Creating and Manipulating Constrained Models

Michael Gleicher
Andrew Witkin

January 8, 1991

CMU-CS-91-125

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The success of constraint-based approaches to geometric modeling has been limited by difficulty in creating constraints, solving them, and presenting them to users. This paper addresses all three issues. We facilitate the creation of constrained models by using placement operations to specify constraints as well as positions, with no extra work on the part of the user. We augment *Snap-Dragging*[Bie89], an earlier successful technique for specifying geometric models, to give the user the option of making persistent the relations it helps create. Because the technique provides both the constraints and an initial solution to them, we can use the techniques of *constrained dynamics* to maintain the relationships as a user drags the models. This *differential approach* to constraints avoids many of the difficulties in constraint-based systems, such as solving non-linear algebraic equations. Our approach suggests methods for displaying and editing constraints as well.

This research was sponsored in part by Apple Computer, Siemens Corporation and Silicon Graphics Incorporated. The first author is supported in part by a fellowship from the Schlumberger Foundation, Inc. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Schlumberger, Siemens, SGI or Apple.

Keywords: Interaction Techniques, Computational Geometry and Object Modeling, Computer Aided Design

1. Introduction

Geometric models contain shape and position information, but typically do not explicitly represent geometric relationships among parts. Constraints expressing these relationships are an attractive addition to geometric models. The process of editing a model can be facilitated by maintaining the constraints, saving the user the effort of re-establishing the relationships. Constraints in a model can also be used to simulate its motion or to control it in an animation. For reasons such as these, constraint-based approaches have enjoyed success in several domains, such as mechanical design[LGL81, Sap89, Ser87], animation[IC87, WK88, WW90], and user interface specification[BD86, MGD+90, Hil91].

Unfortunately, creating constrained models is an error-prone process which can involve significantly more work than creating a scene without constraints. Since both an initial configuration of the model and the constraints are required, the user must effectively specify the relations twice. The domains in which constraint methods are successful are those in which the user can expect the extra effort expended in specifying the constraints to be worthwhile. Our goal is to make constraint techniques viable in a broader class of applications by reducing the extra effort required to specify constraints, and by making it easier to manipulate the resulting models.

In this paper, we will show how tools which aid users in positioning objects can also convey the underlying relationships in the scene. We augment *Snap-Dragging*[Bie89, BS86], a technique which makes it easy for users to create unconstrained models rapidly, to specify not only the immediate geometry of a scene but also the underlying constraints, with no extra work on the part of the user. As originally presented, Snap-Dragging provides methods for helping users to establish precise relationships in scenes, but provides little help in keeping these relationships met as the user subsequently edits the scene. By making these relationships persistent, we can use constraint techniques to maintain them during dragging so that the user does not need to re-establish them after each editing operation.

Constraint techniques have been used in computer graphics for a long time[Sut63, Bor79]. Typically, constraint-based systems use an algebraic equation solver to bring a model from a state that does not satisfy the constraints to one that does. However, by adopting a differential approach to constraints, we need only maintain constraints that have already been established. In principle, the model is never in an illegal configuration.

We permit the user to manipulate a constrained model like an unconstrained one by dragging it with continuous motion. This motion can be described by a system of ordinary differential equations with algebraic constraints. Techniques from physics provide a way of converting these constrained differential equations into unconstrained ones by solving systems of linear equations, even if the constraints are nonlinear. These *constrained dynamics* techniques are well suited to manipulating geometric models, and in this paper we adapt them further to the challenges of geometric modeling.

The differential approach has several advantages over one which relies on jumping to legal configurations. Not only does it avoid the difficult problem of solving non-linear algebraic equations, but it also makes it easier to respond to issues in presenting constraints to users. This paper addresses many of the issues in constraint-based systems, including creating, editing, and comprehending constraints.

The successful marriage of Snap-Dragging and constraint techniques depends on careful attention to user interface principles. By strictly adhering to some of the most applicable rules, such as providing adequate feedback, we are able to address many of the issues that created problems for previous systems.

2. Creating Constrained Models

Ordinarily, when the user of a drawing program places an object, the system cannot know why it was placed where it was. However, the use of a positioning aid can provide more information than just the position of the object. Typically, this information is not exploited by the program. Our approach to creating constrained models is to remember the relations that these drawing operations specify, providing the user with the option of making them persistent.

In this section, we will describe how we augment drawing tools so that they specify the underlying relationships, in addition to positions. We survey the work on positioning tools to introduce Snap-Dragging, an earlier technique for establishing relations in drawings. We then show how the positioning operations of Snap-Dragging map neatly to constraints. The two issues in making the scheme work smoothly, handling ambiguity and avoiding redundant or degenerate constraints, are then addressed.

Attention to user interface issues[Nor90] is important in making the constraint generation process work well. In this work we need to pay particular attention to those principles which might be described collectively as "no guessing" rules. The user's actions must be unambiguous so that the system does not need to guess the user's intent. Conversely, the system must provide sufficient feedback so that the user does not need to guess what the system is doing.

2.1. Tools for Establishing Relations

When drawing, it is difficult to position a pencil precisely without using some form of aid. Unaided, it is similarly difficult to draw precisely with a mouse or other pointing device. Computer software can provide tools for precise placement by drawing from a software-positioned cursor[1] rather than using the pointer location. The software cursor's location is influenced by the position of the pointer, but determined by a function which helps the user position elements precisely.

