UIST '90 (Snowbird, Utah, October 3-5, 1990) 102-107.

[Pavlidis85] Pavlidis, Theo and Van Wyk, Christopher J. An Automatic Beautifier for Drawings and Illustrations. Proceedings of SIGGRAPH ‘85 (San Francisco, CA, July 22-26, 1985) In *Computer Graphics 19,* 3 (July 1985). 225-234.

[Press88] Press, William H., Flannery, Brian P., Teukolsky, Saul A., and Vetterling, William T. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, 1988.

[Sutherland63a] Sutherland, Ivan E. SketchPad, A Man-Machine Graphical Communication System. Ph.D. Thesis. Electrical Engineering. January 1963.

[Sutherland63b] Sutherland, Ivan E. SketchPad: A Man-Machine Graphical Communication System. AFIPS Conference Proceedings, Spring Joint Computer Conference. 1963. 329-346.