Snap-Dragging

Eric Allan Bier
Dept. of EECS
UC Berkeley
Berkeley, CA 94720

Maureen C. Stone
Xerox PARC
3333 Coyote Hill Rd.
Palo Alto, CA 94304

Abstract

We are interested in the problem of making precise line drawings using interactive computer graphics. In precise line drawings, specific relationships are expected to hold between points and lines. In published interactive drawing systems, precise relationships have been achieved by using rectangular grids or by solving simultaneous equations (constraints). The availability of fast display hardware and plentiful computational power suggest that we should take another look at the ruler and compass techniques traditionally used by draftsmen. Snap-dragging uses the ruler and compass metaphor to help the user place his next point with precision, and uses heuristics to automatically place guiding lines and circles that are likely to help the user construct each shape. Snap-dragging also provides translation, rotation, and scaling operations that take advantage of the precision placement capability. We show that snap-dragging compares favorably in power and ease of use with grid or constraint techniques.

CR Category: I.3.6 [Methodology and Techniques, Interactive techniques]

Additional Keywords: Interactive design, geometric construction, constraint systems

1. Introduction

Artists drawing technical illustrations often require that precise relationships are held among picture elements. For example, certain line segments should be horizontal, parallel, or congruent. In the past, interactive illustration systems have provided techniques such as grids and constraints to facilitate precise positioning. Both of these techniques have significant limitations. Grids provide only a small fraction of the desired types of precision. Constraints, while very powerful, require the user to specify additional structures that are often difficult to understand and time-consuming to manipulate. Our interactive technique, *snap-dragging*, is a compromise between the convenience of grids and the power of constraints.

Snap-dragging is based on the familiar idea of snapping the display cursor to points and curves using a gravity function. The new idea is that a set of *gravity-active* points, lines, and circles called *alignment objects* are automatically constructed from hints given by the user plus heuristics about typical editing behavior. The result is a system with as much power as a ruler, compass, protractor, and T-square drafting set, but with little time spent in moving the tools and setting up the constructions.

Snap-dragging can be seen as an extension of the idea of gravity-active grids, used in illustrators such as Griffin [Stone80], Gremlin [Opperman84], MacDraw® [MacDraw84], and Star® Graphics [Lipkie82], or as a simple form of constraint solver, used in systems such as Sketchpad [Sutherland84], ThingLab [Borning79], and Juno [Nelson85]. As in a grid system, the cursor may snap to gravity-active objects provided by the system. However, the set of gravity-active objects is richer, and varies with the current scene and the operation being performed. As in a constraint-based system, points can be placed to satisfy angle, slope, and distance constraints. However, with snap-dragging, constraints are solved two at a time, with the user explicitly controlling the placement of each point, and the constraints are forgotten as soon as they are used, rather than becoming part of the data structure.

Snap-dragging is currently implemented as part of the Gargoyle two-dimensional illustrator, running in the Cedar programming environment [Swinehart85], on the Xerox Dorado high-performance personal workstation [Pier83]. All figures in this paper were created with Gargoyle.

In Section 2 we will present snap-dragging in more detail. Section 3 is an analysis of design issues for geometric precision techniques: comparing grids, constraints, and snap-dragging. Section 4 presents more detail on the implementation of snap-dragging. Section 5 summarizes our results.

2. Snap-Dragging

This section describes how snap-dragging is used to construct shapes and perform transformations. In Gargoyle, a special point called the *caret* can be placed, with precision, using gravity. The caret is distinct from the *cursor*: the cursor always moves with the mouse, while the caret can stray from the cursor if a gravity-active object attracts it. The caret is the source of precision. When control points are added to a scene, they are always placed at the position of the caret. Positioning operations use the position of the caret as a parameter.

When one control point of an object is moved, affected line segments are adjusted, in real time, using a technique called rubber-banding [Newman79]. Transformed objects are smoothly dragged into position. The user specifies which region of the scene and which

kinds of alignment objects are of interest. The system automatically constructs alignment objects of that kind in the requested region. Alignment objects are visible only when the user is in the process of moving a point or object, and are drawn in gray so they may be easily distinguished.

Figure 1 shows how a user would adjust a point in an existing triangle to make the base horizontal. In figure 1(a), the user has activated horizontal alignment lines, and picked up a vertex with the caret. In figure 1(b), the user drags the vertex until it snaps onto one of the alignment lines. When the operation is finished, the alignment lines will disappear.

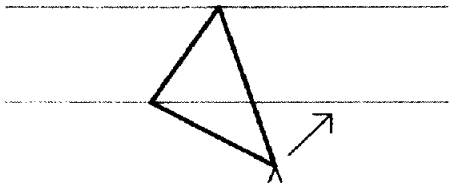

Figure 1(a) Picking up a vertex with the caret.

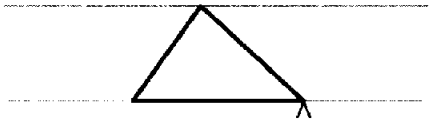

Figure 1(b) Snapping the caret onto an alignment line.

