Snap-Dragging: Interactive Geometric Design in Two and Three Dimensions

Eric A. Bier

ABSTRACT

A new technique has been developed for designing precise two and three-dimensional shapes rapidly and interactively. A synthesis of the best properties of grid-based systems, constraint networks, and drafting has resulted in a new technique called ''snap-dragging''.

To aid precise construction, a set of lines circles, planes, and spheres, called 'alignment objects' are constructed by the system at a set of slopes, angles, and distances specified by the user. These alignment objects are constructed at each vertex or edge that the user has declared to be 'hot' (of interest). Vertices and edges can also be made hot by the system automatically through some heuristics.

The user can snap the cursor onto these alignment objects, onto their points or lines of intersections, or onto scene objects using an adjustable 'gravity mapping'. Finally, the user can translate, rotate, scale, and skew shapes, by specifying which operation is desired, then grabbing a point on the object and dragging it to a suitable, precise location.

Snap-dragging was encorporated into a prototype two-dimensional illustration system at Xerox PARC which has been used successfully by a community of users since 1986.

May 19, 1988

Acknowledgments

I would like to thank a number of people and organizations who have supported and continue to support this work. Thanks to Carlo Sèquin, my advisor at U.C. Berkeley, for providing ample resources, encouragement, prompt feedback, and good advice. Thanks to Larry Rowe and Alice Agogino, my other committee members, for good suggestions on the dissertation, particularly for suggestions that led to a more complete account of the experiences of users, and to a more satisfactory introduction and conclusion. Thanks to Jock Mackinlay for excellent comments on the earliest of drafts. Thanks to Ken Pier for turning Gargoyle into a real illustrator, for reading the dissertation with his eagle eyes, and for being a pleasure to work with. Thanks to John Ousterhout and Brian Barsky for participating in my oral exam and listening to these ideas when they were just beginning to gel. Special thanks to Maureen Stone for adopting snap-dragging early on, co-authoring my first SIGGRAPH paper, contributing ideas and code to many parts of Gargoyle, and for lots of encouragement.

I gratefully acknowledge support from AT&T Bell Laboratories, which provided fellowship support during the early stages of this work, and to Xerox PARC for providing office space, computers, and financial support for the last two years and for many summers before that.

Thanks to the other members of the Imaging Area of the Electronic Documents Laboratory at Xerox PARC (formerly the Imaging Area of the Computer Science Laboratory) for providing some of the world's best tools for preparing a computer graphics dissertation. Thanks to Subhana Menis for proofreading this dissertation as the deadline approached. Thanks to Michael Plass for thinking of the name "snap-dragging" one day over lunch. Thanks to all of the users of Gargoyle for answers to questionnaires, suggestions, bug reports, for sharing their favorite constructions, and for keeping me honest. Thanks to Andrew Glassner, Polle Zellweger, Jock Mackinlay, Pavel Curtis, Maureen Stone, David Kurlander, Steve Wallgren, and Doug Wyatt for contributing the illustrations that appear in Chapter 6. Particular thanks to Steve Wallgren, PARC's prolific artist, for fearlessly using research software, Gargoyle included.