The uniform grid is the most common function for mapping pointer location to cursor position. In such a scheme, all drawing operations are displaced to points on an equally spaced rectangular grid. This aids in establishing inter-object relationships only indirectly: if two points appear close to each other, then they must be in the same location.

"Gravity" is another cursor positioning function. When the pointer is brought sufficiently close to an interesting element in the scene, the cursor snaps to it. The idea of gravity has existed for a long time, having been demonstrated as early as Sketchpad[Sut63]. It allows many relationships in drawings to be established easily.

Snap-Dragging[Bie89, BS86] enhances the usefulness of gravity. The cursor snaps not only to the edges of objects, but also to interesting points in the scene such as intersections and vertices of objects. The ability to snap to intersections enables the use of traditional drafting compass-and-straightedge constructions.

Relations other than contact are created in Snap-Dragging through *alignment objects,* objects that are not part of the drawing *per se,* but exist only to be snapped to. The original Snap-Dragging work includes several types of alignment objects, each corresponding to a type of relationship which is useful in drawings.

[1] In Snap-Dragging terminology[Bie89], the position of the hardware pointing device is known as the cursor and the software cursor is known as the caret.

For example, distance from a point can be specified by placing an alignment circle around that point. The usefulness of alignment objects is further enhanced by making them easy to place. In fact, they can often be placed where needed automatically.

Snap-Dragging's system of gravity and alignment objects allows a user to rapidly establish precise relationships. However, once made, these relationships are immediately forgotten. It is the user's responsibility to re-establish relationships that are disrupted during subsequent editing. Like most techniques which aid users in establishing relations in drawings, Snap-Dragging does not address the issue of creating constrained models.

Ours is not the first attempt to spare users from the additional effort required to explicitly specify constraints. Previous systems have attempted to infer relationships after drawing operations by looking at the resulting drawing[PW85], or at a trace of user actions[MKW89]. Because this information typically does not specify the relationships unambiguously, these systems relied on heuristics or asked the user[MB86] to resolve the ambiguity. Our approach provides positioning methods which unambiguously specify constraints, eliminating the need for inferences.

In this paper, we have chosen to augment Snap-Dragging to make the relations it specifies persistent. The techniques are not specific to Snap-Dragging, but should work with many schemes which help the user create relationships among objects in drawings. For example the "Graphic Guides" in Claris CAD [Cor89] are a candidate for these techniques.

2.2. Relations from Drawing Operations

Our basic idea is that placement operations contain information about why an object was positioned where it was. Suppose the user, while dragging an object, moves the pointer near another object so that the cursor and the point being dragged snap to the second object. This might have been an accident, but the user might have been trying to achieve this relationship. We provide the user the option of making the relationship persistent, so if it was intentional it can be preserved during subsequent editing.

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

The two simple snapping operations form the basis for establishing a wide variety of relationships. The simple mapping from them to constraints similarly extends to a variety of constraints. For instance, snapping to an intersection is a conjunction of the simpler constraints. More complicated relationships are established by snapping to alignment objects. For example distance-from-point constraints are created by snapping to an alignment circle. The snapping operations can be easily translated into these constraints. However, the language provided by the basic operations and alignment objects can be used not only for specification but also for representation and visual display of many types of constraints (figure 1).

After a snapping operation has taken place, we provide the user with the option to make the relationship persistent. When a new relationship is established, the system acknowledges it. The user can accept it to make it persistent or ignore it. If the automatic constraint generation process is good, the user will want to accept most constraints so the option of making this the default should be provided. In such a mode, it must be easy for the user to say that an action was an accident.

After adding persistence to Snap-Dragging, there are still some issues remaining in building a system which creates constrained models. We will describe how we can be confident that we are obtaining the correct constraints and avoiding redundant, degenerate, and unsolvable constraints.

2.3. Handling Ambiguity

We only generate constraints which are unambiguously specified by the users actions. Therefore, it is important that the actions we provide to users can't unambiguously specify the desired relationships.

Snapping operations unambiguously specify a relationship between the point dragged and the point or object snapped to. However, if multiple objects coincide, there is ambiguity in which to snap to. If we are not remembering the snap relations, the ambiguity is irrelevant: only the target location is important. With persistent relations this distinction becomes significant. If the two coincident points are later separated, the correct attachment relationship must be maintained.

Feedback, along with Snap-Dragging's cycling mechanism, solves the problem of ambiguity in the thing snapped to. Our feedback mechanisms (figure 2) clearly show the user what is being snapped to. If this is incorrect, the user can click the cycle button and the system will snap to the next in the the list of objects within gravity range.

Cycling resolves ambiguity, but can be awkward. By choosing the semantics of constraints properly, we can greatly reduce the amount of cycling required. For example, instead of remembering equality relations, we keep connected points as an equivalence class. Connecting to one is the same as connecting to another, so they need not be distinguished. Similarly, if two alignments become constrained to be equivalent, they can be merged.

A related problem is that the user might construct a model in a manner which does not convey the desired constraints. As an example, consider the equilateral triangle construction (figure 3) which is often used to demonstrate Snap-Dragging[Bie89, BS86, Whi88]. In this example, alignment circles of $3/4$ inch are used to create a triangle whose sides are all of equal length. The user has specified a triangle with all sides equal to $3/4$ inch, not a triangle whose equal sides can be scaled as well as rotated and translated. The program only knows what the user has specified and, therefore, cannot guess another option.