Figure 2 demonstrates construction of an equilateral triangle, 3/4 inch on a side, with its base at 15 degrees to the horizontal. Constructing the triangle takes six steps:

1) Activate lines of slope 15 degrees (click on a menu).
2) Activate 3/4 inch circles (click on a menu).
3) Place the lower left vertex. A 3/4 inch alignment circle appears centered on the new vertex, and an alignment line sloping at 15 degrees appears passing through the new vertex.
4) Place the second vertex at one of the intersections of the circle and the 15 degree line. Figure 2(a). A second alignment circle appears, centered on the second vertex.
5) Place the last vertex at the intersection of the two circles. Figure 2(b).
6) Invoke a close polygon command, or place a fourth vertex on top of the first. Since the operation is finished, the alignment objects disappear to allow the artist to inspect the shape. Figure 2(c).

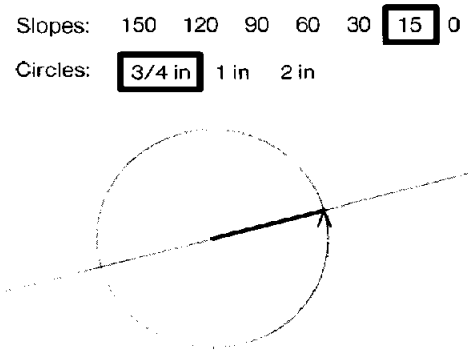

Figure 2(a) The user places the first two points of an equilateral triangle with side of length 3/4 inch and with its base at 15 degrees. A 3/4 inch alignment circle and a 15 degree alignment line appear.

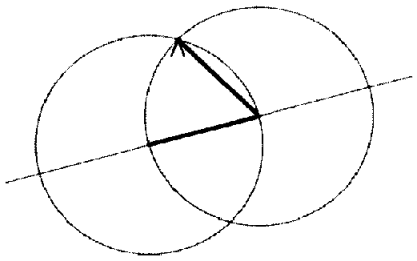

Figure 2(b) Another circle appears. The last vertex is placed at the intersection of the two circles.

Figure 2(c) The triangle is closed and all alignment lines disappear.

The whole process takes six keystrokes and all but two of these would be needed to draw any triangle. Note that in a grid system, it is impossible to construct an equilateral triangle, and in most constraint systems, only a triangle with a horizontal base would be easy to construct.

Once an object has been created, it can be translated, rotated, scaled, and combined with other objects. The ability to position the cursor on alignment objects makes it easy to perform these operations with precision. Each transformation is performed in three steps. First, the objects to be transformed are selected. Second, the caret is placed at an initial position in the scene, usually on a selected scene object or alignment object. Third, the selected objects are smoothly transformed using the displacement between the original position of the caret and the new (constantly updated) position of the caret to determine the current transformation. Like the original caret position, the new caret position can be snapped to alignment objects. We will discuss the operations translate, rotate, and scale in more detail next.

2.1 Translation

When performing translation, we simply add a displacement vector (old caret - new caret) to the original position of each selected object to get its new position. We move one polygon P to touch another polygon Q by selecting P, placing the caret at a point on P, and snapping the caret onto a point on Q. Figure 3 shows how point-parameterized translation can be used to align two rectangles: the caret is snapped onto a vertex, A, and then onto a 90 degree line in this example.

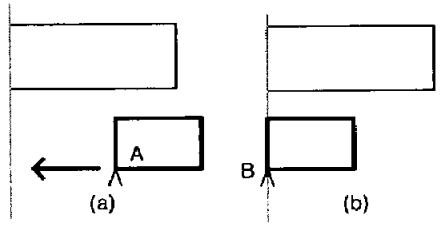

Figure 3 An example of translation parameterized by two points (A and B). (a) The caret snaps onto point A on the object. (b) The user then drags and snaps the caret onto a vertical alignment line at point B. A single vertical alignment line appears because the user has expressed an interest in the left edge of the fixed rectangle.

2.2 Rotation

The center of rotation, called the *anchor*, is placed by positioning the caret and invoking the Drop Anchor command. During rotation, the angle of rotation is kept equal to the angle through which the displacement vector (old caret - new caret) has moved from its initial position. If we initially position the caret on a point A of the selected object, then the anchor, the point A, and the caret remain collinear throughout the rotation. Consider the example in figure 4. We place the anchor point at the base of an arrow, place the caret initially onto its tip (point A), and drag the caret to the object we would like the arrow to point to (point B).

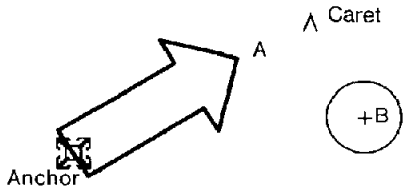

Figure 4(a) The caret, which was initially placed on point A, has been dragged to an arbitrary position. Anchor, A, and caret are kept collinear.

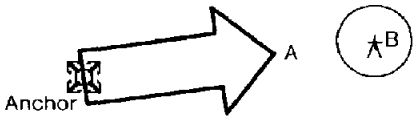

Figure 4(b) When the caret is placed on point B, the arrow points at it; Anchor, A, and B are collinear.

2.3 Scaling

The anchor point is also used as the center of scaling. The scale factor is the ratio of the magnitudes of the new caret displacement vector (caret - anchor) to the original caret displacement vector. This transformation can be used to scale one object until one of its dimensions matches a dimension of another object. In figure 5 the square is enlarged to fit onto an edge of the hexagon.