Table of Contents

1. Introduction **1**

1.1 The Problem: Making Precise Geometric Models Quickly 1

1.2 A Refinement of the Problem: Scene Composition 4

1.3 Goals of the work 4

1.4 Contributions 6

1.5 Structure of the Dissertation 7

2. Previous Work **9**

2.1 Precise Point Placements 9

2.2 Precise Affine Transformations 16

3. Snap-Dragging: A User's View **18**

3.1 Snap-Dragging in Two Dimensions 18

3.2 Snap-Dragging in Three Dimensions 42

3.3 Feedback: Graphical and Textual 59

3.4 Selecting Objects 62

3.5 Unusual Constructions 64

4. Snap-Dragging Implementation **66**

4.1 Gravity 67

4.2 Alignment Objects and Gravity-Active Scene Objects 84

4.3 Interactive Transformations 95

5. Productive Scene Composition **99**

5.1 The Elements of Productivity 99

5.2 Comparison of Several Techniques 103

6. Experiences with Users **117**

6.1 A Brief History of the Gargoyle Project 117

6.2 Feedback from Users 119

6.3 Some Operation Count Data 126

6.4 A Picture Gallery 130

6.5 Discussion 137

7. Plans for Future Work **141**

7.1 A Duality: Moving Lines to Points vs. Moving Points to Lines 141

7.2 Symmetry Operations 144

7.3 Positioning the Camera 146

7.4 Improving the User Interface 146

8. Summary **148**

References **152**

A. Gravity Algorithm Details **157**

1. Introduction

There once was a Square, such a square little Square,
And he loved a trim Triangle;
But she was a flirt and around her skirt
Vainly she made him dangle.
Oh he wanted to wed and he had no dread
Of domestic woes and wrangles;
For he thought that his fate was to procreate
Cute little Squares and Triangles.

Now it happened one day on that geometric way
There swaggered a big bold Cube,
With a haughty stare and he made that Square
Have the air of a perfect boob;
To his solid spell the Triangle fell,
And she thrilled with love's sweet sickness,
For she took delight in his breadth and height—
But how she adored his thickness!

So that poor little Square just died of despair,
For his love he could not strangle;
While the bold Cube led to the bridal bed
That cute and acute Triangle.
The Square's sad lot she has long forgot,
And his passionate pretensions . . .
For she dotes on her kids-- Oh such cute Pyramids
In the world of three dimensions.

— Robert Service
"Maternity"

1.1 The Problem: Making Precise Geometric Models Quickly

Graphic artists, mechanical designers, architects, animators, authors of technical papers and others create geometric models as a major part of their daily efforts. Typically, some part of this model construction must be done with precision. For instance, certain line segments should be horizontal, parallel or congruent. In the last 25 years, it has become possible to construct precise geometric models interactively using a computer. Two-dimensional illustration systems, drafting systems, and solid modelers are readily available from commercial vendors.

Unfortunately, building precise geometric models with these systems is often awkward and time-consuming. The chief tools provided by two-dimensional design systems—grids, constraints, and ruler and compass (drafting)—have significant limitations. Grids provide only a small

fraction of the desired types of precision. Constraints, while very powerful, require the user to specify constraint networks, which are often difficult to understand and time-consuming to manipulate. Drafting systems generally require a tedious sequence of construction steps to complete a precise drawing. The chief tools provided by three-dimensional design systems—dial-controlled affine transformations and large menus of special-purpose multi-argument functions—also have limitations. The dials usually don't provide precise transformations at all, and the special-purpose functions take a long time to master and are inherently time-consuming because of all the menu manipulations required. Kasik [Kasik82] discusses techniques for helping the user cope with large menus.

Finally, none of the available techniques can be efficiently used for both two- and three-dimensional editing. While one could edit a two-dimensional scene with dials and special-purpose functions, this would be needlessly tedious. Likewise, one could edit a three-dimensional scene with a three-dimensional grid, with constraints, or with drafting, but the grid would be unwieldy, a large number of constraints would be needed to handle all of the degrees of freedom present in three dimensions, and the drafting approach would be even more tedious than in two dimensions.

This dissertation describes a technique, called *snap-dragging*, that can be used to precisely position points and scene objects relative to each other. Snap-dragging uses a ruler and compass approach to achieve a compromise between the power of constraints and the convenience of grids. By doing many of the construction steps for the user, snap-dragging reduces the time for constructions compared to traditional drafting approaches. Furthermore, it is readily applicable to geometric design in both two and three dimensions. Some early work on snap-dragging in two dimensions has been published [Bier86].

Figure 1-1 shows a short example of snap-dragging at work. The user wishes to modify an isosceles triangle to be an equilateral triangle. He selects the points A and B and invokes a command to make them *hot*. Hot points, shown with white squares around them, cause alignment lines and circles to be constructed when the user requests them. For instance, here the user has activated circles of radius 2/3 inches. Two circles are constructed automatically around the points A and B. The software cursor, the *caret*, shown as an upside-down "v" can be placed precisely on point C (Figure 1-1(a)). The user can *interactively translate* point C by moving a hardware cursor, shown as a small dark circle. Point C and the caret move together. When the

hardware cursor comes close enough to the intersection of the two circles, the *gravity function* places point C precisely on this intersection point (Figure 1-1(b)). Because the distance AB was 2/3 inches, the triangle is now equilateral. More examples of snap-dragging will be given in Chapter 3.