We manage this problem by keeping the automatic generation process predictable, using ample feedback so the user is reminded exactly what the system has been told, and making it as easy as possible for users to convey the desired relations directly. Since our goal is to extract constraints without extra work, it is wrong to require users to expend extra effort to use constructions which create the correct constraints. Therefore, we must make it as easy as possible for the user to convey what is really intended. For example, the correct equilateral triangle could have been created not by creating sides of $3/4$ inch, but by using a measurement tool to create circles of radius equal to the length of the first line segment drawn.

As we find relationships that are needed in drawings but are difficult to express directly with current tools, we can develop new tools to expand the set of what can be said easily. By expanding the vocabulary of what can be expressed easily, a wider assortment of tools only makes modeling easier to a point, after which the larger number of commands becomes unmanageable. Fortunately, Snap-Dragging shows that a small number of tools can express most of the relationships found in drawings. It is unlikely that many new tools will be required.

2.4. Redundant, Degenerate, and Unsolvable Constraints

Constraints which have no solutions or require the drawing to degenerate (for example, collapse to a point) should be avoided. Similarly, our numerical methods tolerate redundant constraints, but such constraints should be avoided as they add unnecessary complexity to models and make the problem less numerically well-conditioned. Since these problems often plague constraints specified manually by users, there is cause for concern about them in automatically generated constraints as well.

Redundancy can be limited by filtering the set of objects which the cursor can snap to. Snapping creates relations among objects. If a relationship between the point dragged and an object is already known to be persistent, there is no point in snapping to this object. It can only cause a nuisance such as creating a redundant constraint or forcing the user to cycle. As an example, when dragging a point on an object, the part of the dragged object is always close to the cursor, but snapping to it is unnecessary; the point dragged is already connected to it. Unfortunately, in a complicated constrained system, it might require a difficult geometric proof to show that such a relationship already exists implicitly.

Recognizing that a relationship already exists is only an optimization. Failure to recognize one might require the user to do an extra cycling operation or create a redundant constraint. From our experience, it appears that handling the simple cases—not snapping to the object being dragged, checking for existing connection constraints, basic geometric identities, etc.—filters out the vast majority of redundant constraints.

In the original Snap-Dragging work, redundant snaps are avoided by not permitting the cursor to snap to moving objects[Bie89]. Unfortunately, in the presence of constraints this is an unacceptable solution. When dragging a constrained object using the techniques of the next section, a large number of objects can potentially be moving, not just the one that is connected to the mouse. It can be difficult for the system to prove that an object will not move and for the the user to to predict what the system knows will not move.

The automatic generation process also makes it hard to create constraints which cannot be solved or which require the drawing to degenerate in order to be met. Since the user supplies a solution along with the constraint, we know that there exists at least one solution. Constraints must be made by snapping, so that if the constraint cannot be met, the pieces will not reach together and will not snap. If the constraints force the drawing to degenerate, the user would have had to provide this solution.

An important corollary to this, as we will see in the next section, is that we always not only get the constraints, but also an initial solution to the constraints.

3. Manipulating Constrained Models

In the previous section, we presented a technique which allowed users to specify constrained models. This section considers how to use them in model editing. We provide a method to drag constrained models with continuous motion just as unconstrained models are dragged. Because we are provided with an initial legal configuration of the model, the constraint problem becomes a differential one of maintaining the relationships as the parts move continuously.

We begin this section by considering the limitations of previous algebraic-solving constraint approaches. We then describe the differential approach to constraints and we show how this approach avoids these problems. We conclude this section by showing how we have addressed some of the additional issues of constraint-based systems in our programs.

3.1. Algebraic-Solving Constraint Systems

In most previous constraint-based systems, beginning with Sketchpad[Sut63], the user specifies an initial configuration of the scene and some relationships on the scene which should hold. The system of algebraic equations which results from the constraints is then solved for a legal configuration. We call such approaches *algebraic-solving* approaches since they use an algebraic equation solver to move to a legal configuration from a state where the constraints are violated. Manipulation by dragging parts of the model is possible, but if constraints are violated, the system must again find a legitimate configuration for the model.

The algebraic-solving approach has three serious drawbacks which arise from the need to jump from an illegal configuration to a legal one: (1) solving non-linear systems of algebraic equations is hard problem; (2) the system must often guess which legal state to jump to; and (3) it can be very difficult for the user to understand the behavior of an algebraic-solving style system. We now consider these problems.

Geometric relationships typically lead to nonlinear equations. Therefore, finding a configuration which satisfies these relationships requires solving a system of nonlinear algebraic equations. There are no good general techniques for solving nonlinear systems, in fact, [PFTV86] argues no such method can exist. Systems may restrict the class of constraints they can handle to those which yield systems of equations which are solvable, for example to linear[Li88] or near-linear equations[VW82, Lel88], or require the user to specify a method for solving the constraints, such as a geometric construction[FP88, NKK$^+$88]. However, without these restrictions systems must resort to numerical searches[Nel85, LGL81] to solve the nonlinear algebraic systems.

The solution to the algebraic equations will most likely not be unique, since certain relations do not lend themselves to being expressed with constraints and not all parameters are determined by precise relationships. Because it is difficult to determine when the scene is exactly and uniquely specified by constraints so forcing a user to create such scenes is liable to lead to redundant and conflicting constraints. Therefore, constraint-based programs must be able to deal with underconstrained cases.

Underconstrained systems pose a problem for algebraic-solving constraint approaches. When the program is searching for a legitimate state from an illegal starting point, the system must guess which of the many possible solutions the user wants. Systems can employ heuristics, such as minimizing distance to the initial configuration [Whi88], or allowing the user to specify optimization criteria[PEWW90], but users still must be prepared to deal with incorrect guesses by attempting to either constrain away extra degrees of freedom or try new starting conditions for the solver.