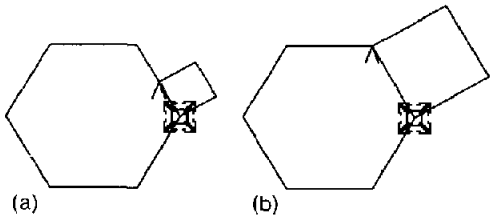

Figure 5(a) The anchor is fixed on one corner of the square and the caret is positioned on the adjacent corner. (b) The square is scaled as the caret is dragged to the vertex of the hexagon.

3. Design Issues for Geometric Precision Tools

The ideal tool is powerful but easy to use. Good tool design, therefore, requires a compromise between functionality and user interface complexity. We measure the functionality of a geometric precision tool by examining which geometric relationships can be expressed directly. These relationships must be weighed against the total number of commands the user must learn and the time taken to specify all the arguments and invoke each command.

Ease of use involves more than the reduction of commands and keystrokes. A system that is easy to use will be predictable, require user input in proportion to the difficulty of the task, and be free from catastrophic failures that destroy hours of work. We will show that snap-dragging makes it easy to reach these goals.

In this section, we will describe the sets of relationships we use to define the power of a geometric tool, and describe some important factors in making a system easy to use. In the process, we will compare snap-dragging to grids and constraints.

3.1 Geometric Power

Our approach to the problem of making precise line drawings has been to assume that the designer will define shapes by carefully placing points rather than by sketching and aligning afterwards, which is the method used by Pavlidis and VanWyk [Pavlidis85]. Our design philosophy focuses the design issues on methods for positioning points, and especially on the issue of positioning a new point with respect to the other points in a scene or with respect to the boundaries of the drawing space. Whether or not a picture is actually produced by placing one point at a time, we can measure the power of a geometric design system, particularly a polygon-based one, on the point-to-point relationships (e.g., distance, slope, angle) that its user can express. We describe a taxonomy of point-to-point relationships and a taxonomy of geometric constructions and use them to compare grid, constraint, and snap-dragging systems.

One way to organize the point-to-point relationships that a user might want is by the set of affine transformations that preserve a given relationship. We use six groups of transformations in the classification scheme shown in figure 6. Each group is a subset of the group above it. They are: general affine transformations (called the affine group in algebra texts); combinations of scaling, rotation, and translation (the extended similarity group); combinations of scaling and translation; combinations of rotation and translation (the Euclidean group); pure translation (the group of translations); and the identity transformation. The point-to-point relationships that can be preserved under these transformations are parallelism (which includes collinearity), angles, ratios of distances, slopes, distances, vectors, and coordinates. Each relationship in figure 6 is invariant to the group named in its box, and to all those groups below it. We call these the *seven basic relationships* because, when a designer scales, rotates, and translates shapes to assemble an illustration, he can count on these relationships to be maintained.

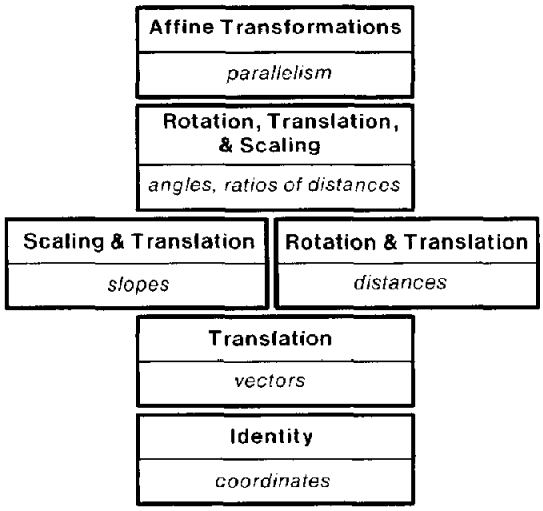

Figure 6 A taxonomy of our seven basic geometric relationships. Relationships appear in the bottom half of each box. Each relationship is invariant to the transformations listed in the top half of its box and to all transformations below its box.

One way to organize geometric constructions is by the number of points in the construction. We view a construction as a function that takes a set of points as arguments and returns a point or curve in the plane. The number of point arguments that a construction takes has a direct bearing on the user interface: constructions involving more points will take more work to specify or will require more guesswork on the part of the tool. However, constructions that take many point arguments can potentially be more powerful than those that take fewer. The seven basic relationships can be produced using constructions that take only zero, one, or two points as arguments. One way to do this is:

> **Group 0**: No points are needed as arguments. Construct a line of known *x coordinate*, or known *y coordinate*.
>
> **Group 1**: One point, A, is needed. Construct a circle of points that are a known *distance* from A, construct a line through A of known *slope*, or construct a point at a known *vector* (distance *and* slope) from A.
>
> **Group 2**: Two points, A and B, are needed. Construct lines that are a known *angle* from segment AB, construct lines that are *parallel* to AB at a known distance, or construct the midpoint of AB. Midpoints are a special case of the *ratio of distances* relationship.

Each relationship can be expressed using a different construction group. For instance, parallelism can be created with a Group 3 construction:

> **Group 3**: Three points, A, B, and C, are arguments. Construct a line, parallel to A and B, that passes through C.