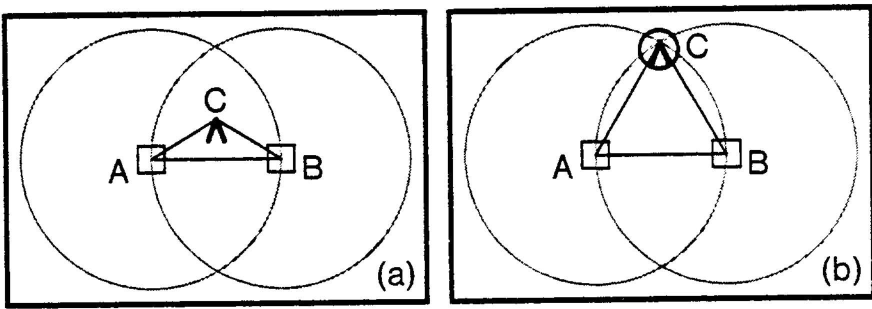


Figure 1-1. (a) Points A and B of triangle ABC are *hot*. They trigger two *alignment circles*. The software cursor, the *caret*, is shown on point C. (b) Point C is *interactively translated* until the caret *snaps* to the intersection of the two circles, producing an equilateral triangle.

Snap-dragging is the combination of three interactive techniques that work very well together—gravity, affine transformations that are parameterized by the cursor, and alignment objects. The gravity function snaps a software cursor to points, curves, surfaces, and their intersections. Objects are translated, rotated, and scaled by selecting them and moving the cursor; the amount to translate, rotate or scale is computed from the cursor motion. Because the cursor continues to obey gravity during these transformations, the transformations can be specified precisely. Finally, alignment objects are lines, circles, planes and quadric surfaces created by the system to help the user construct a shape, much as intermediate lines and circles are used in traditional compass and straightedge constructions.

Snap-dragging has been implemented in Gargoyle [Bier86, Pier88], a two-dimensional illustration system, and Gargoyle3D, a three-dimensional illustration system. Both Gargoyle editors produce poster art and document illustrations. They are implemented in the Cedar environment [Swinehart86] running on a Dorado high-performance personal workstation [Pier83]. Gargoyle was implemented by the author in 1985 in order to try out snap-dragging in two dimensions but has since grown into a full-fledged illustrator, with contributions from Ken Pier, Maureen Stone, Michael Plass, David Kurlander, and John Eisenman. Gargoyle3D is a revision of the Solidviews three-dimensional illustrator [Bier83], to which snap-dragging operations have been added.

1.2 A Refinement of the Problem: Scene Composition

One can imagine many operations that position objects precisely relative to other objects, and one can imagine many kinds of precise relationships that might be desired. This dissertation focuses on a specific set of operations and relationships. In particular, two activities are considered: (1) placing a point in the drawing space such that desired angle and distance relationships are achieved with one or more existing points and (2) specifying an affine transformation (e.g., translation, rotation, scaling, or skewing) such that desired angle and distance relationships are formed between a small number of points on the object being transformed and a small number of points in the rest of the illustration. In this dissertation, these two activities, taken together, are called *scene composition.*

This refinement of the problem of constructing precise geometric models makes the general problem more concrete without restricting it unduly. Most positioning operations available in computer illustration systems can be decomposed into sequences of these two activities.

Scene composition is an important part of many tasks that involve producing a geometric model, including technical illustration, graphic arts, computer-aided mechanical design, desktop publishing, and animation. Scene composition actually takes place at several different phases of geometric design. Precise point positions can be used as parameters to define the shape of primitive objects (e.g., a quadrilateral can be defined by four points) or to position two assemblies relative to each other to form a composite assembly.