In an algebraic-solving approach, it can be difficult for the user to understand the behavior of the constrained model. Before solving, the system must help the user predict what will happen when solving occurs. After the system has jumped to a new state, the program must help the user correlate the new and old model and figure out why the program chose that solution.

The differential approach to constraints that we use in this paper allows us to use techniques akin to those used in computer animation. In animation, constraints are popular for holding models together as they move. This is often done by simulating the behavior of physical models [WW90, IC87, Pla89, BB88]. The advent of high performance graphics workstations have created an interest in developing interactive simulation techniques[WGW90, SZ90, GW91b], and applying the techniques to domains other than physical objects[GW91a], such as drawings.

3.2. Differential Constraints

Because our techniques provide us with a model that initially meets the constraints, the problem of manipulation becomes the differential problem of maintaining the constraints. With a differential approach, the model is always in a legal configuration: the system never needs to search for a legal configuration starting from an illegal one.

Manipulating differential constraints is analogous to placing pieces of string on a table. Imagine trying to lay out strings to create a design which has several connection constraints. If the strings are arranged so they meet the constraints, they can then be tied together so these constraints remain met as they are moved. The possible configurations consistent with the constraints can easily be explored by pulling the strings around. In contrast, a close analogy to an algebraic-solver constraint approach would be to call a genie who would take a list of constraints and re-arrange the strings for us. If the strings are subsequently moved, we must call the genie again.

The strings-on-table analogy hints at the technique which we use to maintain differential constraints. We use the methods of physical simulation, although applied to simplified models. We treat the geometric entities as physical objects and the constraints as mechanical connections[GW91a].

The continuous motion of a constrained geometric model during dragging is described, as mechanical systems are, by a constrained differential equation. The method of Lagrange Multipliers[Gol80, Pla89, WGW90] provides a technique for converting this to an unconstrained differential equation by solving a linear system, even if the constraints are nonlinear. There are many good techniques for solving the resulting ordinary differential equations[PFTV86].

The Lagrange Multiplier technique is well described elsewhere in the graphics literature[WGW90, GW91a, Pla89]. Briefly, if an external force, such as the user pulling, is applied to one part of the model, the Lagrange Multipliers compute the forces transmitted to other parts of the model. This computation is accomplished by projecting the applied force into the space of forces that will not cause the constraints to break.

Constrained dynamics provides a useful mechanism for manipulating a constrained model. The user can drag parts around so the legal configurations of the model can be explored. Combined with adherence to our "no guessing" user interface rule which requires unambiguous user operations and sufficient feedback from the system, constrained dynamics allows us to respond to many of the issues in constraint-based systems, as we will now describe. Three difficult issues in algebraic-solving constraint systems, solving, guessing and comprehending, are avoided by the differential approach. Solutions are provided for the issues of displaying and editing constraints. We also show how our approach addresses the issue of scaling, a concern in constraint-based systems.

3.3. Avoiding the Algebraic-Solving Issues

When applying a differential approach, the use of constrained dynamics techniques allows us to avoid the difficult problem of solving non-linear algebraic equations. In this section we show why it also avoids the two other problems since it handles underconstrained systems and facilitates comprehension.

Because it is provided with an initial solution, a program using a differential approach to constraints never needs to guess which solution the user wants if the system is underconstrained. In such cases, after creating the constrained system the user can manipulate the scene until the desired geometry is achieved. This is preferable to the user trying to create a more complicated constraint set which more completely specifies the desired solution or changing the initial condition of an equation solver to achieve the desired appearance.

The comprehension problem is different and in many ways easier in a differential constraint system than in an algebraic-solving approach. Since all motion is continuous, there is no puzzling over how a model got from one state to another. Since the system never has to select a solution, there are no decisions to explain to the user.

By superimposing a visual representation for the constraints on the model, the user can look at the motion and see why it is happening. The connection between the relationships that will be maintained and the objects they affect is shown to the user. Continuous motion also helps the user understand the model. The use of a visual representation of constraints drawn directly on the model combined with the continuous motion of the model allows a user's perceptual skills to aid in the comprehension process.

Because it easy for a user to grab a model and pull it with continuous motion, users can experiment with models. This ability not only permits users to see how models are constructed but also provides a method for understanding the model. For example, it can aid in the understanding of the kinematic behavior of a mechanical device (figure 4). Dynamic Drawings, models created primarily for experimentation by manipulating them, are an interesting application made feasible by our constraint methods.

3.4. Displaying and Editing Constraints

A difficult issue in constraint systems is how to make the constraints visible to the user. Some alternatives that have been used are textual languages such as [Nel85], schematic representations such as [Bor86], or visual representations such as [Sut63], which superimpose constraints on the images of the objects they connect.

Augmented Snap-Dragging provides a basis for visually representing constraints as well as specifying them. The two simple snapping constraints combined with other drawing tools such as alignment objects provide a visual language for a wide variety of constraints (figure 1). By using this common language for creation and display, we avoid having multiple constraint representations for users to learn.

Visually representing constraints on the image of the model can be difficult since constraints tend to cluster, often being in exactly the same place. The semantics of constraints can be tuned to make this problem easier. For example, the equivalence classes mentioned earlier simplify visual representation. If a set of points has been made equivalent, just the equivalence class needs to be shown, not the potentially large number of equality relationships all piled on top of each other.