Our taxonomies of relationships and constructions highlight the limitations of grid systems. A grid is the computer graphics equivalent of graph paper. When the grid is turned on, all points placed by the user will be forced to land on the intersection points of the rules of the graph paper (figure 7). This gives us only a discrete set of absolute coordinates. Only a subset of the possible coordinates, distances, and slopes can be built, and only those that are aligned with the grid axes are easy to specify. Operations in Group 2 and higher order groups are particularly difficult to achieve since the chances are small that a larger set of points will satisfy a precise relationship *and* lie on the grid.

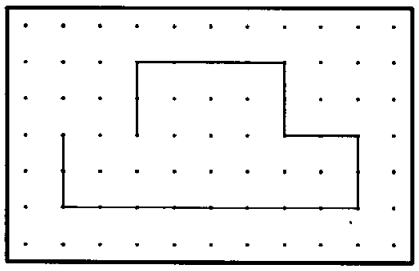

Figure 7 An object with horizontal and vertical lines is constructed with a grid system.

Constraint-based systems can potentially provide all of the relationships mentioned above and more. For instance, the Juno constraint-based illustrator [Nelson85] provides some Group 1 constructions (for horizontal and vertical), and some Group 3 constructions (for parallel and congruent). The user selects as many as four points at a time and then specifies the new relationship between them that the system should enforce. An example is shown in figure 8. Not only are a larger set of relationships possible than with the other two approaches, but many constraints can be solved simultaneously. Constraint-based illustrators are typically very powerful by our metric.

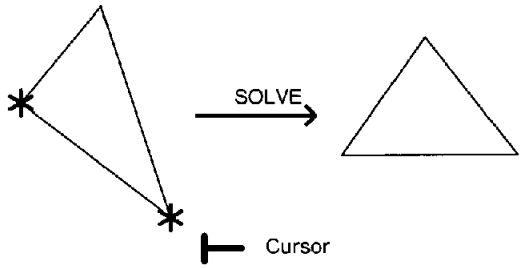

Figure 8 A Juno user touches two points with a T-square shaped cursor to add a horizontal constraint. After the constraint solver is invoked, a new triangle is drawn.

Snap-dragging, as implemented in Gargoyle, provides only Group 1 and 2 constructions. We felt this was an appropriate balance between power and ease of use. Providing more complex relationships would stretch the capabilities of snap-dragging: the technique relies on being able to guess what relationships will be useful, and there would be too many groups of three or more points to make guessing possible. Snap-dragging is generally less powerful than constraints and more powerful than grids.

3.2 Ease of use

To be easy to use, a system should be predictable, require user input in proportion to the difficulty of the task, and be free from catastrophic failures that destroy hours of work. In this section we focus on how the properties of geometric precision schemes affect these aspects of system design.

3.2.1 Predictability and User Confidence

A system is predictable if the designer can confidently predict the results that each action will have on the state of his illustration. A predictable system will provide commands at the right semantic level to help the user concisely express his intent, and will provide good feedback when the intent has been captured. Too low a semantic level requires the user to perform tasks that take too many steps, which is tedious and unreliable. Too high a level results in operations that are hard to learn and understand.

Superficially, grid systems are predictable. The grid itself is a visible reminder of the active constraints. The mechanism of a grid, however, provides no straightforward way to express relationships as simple as specific distances or angles. For example, a user trying to construct a 45 degree line has to express this idea in primitive terms: move 5 points over and 5 points up. To be sure the angle is correct, he may need to count grid points several times. So while a grid is easy to use, it may not inspire confidence that a particular relationship is satisfied because it provides control at too low a level.

Constraint-based systems tend to be difficult to control. A set of simultaneous non-linear equations almost always has multiple solutions. Many points may be moved at once to reach a solution, drastically altering the scene. Presenting and controlling all the constraints and possible solutions is a formidable user interface problem. Some constraint-based systems do not attempt to describe the constraints graphically, relying on a textual description and letting the user correlate the text with the geometry. In any case, getting the constraints correct can be more like debugging code than like graphical editing.

With snap-dragging, one point is moved at a time, and all constraints are represented as visible lines and circles so that the constraints remain simple and visible. We believe snap-dragging presents commands at the correct semantic level, providing the same operations that a designer would use to describe the construction.

3.2.2 Appropriate User Input

A system makes appropriate use of user input if a small change to the picture always requires a short period of designer time. The amount of work required to produce a design must be measured not only in keystrokes but in the amount of effort it takes to make use of the design tools. Most systems do well at providing local optimization of keystrokes by taking advantage of *coherence* in the editing session, providing ways to free the user from repeatedly indicating that the same relationship and/or values are needed in a sequence of editing steps. However, in systems that are optimized for specific directions, such as horizontal and vertical, or that limit precise editing to a discrete grid, a significant amount of planning and extra construction may be needed for precise editing to proceed at all stages of the design.

Grid systems are both quantized and optimized for constructions that align with the edges of the page. To maintain precise control throughout the editing session, the designer must plan ahead to make sure that all his vertices can be placed on grid points. For example, if an arrowhead is to be attached precisely to the middle of the edge of a rectangle, the rectangle must be an even number of grid units long, and the designer must take this into account when the rectangle is created. Otherwise, it may be necessary to change the size of the rectangle when the arrowhead is added, which may require further changes to the picture.