1.3 Goals of the Work

This dissertation describes the design of an interactive scene composition technique that is intended to achieve four major goals:

1) Precision. Facilitate construction of precise geometric objects and precisely composed collections of objects.

2) Speed. Increase the rate at which precise geometry can be created. In particular, increase the number of points that can be precisely placed per unit of time.

3) Applicability. Provide a system that can be used by a wide class of users, including artists, engineers, scientists, and technical writers.

4) Two dimensions or three dimensions. Make a system that builds either two-dimensional shapes or three-dimensional shapes with essentially the same primitive operations.

To accomplish these goals, a number of techniques are brought to bear from the fields of computer science, mathematics, and user interface design. These techniques can be roughly divided into five categories:

1) The drafting paradigm. The tools of the draftsman, such as ruler, compass, protractor, and T-square, are an appealing foundation on which to build a computer-aided geometry system for several reasons. The tools and their interactions are familiar to people who have taken high-school geometry. The lines and circles of compass and straightedge construction are easy to draw on computer displays, and consequently are easy to point at with a screen cursor.

2) System-wide use of the cursor point. In general, a geometric design system provides a set of commands that position the software cursor and another set of commands that use the position of the cursor as an argument. If enough commands work in this way, an amplification occurs. New cursor-positioning commands benefit all of the commands that rely on cursor position. New commands that use the cursor position automatically take advantage of all of the cursor-positioning commands.

3) Ubiquitous graphical feedback. Graphical feedback in the same drawing space with the objects being designed can keep the user informed of non-geometric properties of these objects and of the state of the drawing session. In addition, when this feedback is superimposed on the graphical presentations of the objects, it allows the user to quickly associate an object's properties with the object itself. This association would be more difficult to discover if, for instance, a textual description of the properties of all of the objects were displayed next to the drawing space.

4) Computational leverage. Use computer cycles to reduce the effort that the user must expend communicating with the machine. Perform operations even without being asked, in the hopes that they will be of use to the user. Compute several possibilities and let the user select the one that was meant, instead of waiting for the

user to spell it out. For instance, in snap-dragging many alignment objects (lines, circles, planes, and spheres) and intersection curves are computed even though the user will only use a few of them.

5) Sketch and constrain at the same time. There is a certain appeal to roughing in a shape first and tidying it up later. However, two major benefits come from imposing constraints while sketching. First, the total number of commands needed to complete a shape may be reduced because sketching commands are doing double duty. Second, to sketch in three dimensions with a two-dimensional pointing device, we must use some constraint to provide the third coordinate of the cursor position. If this constraint correctly positions the object points at the same time, labor can be saved.

In addition, the implementations of snap-dragging described here rely on a flexible set of user interface tools including extensible menus, mode indicators, and a mouse pointing device.

1.4 Contributions

I believe that the following aspects of this work are novel contributions to interactive geometric design:

1) *MultiMap: gravity with three-way arbitration.* Without changing mapping functions, the user can snap a software cursor to points (e.g., vertices, control points, and intersection points), curves (e.g., edges, lines and circles), and surfaces (e.g., polygons, planes, and spheres). Adjustments to this mapping function, called MultiMap, allow the cursor to be placed more easily onto points, curves, or surfaces as desired.

2) *Gravity with on-the-fly intersection points.* The gravity function computes intersection points (e.g., of curves with curves and curves with surfaces) on the fly. By rejecting unpromising curves and surfaces early on, the gravity function is computed with only a very small number of intersection computations per cursor position. In effect, once gravity for curves and surfaces is available, intersection points come for free.

3) *Hot points, edges and faces.* The user can specify those points, edges and faces with which he intends to align other shapes during subsequent constructions.

4) *Alignment objects by Cartesian product.* At each hot point and hot edge in the scene, temporary lines and circles (or lines, planes, and spheres in three dimensions), called *alignment objects*, are constructed, one for each slope, radius, distance, or angle that the user has activated. Because many alignment objects are created at once, the effort expended by the user to align shapes precisely is small when averaged over all of the constructions needed to make an illustration.

5) *The automatic hotness rule.* When this rule is active, the stationary parts of objects that are being modified become hot automatically. This rule reduces the time needed to construct an object whose vertices and control points form precise relationships with each other.