These grouping semantics also help simplify constraint editing. Deletion becomes removal of a point from a group, so the user need not worry about the order in which constraints were created. Also, removing a point from a group provides a consistent mechanism for handling other constraints on the points: the other constraints stay with the group. This frees the user from having to remember how the constraints were created.

Facility in editing constraints is important; relations should be as easy to break as to make. Users may change their minds as to what relationships should be in the model, or may simply have made a mistake in specifying the constraints.

We have developed schemes for editing constraints which allow the user to remove constraints not by specifying the constraints themselves, but instead by referring to the effects that the constraints have. In both schemes, the user pulls on objects. This, as well as the ability to refer to and delete alignment objects as regular objects, avoids the need for a separate mode in which the user refers to constraints, which can be difficult to do well since constraints tend to cluster together.

A direct method of removing constraints is to "rip" them. When an object is grabbed for dragging, holding down a modifier key causes the grabbing operation to "grab hard," removing any constraints on the point grabbed.

Another technique for deleting constraints is to allow the user to temporarily disable constraints. In this mode, the user can manipulate the objects as if they did not have any constraints on them. When constraints are re-enabled, constraints which the user has broken by pulling them apart are discarded. Feedback as to which constraints are broken and the ability to use snapping operations to reassemble things help make this a useful technique for editing the constraints.

The "undo" command is a simple example of another method we employ to remove constraints. After a new constraint is automatically created, the user is able to tell the system that it is incorrect. When a constraint is initially created, it is displayed prominently so the user can check it, and discard it if it is unwanted.

3.5. Performance and Scalability

Scalability is an important and often difficult issue in constraint-based systems. Since we rely on the continuous motion of models, it is important that our system achieve iteration rates high enough to give the appearance of smooth motion. Here, we argue that constrained dynamics will scale well so the needed performance can be achieved even on large models. We also provide techniques to further enhance performance.

The complexity of the constrained dynamics computation is dominated by solving the linear system. However, the systems of constraints are typically sparse so they can be solved much more quickly than the $O(n^3)$ time required to solve dense linear systems. A variety of fast algorithms exist to solve sparse problems efficiently[DER86, PFTV86]. Non-linear algebraic equation solvers also typically solve systems of linear equations, but require iterations until the solver converges (if ever).

Solving isn't the only potential expense of constraints. Traditional drawing programs are possible on modest computers since the main actions requiring rapid feedback are $O(1)$ because only a small number of objects are moving at a given time regardless of the size or complexity of the drawing. With constraints this is no longer true. All objects in the scene can potentially be moving at the same time if they are connected. The basic operations which must happen every iteration, such as redrawing the moving objects, become $O(n)$.

However, typically not all objects need to move at once. This worst case only occurs in highly connected models. If we know an object is not moving, we neither need compute the forces on it nor continuously update it. Therefore, we can turn it off so it does not take part in the calculations or need to be updated every iteration. Turning things off is an important performance optimization, but since it is only an optimization, we should not require the user to specify moving objects explicitly, as was done in [PEF$^+$90].

When the user grabs a point in the scene to manipulate it, our system checks to see which objects and constraints must be turned on. Any object which is connected to the mouse through the constraint graph could potentially move. Objects, or parts of them, are sometimes forced to be stationary by some complicated combinations of constraints making it difficult to prove that an object cannot move. Since it is only an optimization, it is unimportant to find all stationary objects, but finding as many dead objects as possible is useful since it can substantially speed solving and refresh.

A user may want to freeze an object in place. This is an example of a constraint which can be accomplished by simplification. In this case instead of adding constraint equations we just turn the object off. Such constraints make things go faster because they do not add additional equations and they remove variables from the computations. Constraints maintained by simplification also have the advantage that they cannot be violated by numerical error.

Hierarchies also provide constraints which simplify the model. Being able to hierarchically group objects together and treat them as one is a popular and useful technique in drawing programs. In our system, users can group objects and then rotate, translate and scale these compound entities. The objects in a group are effectively asleep: their parameters cannot change, only their parent transformation can.

In order to have hierarchies in a constraint system, the program must be able to rewrite the equations on the fly, since when grouped, an object's equations change. Often these new rewritten equations become simpler. Many constraint relationships, such as "point–on–object" and "points–coincident", are inherently preserved by grouping, and are therefore not needed inside groups. The constraint rewriting mechanism allows simplification of constraints other than those in hierarchies as well.

3.6. Lightweight Constraints

The facility with which we handle constraints opens the possibility of using constraints to aid in manipulation. Most constraints are used to represent structure in the drawing, describing relationships which are really in the scene. However, constraints can also be useful in the manipulation process if they are easy enough to create and destroy. We call constraints which are meant to be short-lived "lightweight" constraints.

A very useful lightweight constraint is the tack. Tacks are lightweight constraints which hold a particular point in place. They act as an extra hand, making it easy to stretch or rotate an object. Tacks can perform the tasks that anchors do in traditional Snap-Dragging[Bie89].

The constraint mechanism can also be used for dragging objects, as was done in [BDFB$^+$87]. The mouse is connected to the point grabbed. This constraint is unique in that it has a lower priority than other constraints: satisfying it should not mean breaking other constraints. Grabbing is also a unique constraint in that it is chasing a moving target. Because of the limited speed of computers, the grabbed object is often unable to keep up with the mouse, especially when we limit the speed that objects can move in order to keep things mathematically stable. We use control techniques, as were used in [WW90], to keep the grabbed object moving towards the cursor at a (nearly) constant rate.