Constraint-based systems often include direction-specific constraints such as horizontal and vertical. A shape defined using these constraints cannot be rotated without either changing or violating the constraints. Editing the constraints to make them consistent with the new rotation requires the user to change the constraint network everywhere the horizontal and vertical constraints are used. It is possible to design a shape with only a single mention of absolute orientation so that it can be rotated by altering that one constraint. However, designing such a re-orientable shape requires extra planning and effort.

The most important feature of an alignment paradigm based on points and geometric constructions is that it works equally well at arbitrary scales and rotations. The transformation independence in Gargoyle comes from three sources. First, Gargoyle treats all angles and scaling factors in an equivalent fashion: it is just as easy to align the base of the triangle in figure 2 with a 15 degree line as it would be to make the base horizontal. Second, Gargoyle implements relationships that are preserved under scaling and rotation, as described in section 3.1 above: if the designer uses one inch circles to build a shape with one inch features, he can rotate the object and then continue to edit it without difficulty. Third, Gargoyle allows the user to set the scaling unit to an arbitrary value. If the designer uses 1 inch circles to create a shape, and later scales the shape, he cannot use his one inch circles to modify the shape further. However, if the designer can temporarily alter the scaling unit, multiplying it by the same factor used to scale the object, the "one inch" circles will again line up with the shape, allowing it to be edited in place. This is like fitting the grid to an object in a grid-based system, instead of moving the object back onto the grid.

We believe that the transformation independence of snap-dragging is the most significant advantage it has over grids and constraints in reducing the work needed to make an illustration.

3.2.3 Preventing Catastrophic Failures

Catastrophic failure occurs when a short sequence of actions can destroy large amounts of work, requiring the user to redo significant parts of the design. Such failures include scaling by zero, constraint solutions that collapse several lines or points together, and accidental deletions. In addition, it is a catastrophic failure if, late in the illustration process, a problem is uncovered that requires significant rework of pieces of the illustration. These problems include shapes moved prematurely off a grid, designs that are incorrectly quantized, subsections of a design that don't fit together, constrained objects that can't be rotated, and other failures of planning. We feel snap-dragging is a good methodology for preventing both types of catastrophic failures.

Most work losses can be prevented by good feedback and by the ability to undo an action. Snap-dragging provides visible feedback for transformations as they occur. Likewise, undo operations and incremental checkpoints of the system state are easy to implement since each operation makes a small change and leaves the scene objects in a well-defined state.

Failures of planning can never be completely eliminated but a tool that is transformation-independent is clearly more adaptable than one that provides its full power only in specific directions, such as horizontal and vertical, and specific scale units, such as multiples of the grid spacing. Errors can be corrected in place, independent of orientation. Shapes can be combined and adjusted independent of the original orientation or size. Overall, we feel that snap-dragging offers a significantly better compromise between power and ease of use than grids or constraints.

4. Snap-Dragging Implementation Details

In this section we will provide more detail about three important aspects of snap-dragging: how we decide which alignment objects are active, how allowing the user to measure existing geometric relationships can increase the utility of alignment objects, and how we get snap-dragging to work in real time.

4.1 Choosing Alignment Objects

Ruler and compass constructions have traditionally been performed by adding one circle or one line to the scene at a time. With computer-aided illustration this approach involves specifying a number of points and invoking a construction command for every line or circle that is desired. In Gargoyle, we try to reduce the effort spent on constructions by constructing many alignment objects at the same time. Our technique takes advantage of two kinds of *coherence* in a design session: often consecutive changes will be made to the same region of the illustration (*spatial* coherence), or the changes will involve similar construction operations (*construction* coherence). If little coherence is present, we provide several ways to turn off alignment object construction. Below, we discuss the creation of alignment objects in detail.

We motivate the construction of multiple alignment objects with two examples. First, consider a user who is interested in extending a mesh of equilateral triangles. He knows that he will be adding edges at 0, 60, and 120 degrees, and that he will placing new vertices relative to existing triangle vertices. He might ask Gargoyle to construct, at each vertex of the existing triangles, three lines of slopes 0, 60, and 120 degrees (see figure 9).

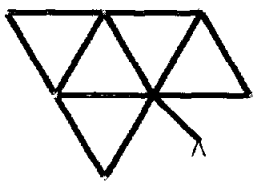

Figure 9 The designer adds another segment to a mesh of equilateral triangles. The alignment lines are shown in gray.

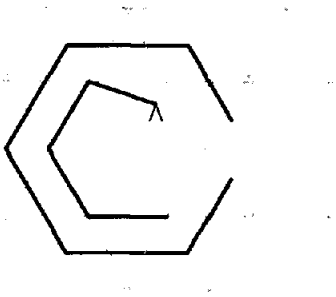

Figure 10 The user builds the inner polygon at an offset of 1/4 inch from the outer polygon. Alignment lines, constructed 1/4 inch from each edge of the outer polygon, are shown in gray.

Now consider a second construction. The user would like to add a polygon that is 1/4 inch from an existing polygon. He might request the construction of all lines that are 1/4 inch away from the existing edges, as shown in figure 10.