6) *Applicability to both two and three dimensions.* The availability of alignment objects, such as lines, planes, and spheres in three dimensions makes it possible to place the software cursor in 3-space with a two-dimensional pointing device in a single perspective view. As a result, it is possible to construct precise shapes in both two and three dimensions using essentially the same commands and receiving essentially the same graphical feedback.

1.5 Structure of the Dissertation

The remainder of this dissertation is organized as follows:

Chapter 2 describes previous work in interactive geometric design systems.

Chapters 3 and 4 describe snap-dragging in detail. Chapter 3 describes the technique as seen by the user, both in two and three dimensions. Chapter 4 describes the implementation of the three snap-dragging sub-techniques: gravity, alignment objects, and interactive transformations.

Chapter 5 describes three strategies for improving the productivity of a geometric designer and compares the extent to which grid, constraint, and snap-dragging systems make use of these strategies. In the process, different geometric design tasks are categorized and appropriate scene composition techniques are suggested for accomplishing each task.

Chapter 6 describes the use of snap-dragging by a community of users. The performance of snap-dragging in this community is evaluated by considering responses to questionnaires, recorded user sessions, and a few of the pictures that have been produced.

Chapter 7 describes additional work that is suggested by the results described in this

dissertation. In particular, it discusses extensions to snap-dragging that would better support editing of curved objects, symmetrical objects, and of the viewing position. Finally, it describes on-going work aimed at improving the user interface to snap-dragging.

Chapter 8 summarizes the dissertation.

2. Previous Work

> He learned by heart the fantastic legends of the crumbling books, the synthesis of the studies of Hermann the Cripple, the notes on the science of demonology, the keys to the philosopher's stone, the *Centuries* of Nostradamus and his research concerning the plague, so that he reached adolescence without knowing a thing about his own time but with the basic knowledge of a medieval man.
>
> – Gabriel Garcia Marquez
> *One Hundred Years of Solitude*

Interactive geometric design systems have been in use for over twenty years. These systems have provided the user with several mechanisms for communicating precise shape descriptions to the computer. Communicating a precise shape description involves describing points and affine transformations precisely. Existing mechanisms for describing points precisely include constraints, grids, ruler and compass techniques, and gravity. Existing mechanisms for describing affine transformations precisely include manipulating dials (or other continuous valuators), defining the transformation in terms of object vertex positions, or modifying the transformation based on the position of a two-dimensional cursor (controlled by a mouse or tablet). The two sections of this chapter present examples of mechanisms for placing points and applying affine transformations, respectively.

2.1 Precise Point Placement

2.1.1 Constraints

A number of geometric design tools allow the user to define the positions of shape points with a system of simultaneous equations. By changing one or more equations and invoking a constraint solver, the user can make global changes to a shape while preserving important shape properties. There are four main problems associated with this approach:

1) In general, the constraint solver generates an appropriate new shape only if it has a sensible shape from which to start. In particular, there must be a way to sketch in an initial shape. In three dimensions, this is difficult.

2) There may be multiple solutions requiring interaction with the user to choose the desired one.

3) Building up and debugging the network of constraints can be time-consuming.

4) Solving the simultaneous equations may take noticeable computation time.

While these problems are far from solved, the systems described below have made significant progress.

Sketchpad, described in Sutherland's seminal dissertation [Sutherland63a, Sutherland63b], introduces all at once a set of techniques for representing shapes and a set of techniques for interactive editing, many of which are still in use. Holding a light pen, a user can sketch lines and arcs. A gravity function can be used to snap one object to another. Sketchpad preserves the resulting connectivity if desired by periodically running a constraint solver. Other constraints can be added to the scene, such as making line segments congruent, parallel, horizontal, or vertical. Such constraints are displayed, if desired, with a characteristic picture for each constraint including dashed line segments pointing to the scene parts that are being constrained. Figure 2-1(a) shows a quadrilateral with its left and right edges constrained to be parallel and congruent, as they might appear in Sketchpad. When the constraints are solved, the quadrilateral becomes a parallelogram like the one shown in Figure 2-1(b).