4. Conclusions

The concepts in this paper have been built into a two dimensional drawing program (figure 5). The program contains Snap-Dragging operations and automatically creates constraints, but allows these features to be disabled so that other styles of use are possible, including using the program as an algebraic-solving system.

The application uses several techniques to prune dead objects and to filter redundant snapping operations. Drawings can be included in documents created by the text formatter we use. The program also lets us play with dynamic drawings, manipulating constrained objects to experiment with them (figure 4). Experience with the program helps us asses how well our ideas work in practice. We are enhancing it to attract a larger number of users, which will provide even more feedback.

We are not limited to the set of tools which are provided by the original Snap-Dragging. New tools in the paradigm can be developed to express more types of relationships and constraints, sculpt different kinds of objects such as free-form curves, and provide help for manipulating text and symbols. Tuning the tools for specific domains, such as mechanical devices, is promising since semantic information can be used to further reduce the possible (or likely relationships).

Both Snap-Dragging and Constrained Dynamics work in 3D [Bie90, WGW90]. The combination should also work in 3D. Making the feedback of snapping operations and constraints understandable might be a challenging task, but techniques such as hardware transparency, real time realistic rendering and stereo displays might help provide solutions.

Since we now have a way to easily create and manipulate constrained models, we can again think about new applications for constraint technology, such as preliminary design, interactive animation and dynamic drawings.

In this paper, we have presented methods for aiding users in the construction and manipulation of constrained models. We showed how Snap-Dragging, a successful technique for drawing, can be augmented to create constrained models with no extra effort on the part of the user. Equipped with the constraints and a solution to them, we provided techniques for manipulating the constrained models based on physical simulation techniques. The approach works well for drawing in two dimensions, and shows promise for use in other applications.

References

[BB88] Ronen Barzel and Alan H. Barr. A modeling system based on dynamic constaints. *Computer Graphics*, 22:179–188, 1988.

[BD86] Alan Borning and Robert Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, 5(4):345–374, October 1986.

[BDFB+87] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. In *Proceedings OOPSLA*, pages 48–60, October 1987.

[Bie89] Eric Bier. Snap-dragging: Interactive geometric design in two and three dimensions. Technical Report EDL-89-2, Xerox Palo Alto Research Center, 1989.

[Bie90] Eric Bier. Snap-dragging in three dimensions. *Computer Graphics*, 24(2):193–204, March 1990. Proceedings 1990 Symposium on Interactive 3D Graphics.

[Bor79] Alan Borning. *Thinglab – A Constraint Oriented Simulation Laboratory*. PhD thesis, Stanford University, July 1979.

[Bor86] Alan Borning. Defining constraints graphically. In *Proceedings CHI 86*, pages 137–143, April 1986.

[BS86] Eric Bier and Maureen Stone. Snap-dragging. *Computer Graphics*, 20(4):233–240, 1986.

[Cor89] Claris Corporation. Claris CAD 2.0. Computer Program, 1989.

[DER86] J. S. Duff, A. M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Oxford, UK, 1986.

[FP88] N. Fuller and D. Prusinkiewicz. Geometric modelling with euclidean constructions. In M. Magnenant-Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics: Proceedings of CG International '88*. Springer-Verlag, 1988.

[Gol80] Herbert Goldstein. *Classical Mechanics*. Addison Wesley, 1980.

[GW91a] Michael Gleicher and Andrew Witkin. Differential manipulation. *Graphics Interface*, June 1991.

[GW91b] Michael Gleicher and Andrew Witkin. Snap together mathematics. In Edwin Blake and Peter Weisskirchen, editors, *Advances in Object Oriented Graphics 1: Proceedings of the 1990 Eurographics Workshop on Object Oriented Graphics*. Springer Verlag, 1991. Also appears as CMU School of Computer Science Technical Report CMU-CS-90-164.

[Hil91] Ralph D. Hill. A 2-d graphics system for multi-user interactive graphics based on objects and constraints. In E. Blake and P. Weisskirchen, editors, *Advances in Object Oriented Graphics 1: Proceedings of the 1990 Eurographics Workshop on Object Oriented Graphics*, pages 67–92. Springer Verlag, 1991.

[IC87] Paul Issacs and Michael Cohen. Controlling dynamics simulation with kinematic constraints, behavior functions and inverse dynamics. *Computer Graphics*, 21(4):215–224, 1987.

[Lel88] Wm. Leler. *Constraint Programming Languages: Their Specification and Generation*. Addison-Weseley, 1988.

[LGL81] V. C. Lin, D. C. Gossard, and R. A. Light. Variational geometry in C.A.D. *Computer Graphics*, 15(3):171–177, 1981.

[Li88] Jiarong Li. Using constraints in interactive text and graphics editing. In P. A. Duce and P. Jancene, editors, *Eurographics*, 1988.

[MB86] Brad A. Myers and William Buxton. Creating highly-interactive and graphical user interfaces by demonstration. *Computer Graphics*, 20(4):249–258, 1986.

[MGD+90] Brad A. Myers, Dario Guise, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Ed Pervin, Andrew Mickish, and Phillipe Marchal. Comprehensive support for graphical, highly-interactive user interfaces: The garnet user interface development environment. *IEEE Computer*, November 1990.

[MKW89] D. L. Maulsby, K. A. Kittlinz, and I. H. Witten. Metamouse: Specifying graphical procedures by example. *Computer Graphics*, 1989.

[Nel85] Greg Nelson. Juno, a constraint based graphics system. *Computer Graphics*, 19(3):235–243, 1985.