The user requests the construction of alignment objects in two steps. First, the user identifies those vertices and segments in the scene at which the alignment lines should be constructed. These vertices and segments are said to be *hot*. Next the user specifies the types of construction to be performed. In the examples, the user asks for lines with specified slopes and lines at a specified offset.

Figure 11 illustrates the process by which alignment objects are computed by Gargoyle. Those vertices and segments that the user has made hot are combined with other vertices and segments suggested by heuristics. The resulting vertices and segments are called the *triggers*; each of them will trigger the construction of alignment objects. From this set, the system removes any vertices or segments that are currently being dragged or rubber-banded, so that all alignment objects will be stationary. Finally, the list of user-specified constructions is consulted. Each specified construction that takes a vertex as an argument *fires* once per vertex, and each specified construction that takes a segment as an argument fires once per segment, creating the final set of alignment objects.

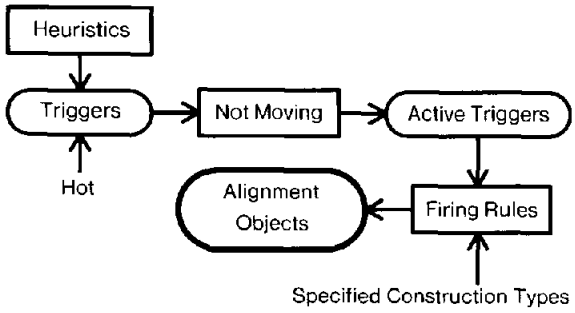

Figure 11 The process of computing the alignment objects.

Below, we give examples of heuristics that we have found useful for augmenting the set of triggers and describe the set of constructions that we have implemented.

We can often save the user the trouble of explicitly making objects hot and cold by inferring the region of interest from his actions. One such inference follows from Observation 1:

> **Observation 1:** When the user moves one point of a polygon, he will often want to align it with other parts of that same polygon.

A Gargoyle heuristic has been implemented from this observation. When the heuristic is activated and the user modifies a polygon, Gargoyle uses as triggers the set of vertices and segments of that polygon. Additional vertices and segments are used as triggers only if the user has explicitly made them hot.

When polygons are being repositioned rather than modified, we cannot use Observation 1. Instead, we rely on a second observation:

> **Observation 2:** When a polygon is translated, rotated, or scaled in entirety, the resulting operation is often performed to make that polygon touch an existing polygon.

No heuristic for augmenting the set of triggers follows from this observation. Instead, the observation suggests that it is important for the scene objects themselves to be gravity-active. In Gargoyle, all scene objects are always gravity-active unless they are being dragged or rubber-banded.

In Gargoyle, we have implemented five types of construction to be used with snap-dragging. Two types of construction are triggered by vertices, and three types of construction are triggered by segments. They are:

> **Vertex Constructions:** Construct a line of a specified slope through the vertex, or construct a circle of a specified radius centered on the vertex.
>
> **Segment Constructions:** Construct the two lines (one on each side) that are parallel to the segment, at a specified distance from it, or construct the two lines (one from each end) that make a specified angle with the segment, or construct the midpoint of the segment.

From the five constructions, we have most of the power of the seven basic relationships. The vertex constructions give us *slope* and *distance*. The segment constructions give us *parallel* lines, *angles*, and one form of *distance ratio* (midpoint). *Vectors* can be made by using slope and distance constructions at the same time. Vertical and horizontal lines at known x and y coordinates are not triggered by vertices or segments but can be requested by a separate mechanism.

The user activates alignment objects by selecting the appropriate slopes, radii, angles, and offset distances from menus, and by turning midpoint construction on or off. New values can be added to the menus by typing them in or by measuring existing slopes, radii, angles, and offset distances in the scene (see section 4.2).

It is important to note that the system is not really guessing which alignment lines to construct; it is acting on requests from the user. We found this was essential for user acceptance. A large number of alignment objects can quickly become overwhelming. We provide several ways for the user to disable the alignment objects: gravity can be turned off, so that the alignment objects do not attract the caret; all hot objects can be made cold; all selected slopes, radii, angles, and offset distances can be deselected; and the heuristics can be turned off.

4.2 Measuring

It is clear how to use our alignment objects to construct line segments of known slope or known length. However, it is less clear how to perform constructions involving relative values such as making line B parallel to line A, where the slope of line A is unknown. One technique that is easy for the programmer to implement and easy for the user to understand, is to allow the user to measure scene quantities.

To make two segments parallel, for example, it suffices to measure the slope of one segment and then adjust the other segment to have the same slope. Solving the problem in this way takes about the same number of keystrokes as it would take to set up and solve the constraint in a constraint-based system, where the user would point at both segments, choose the parallel constraint, and invoke the constraint solver. Figure 12 shows the use of measuring to make one segment parallel to another.

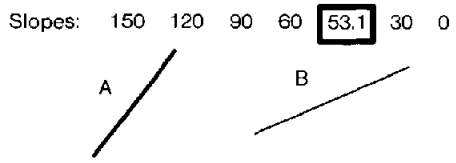

Figure 12(a) Measure the slope of line A, 53.1 degrees, and add it to the menu. Activate alignment lines of this slope.