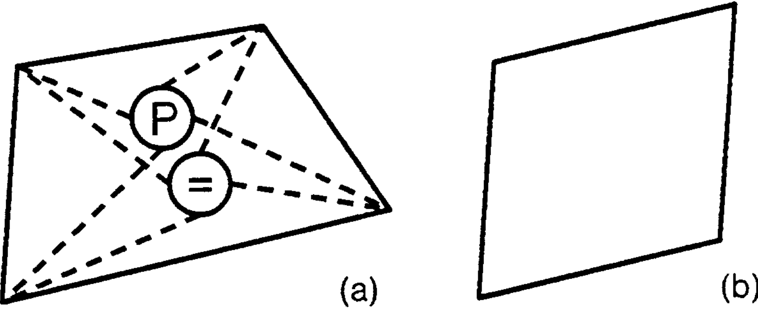


Figure 2-1 (a) Characteristic pictures are used to display constraints. Here "P" stands for the "parallel" constraint, and "=" for the "congruent" constraint. Redrawn from Figure 9 of Sutherland's paper [Sutherland63b]. (b) When the constraints are solved, a parallelogram results.

Sketchpad uses a relaxation solver. While the relaxation method can solve very general systems of equations, it is relatively slow and does not always converge to the intended solution. To get better performance in such a system, the user can structure the scene hierarchically so the constraints can be solved in small manageable chunks. The time invested in building the constraint network and structuring the scene pays off if the model is to be modified repeatedly, as

are the bridge trusses in Sutherland's simulations. However, this investment may not pay off when constraints are used to design technical illustration, for example. Sketchpad was only used for designing two-dimensional shapes. Extending it to three dimensions would require the addition of a technique for sketching in the three-dimensional geometry that is to be constrained.

Borning extends Sutherland's work with his Thinglab constraint-oriented simulation laboratory [Borning79]. Borning augments the Smalltalk language [Goldberg83], adding constructs to represent hierarchies of parts and the constraints on them. He concentrates on systems of constraints that can be solved by propagation. In this case, solution is fast and there is only one solution. When a particular picture part is interactively moved, Thinglab compiles a set of Smalltalk routines to recompute the picture. After a small delay for compilation, the picture rubber-bands in real time. This automatic programming approach quickly solves simple constraint systems, but falls back on the relaxation method to solve constraint networks that have cycles.

Thinglab is an interesting experiment in real-time simulation. A Thinglab user can combine ready-made components, whose constraints were designed by an expert, to try out a simulation. However, the appropriateness of Thinglab for geometric design is less clear. Borning does give geometric examples in his dissertation, but they employ only simple linear constraints such as the "midpoint" constraint. Of course, Thinglab may include (or could be extended to include) other geometric constraints, such as "parallel" and "congruent" that could be used in more general geometric constructions, such as those encountered in drafting applications. However, even with this extension, Thinglab fails to address most of our concerns. The user is still faced with the time-consuming process of constraining the degrees of freedom of his design. Furthermore, constraint networks for geometric design will, in general, have cycles and non-linear constraints. When cycles and non-linear equations are present, ThingLab falls back on the relaxation solver, leaving us with the problems of multiple solutions and relatively long computation times. Finally, we still need a way to sketch shapes in three dimensions.

A body of work on the use of constraints for the design of three-dimensional objects has come out of an on-going project at the Computer Aided Design Laboratory in the Mechanical Engineering Department at MIT.

Building on the work of Hillyard and Braid [Hillyard78], the work of Light in two dimensions

[Light80, Light82] and Lin [Lin81a, Lin81b] in three dimensions led to the *variational geometry* technique, whereby three-dimensional objects are parameterized by points (e.g., a cylinder might be defined by three points, two to determine the centers of its two disks and a third to determine its radius) and all degrees of freedom are accounted for either with parameterless constraints (e.g., constraints that line segments be parallel or congruent) or with dimensioning constraints (e.g., "length = 3 inches"). Parts described in this way can be modified by varying numerical inputs to the dimensioning constraints (e.g., changing the length). For changes that effect only a small number of points, the constraints are solved rapidly using sparse matrix methods [Light82].