[NKK+88] T. Noma, T. L. Kunii, N. Kin, H. Enomoto, E. Aso, and T. Yamamoto. Drawing input through geometrical constructions: Specification and applications. In M. Magnenant-Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics: Proceedings of CG International '88*. Springer-Verlag, 1988.

[Nor90] Donald Norman. *The Design of Everyday Things*. Doubleday, 1990.

[PEF+90] A. Pentland, I Essa, M. Friedmann, B. Horowitz, S. Sclaroff, and T. Starner. The thingworld modeling system. In E. F. Deprette, editor, *Algorithms and Parallel VLSI Architectures*. Elsevier Press, October 1990.

[PEWW90] P. Piela, T. Epperly, K. Westerberg, and A. Westerberg. Ascend: An object-oriented computer environment for modeling and analysis. part 1 - the modeling language. Technical Report EDRC 06–88–90, Engineering Design Research Center, Carnegie Mellon University, 1990.

[PFTV86] William Press, Brian Flannery, Saul Teukolsky, and William Vetterling. *Numerical Recipes in C*. Cambridge University Press, Cambridge, England, 1986.

[Pla89] John Platt. *Constraint Methods for Neural Networks and Computer Graphics*. PhD thesis, California Institute of Technology, 1989.

[PW85] Theo Pavlidis and Christopher Van Wyk. An automatic beautifier for drawings and illustrations. *Computer Graphics*, 19(3):225–234, 1985.

[Sap89] Mark Sapossnek. Research on constraint-based design systems. In *Proceedings of AI in Engineering*, July 1989.

[Ser87] David Serrano. *Constraint Management in Conceptual Design*. PhD thesis, M.I.T., 1987.

[Sut63] Ivan Sutherland. *Sketchpad: A Man Machine Graphical Communication System*. PhD thesis, Massachusetts Institute of Technology, January 1963.

[SZ90] Peter Schroeder and David Zeltzer. The virtual erector set: Dynamic simulation with linear recursive constraint propagation. *Computer Graphics*, 24(2):23–31, March 1990. Proceedings 1990 Symposium on Interactive 3D Graphics.

[VW82] Christopher J. Van Wyk. A high level language for specifying pictures. *ACM Transactions on Graphics*, 1(2):163–182, April 1982.

[WGW90] Andrew Witkin, Michael Gleicher, and William Welch. Interactive dynamics. *Computer Graphics*, 24(2):11–21, March 1990. Proceedings 1990 Symposium on Interactive 3D raphics.

[Whi88] R. M. White. Applying direct manipulation to geometric construction systems. In M. Magnenant-Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics: Proceedings of CG International '88*. Springer-Verlag, 1988.

[WK88] Andrew Witkin and Michael Kass. Spacetime constraints. *Computer Graphics*, 22:159–168, 1988.

[WW90] Andrew Witkin and William Welch. Fast animation and control of non-rigid structures. *Computer Graphics*, 24(4):243–252, August 1990. Proceedings SigGraph '90.

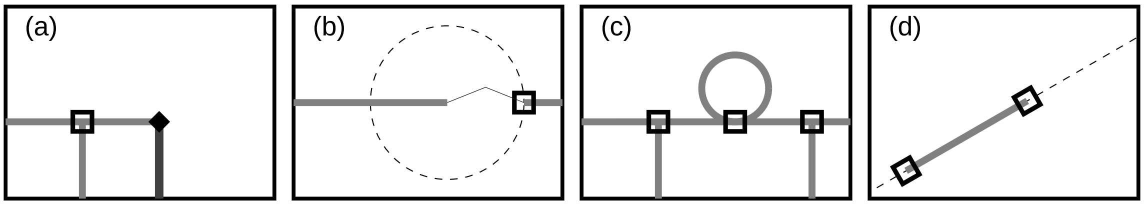

Figure 1: Snap-Dragging provides a basis for visual representation of constraints. The two basic constraints, point-on-object and point-to-point, are drawn as an empty square and a filled diamond (a) respectively. Other constraints are represented using the basic ones and alignment objects, for example, distance-between points (b), points-aligned (c), and orientation (d). Bent green lines emphasize distance constraints.

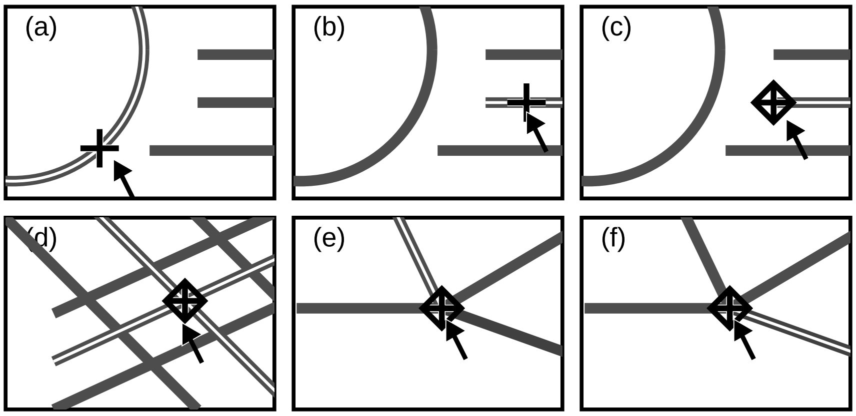

Figure 2: Feedback mechanisms display exactly what is snapped to. The cursor changes shape depending on whether it is snapped to a curve or edge (a,b), or a point such as an endpoint or intersection (c,d). The object snapped to is brightly lit. If several objects are close together, the one desired can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