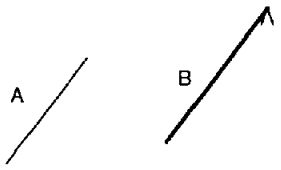

Figure 12(b) Snap one end of segment B onto the new 53.1 degree alignment line to make B parallel to A.

In Gargoyle, the distance between a pair of points, the slope between a pair of points, the angle made by three points, and the distance from a point to a line can be measured and added to the appropriate menu by placing the caret successively on the points of interest and invoking a measuring command.

4.3 Performance

Snap-dragging requires significant rendering and computing resources. Each time the user begins a new operation, all alignment objects must be displayed in less than a second. During transformations, affected objects must move smoothly as they are dragged into position. An additional real-time computational burden is the calculation of points of intersection among all alignment objects whenever the set of alignment objects changes. How can these computations be performed in a reasonable amount of time?

One observation is that the set of alignment objects doesn't usually change much from one moment to the next. Making a vertex hot will add a few alignment objects. Beginning to drag an object will remove any alignment objects that were triggered by its segments and vertices. It makes sense to keep a scan-converted version of the alignment lines around and make small changes to that. Furthermore, we know which line slopes and circle radii are currently of interest. These lines and circles can be scan-converted in advance and then stamped onto the screen as a block of pixels when they are needed. This is already a practical solution for bi-level displays and will become so for color displays.

The problem of smooth motion during dragging is addressed in many commercial systems. Common solutions include reducing detail during dragging (show only a bounding box), or rendering the moving objects into a bitmap that is repeatedly repositioned. These same tricks can be used for snap-dragging.

Calculating the points of intersection of the alignment lines and circles is not particularly time consuming. However, finding the intersections of alignment objects with scene objects and of scene objects with scene objects can be time consuming, since scene objects can be spline curves and other complicated shapes. This cost can be reduced by using spatial sorting techniques (bounding boxes or grids) and by computing only on demand: the intersection calculation can be performed only when both the scene object and the alignment line are within the gravity field of the caret. This same trick can be used to compute the intersections of scene objects with themselves.

5. Conclusion

There are three construction techniques used in computer illustration systems. Grid systems have been commercial successes because they are easy to learn and implement. Constraint-based systems are constantly discussed in the literature because of their great power. We have shown that ruler and compass style construction systems are a viable compromise. Most of the advantages that we claim for snap-dragging are advantages of the ruler and compass approach. In particular, it handles a wide range of alignment types, works at any rotation and scaling, and provides constraints that are easy to add, delete, and modify.

These ideas are beginning to show up in commercial products. Qubix Graphics Systems has begun to demonstrate a technical illustration workstation that employs the ruler, compass, and protractor paradigm.

The obvious drawback of a construction approach is that setting up the constructions may take a long time. Snap-dragging tries to reduce construction time by taking advantage of the repetition that is present in many constructions: often the same slope or distance is needed repeatedly or a polygon's vertex is aligned with other vertices of that same polygon. We believe that snap-dragging is almost as easy to learn as grids, while providing most of the power of constraint systems.

While ruler and compass construction is not inherently computationally expensive, snap-dragging is; many alignment objects must be drawn and many intersection points must be computed. However, the computational requirements are bounded, because the user will turn off alignment objects when the screen becomes too cluttered. Improvements in graphics hardware will make the technique feasible in low-cost workstations in the near future.

Below, we include a number of pictures that have been produced with Gargoyle. Currently, only straight line and arc scene objects are implemented. Figure 13, "Gargoyle Hacker", shows some of the types of precision that snap-dragging makes possible. The fingers of the gargoyle and many of the lines in its wings are parallel even though at odd angles. The bands on the arm are parallel and end exactly on the arm. The gargoyle's pedestal contains a piece of a hexagon. Snap-dragging can be used for traditional ruler and compass constructed letters, as illustrated in figure 14, and for less conventional letters as illustrated in figure 15.

5.1 Future work

Work is in progress to extend snap-dragging to the editing of free-form curves and to three-dimensional objects.

We can easily introduce curves such as Bezier splines that are controlled by points, since we can already place points precisely. In addition, we would like to extend our system to make alignments based on the local tangent to a curve to provide a simple way to position objects on curves themselves, without using control points.

One of the motivations for our work on interactive two-dimensional editing is to develop better ways to edit objects quickly in three-dimensions. In three dimensions, the grid approach is impractical because drawing a full three dimensional grid puts so many lines on the screen that the user has trouble identifying the proper grid point, and notifying the system of the point chosen. Constraint-based approaches are also difficult to use. Some progress has been made in MIT's Mechanical Engineering Department towards using constraints in three dimensions to alter dimensions in mechanical assemblies [Lin81] [Serrano84]. It is more difficult to use constraints to provide a quick-sketch capability. In three dimensions, there is an explosion of degrees of freedom that need to be constrained, requiring a large number of constraints. A modified version of snap-dragging may be successful in providing a precise three-dimensional quick-sketch capability.

Acknowledgments

Eric Bier gratefully acknowledges support from an AT&T Bell Laboratories fellowship and from a Xerox PARC research internship. Both authors are indebted to Xerox PARC for the research environment which made this work possible. We would also like to thank Ken Pier for his many contributions to Gargoyle and for his helpful comments on this paper. Finally, thanks to Subhana Menis for meticulous proof-reading under pressure.