Serrano [Serrano84] illustrates the utility of the variational geometry approach in mechanical design. By allowing constraints on named real-valued variables as well as on three-dimensional points, Serrano's MATHPAK allows the input values of dimensioning constraints to depend on variables that are of interest to mechanical engineers such as pressure or volume. The applicability of Serrano's work to other domains is less clear. Furthermore, MATHPAK's constraint solver requires that the user provide a complete and consistent constraint network for all degrees of freedom in the scene. Building such a network can be tedious.

This tedium can be reduced by creating or deleting some constraints automatically. Congdon [Congdon82] describes an algorithm for discovering geometric properties of a given polyhedral scene, including parallelism of faces and perpendicularity of faces, and building up parameterless constraints that preserve these properties. Lee's system [Lee83] derives a complete set of constraints for a restricted set of scenes built of quadric shapes, given dimensioned drawings of three orthogonal views. Chyz's system [Chyz85] system decides, when a new constraint is added, which of the existing constraints to throw out in order to preserve the completeness and consistency of the constraint network. Unfortunately, Lee's approach requires that dimensioned drawings already exist, and all three approaches require that the user understand the changes to the constraint network that are being made automatically.

Congdon's DIM3IN program [Congdon82] includes a way to sketch in the original geometry of three-dimensional objects using sketch recognition. The user roughs in an axonometric projection of the desired shape. If enough of the lines of the shape are parallel to the projected x, y, and z axes of the reference coordinate frame, the system can compute coordinates for all shape vertices. Otherwise, the user must add extra guiding lines to help the system find a solution. The

resulting shapes can be tidied up with constraints. This is an effective input scheme for shapes that are chiefly aligned with the axes of the reference frame. It is not clear how to extend the scheme to more general shapes.

Another approach to making constraint-based editing more attractive is to reduce the time needed to specify each constraint. In Nelson's two-dimensional picture editor, Juno [Nelson85], there is a mode for adding each of a small number of constraints (horizontal, vertical, congruent, parallel, frozen, or counter-clockwise). Once in a mode, the user can add several constraints of the associated type by clicking the mouse over several groups of scene points in succession. In addition, Juno solves under-constrained systems, so one need not constrain all degrees of freedom. Even so, many keystrokes are required, and for the examples I have tried, all or nearly all degrees of freedom must be constrained before the objects settle into the desired shape.

The chief remaining problems of the constraint approach are the need for a good way to rough in general shapes in three dimensions, and the time needed to build and debug the constraint network. Snap-dragging can be used to sketch directly in three dimensions, making it a candidate to be used with existing constraint systems. When used alone, snap-dragging uses two strategies to reduce the time needed to control the many degrees of freedom found in geometric models. In particular, it reduces the number of keystrokes needed to constrain an individual point and provides operations, namely affine transformations, that effectively constrain many points at once.

2.1.2 Grids

Many design systems provide a mode in which all mouse or tablet coordinates are rounded to the nearest point on a regular grid before being used to control an interactive operation. Typically, the user can vary the spacing of the grid points and can turn the grid on and off. Rectangular grids are the most common but some commercial systems, such as AutoCAD(R) [Lubow87] include skewed grids to facilitate isometric drawings. Other systems that use grids include MacDrawTM [MacDraw84], Griffin [Baudelaire80], Draw [Baudelaire79], and Gremlin [Opperman84].

Grids are popular because they are easy to use (and easy to program). Several factors contribute to making grids easy to use.

1) Pointing with a mouse is easier than typing coordinates.

2) Interactive operations, such as placing a new vertex, are invoked in the same way with the grid on as with the grid off. The only new operations that the user must learn are the few operations that control the grid.

3) The idea of using a grid is familiar to anyone who has used graph paper.

The problem with grids is that they lack power. Precise shapes as simple as an equilateral triangle cannot be constructed with all of their vertices on a regular rectangular grid. Furthermore, grids tend to be hard to use on illustrations that have been rotated or scaled by arbitrary amounts.