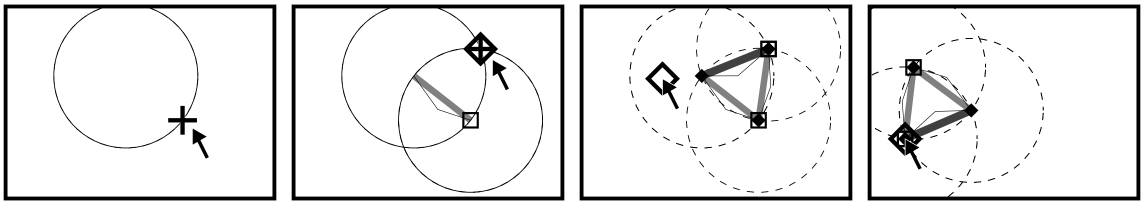

Figure 3: Alignment circles help create a 3/4 inch equilateral triangle. Snapping to a 3/4 inch circle sets the length of the first line. Snapping to the intersection of the circles constrains the point to be 3/4 inches from each endpoint. Snapping the final segment in place leaves three lines constrained to be connected with their endpoints 3/4 inches apart. The system can maintain these constraints as as the user drags pieces of the triangle.

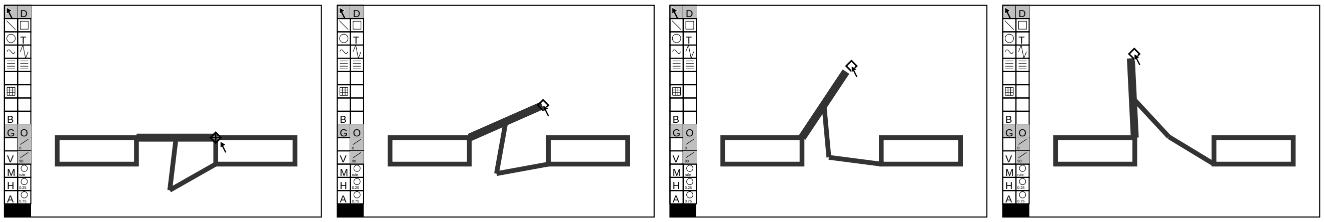

Figure 4: A constrained model of a door-closer mechanism. As the model is dragged, the constraints are enforced, allowing the user to experiment with the kinematic behavior of the object.

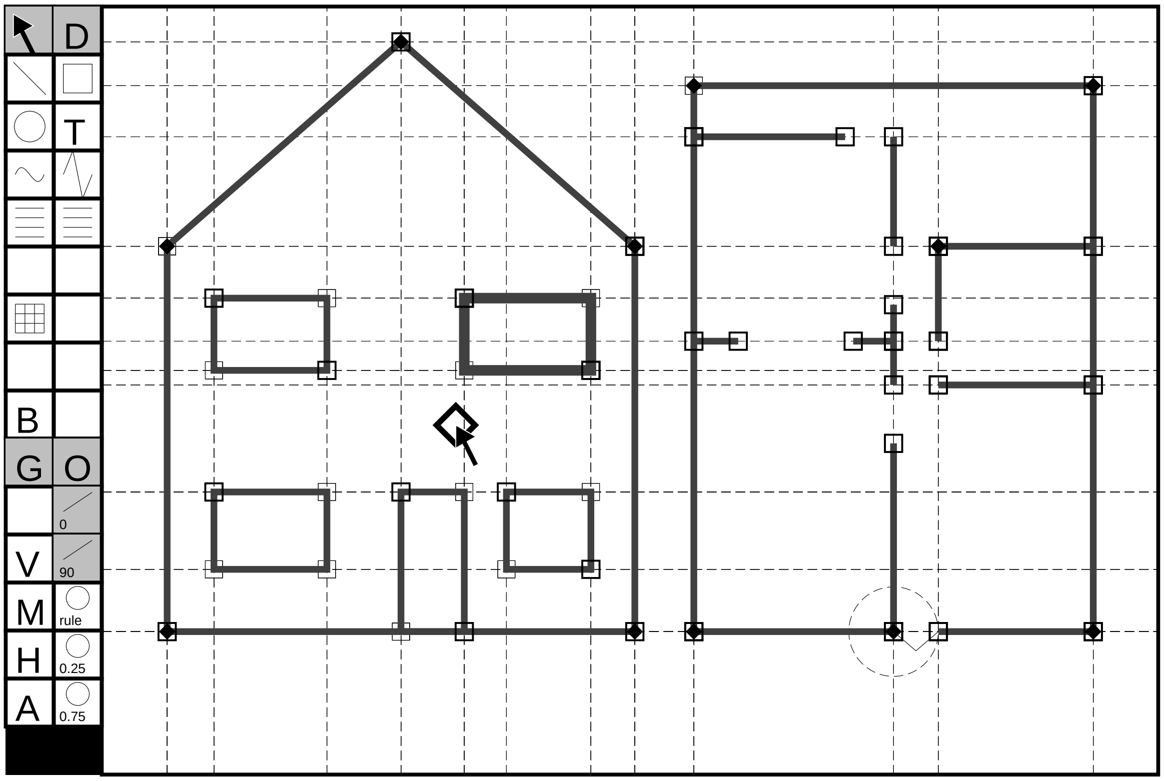

Figure 5: Our two dimensional drawing program editing a constrained model.