References

[Borning79] Alan Borning. *Thinglab—A Constraint-Oriented Simulation Laboratory.* Revised Ph.D. thesis. Report SSL-79-3, Xerox PARC, Palo Alto, CA 94304. July 1979. Also available as Stanford CS Dept. Report STAN-CS-79-746.

[Goines82] David Lance Goines. *A Constructed Roman Alphabet.* David R. Godine, publisher. 306 Dartmouth Street, Boston MA 02116. 1982.

[Lin81] V. C. Lin, D. C. Gossard, and R. A. Light. Variational geometry in computer-aided design. *Computer Graphics* 15(3):171-177, August 1981. SIGGRAPH '81 Proceedings.

[Lipkie82] Daniel E. Lipkie, Steven R. Evans, John K. Newlin, and Robert L. Weissman. StarGraphics: an object-oriented implementation. *Computer Graphics* 16(3):115-124, July 1982. SIGGRAPH '82 Proceedings.

[MacDraw84] MacDraw Manual. Apple Computer, Inc. 20525 Mariani Ave., Cupertino, CA 95014. 1984.

[Nelson85] Greg Nelson. Juno, a constraint-based graphics system. *Computer Graphics* 19(3):235-243. July 1985. SIGGRAPH '85 Proceedings.

[Newman79] William M. Newman and Robert F. Sproull. *Principles of Interactive Computer Graphics.* chapter 12. McGraw Hill, second edition, 1979.

[Opperman84] Mark Opperman. A Gremlin Tutorial for the SUN Workstation. Internal document, EECS Department, UC Berkeley, Berkeley CA 94720.

[Pavlidis85] Theo Pavlidis and Christopher J. VanWyk. An automatic beautifier for drawings and illustrations. *Computer Graphics* 19(3):225-234. SIGGRAPH '85 Proceedings.

[Pier83] Kenneth A. Pier. A retrospective on the Dorado, a high-performance personal computer. Proceedings of the 10th Symposium on Computer Architecture. SigArch/IEEE, Stockholm, pages 252-269, June 1983.

[Serrano84] David Serrano. MATHPAK: An interactive preliminary design system. Master's thesis, MIT Mechanical Engineering Department. 1984.

[Swinehart85] Daniel C. Swinehart, Polle T. Zellweger, and Robert B. Hagmann. The structure of Cedar. SIGPLAN Notices 20(7):230-244, July 1985. Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments.

[Stone80] Maureen Stone. How to use Griffin. Internal Memo, Xerox PARC, 3333 Coyote Hill Rd, Palo Alto CA 94304. 1980.

[Sutherland84] Ivan E. Sutherland. Sketchpad, a man-machine graphical communication system. In Herbert Freeman, editor, *Tutorial and Selected Readings in Interactive Computer Graphics,* pages 2-19. IEEE Computer Society, Silver Spring, MD. 1984. Reprinted from AFIPS 1963.

Figure 13 The Gargoyle Hacker. (drawn by Maureen Stone)

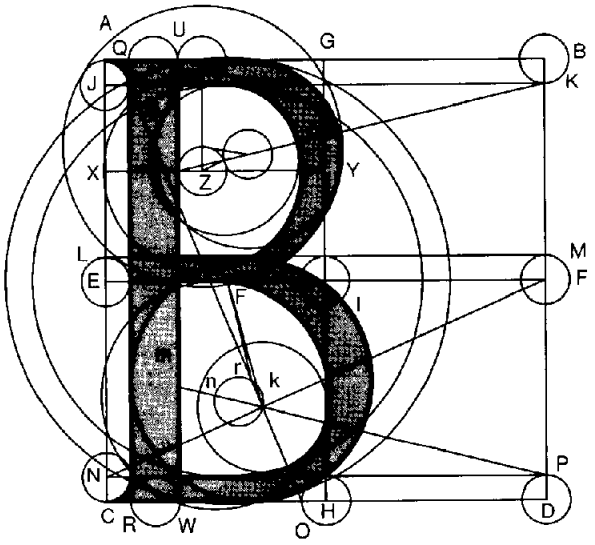

Figure 14 A constructed Roman letter B. The steps of the construction were taken from a book of constructed Roman letters, by David Lance Goines [Goines82]. (drawn by Eric Bier)

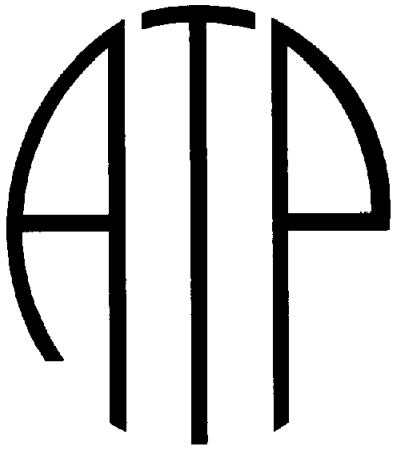

Figure 15 A logo for the Ridge Vineyards Advanced Tasting Program. (drawn by Ken Pier & Maureen Stone)