Snap-dragging is similar in flavor to a grid approach. The user points with a mouse, operations are invoked the same way with gravity on or off, and a small set of operations are provided for turning gravity on and off. On the other hand, snap-dragging overcomes the limitations of grids by allowing the cursor to snap to a more general structure than a regular grid, namely to a collection of alignment objects that play a role similar to lines and circles in compass and straightedge constructions.

2.1.3 Ruler and Compass Constructions

Several drawing systems incorporate construction commands for positioning auxiliary objects such as lines and circles. Ellis's layout editor [Ellis83] and CIMLINC's CIMCAD drafting system [Newell85] edit two-dimensional scenes in a ruler and compass fashion, and the Jesse editor at U.C. Berkeley edits three-dimensional scenes [Siegel86] in this fashion.

CIMCAD uses a stack-based language of construction commands. The language includes commands that compute the line passing through two points, the line tangent to a circle at a point, the intersection point of two circles and so on. The result of each command is placed on a stack for use by subsequent commands. Construction routines written in this language can be invoked from an extensible menu. Jesse incorporates a similar command language to describe constructions directly in three dimensions.

Such languages are useful design tools, but do not solve our interactive design problems. By implementing new routines in the construction language, users can build up a library of commands appropriate to their application domains. However, tailoring the systems requires time to write the commands, to add new commands, and to invoke the commands. Furthermore,

as the number of commands grows, the interface can grow cluttered. Using the operations requires selecting scene parts and finding an appropriate menu button for each step in the construction.

Ellis's editor makes some common construction operations less tedious. For instance, in a certain mode, it allows the user to align a new point horizontally or vertically with any existing point. Horizontal and vertical alignment is a special case of the "slope" alignment lines in snap-dragging.

Snap-dragging takes advantage of the power of ruler and compass constructions. However, it provides only a small number of primitive operations, which reduces the features that the user must learn. To facilitate many common constructions, snap-dragging provides a large variety of alignment object types. To reduce construction time, it constructs many alignment objects at once, making it possible to place a number of objects precisely after investing a short time for alignment object set-up.

2.1.4 Gravity

Many geometric design systems use a gravity function to help the user place a software cursor on vertices, edges, surfaces, and their intersections. In published descriptions, GRIN [Wolfe81, Fitzgerald81], GMSolid [Boyse82], Jesse [Siegel86], Adobe Illustrator™ [Adobe87], and Xerox Pro Illustrator™ [Xerox88] all use a pointing device to snap the cursor to vertices. Draw [Baudelaire79], Sketchpad [Sutherland63a, Sutherland63b], and Sketchpad III [Johnson63a, Johnson63b] implement gravity functions that snap the cursor to lines and curves. Solidviews [Bier83, Bier87] allows the cursor to be placed on object surfaces.

Gravity in snap-dragging allows the cursor to be snapped to vertices, curves, surfaces, and their intersections. This gravity function is similar to the "cylinder mode" of the Pen Space Location program in Sketchpad III. It generalizes that technique to work under perspective projection and to snap the cursor to surfaces and their intersections.

2.1.5 Defining Objects with Points

The ability to specify points precisely can be used to extra advantage in a system where the scene shapes are *defined* in terms of a set of points. For instance, in GRIN [Wolfe81,

Fitzgerald81] and in Lin's work on variational geometry [Lin81a, Lin81b], a sphere is specified by a point that defines its origin and a second point that defines its radius, and in Juno, an arc is specified by its two endpoints and a third point on the arc.

Because snap-dragging is good at placing points precisely, it shares this ability to edit all point-defined shapes. This class of shapes includes conic curves, quadric surfaces, spline curves and spline surfaces.

2.2 Precise Affine Transformations

2.2.1 Dials

Dials, or other continuous valuators, can be used to indicate how much to translate, rotate, or scale an object. Typically, the user turns the dial until the objects look right. With rapid screen refresh, the objects appear to move smoothly. Effectively, the user can try out hundreds of closely related transformations using visual feedback to pick the correct one. The following two- and three-dimensional editors have used dials: Sketchpad, GRAMPS [O'Donnell81], SCOT [Upstill85], Jessie [Siegel86], and Parent's three-dimensional data sculpting technique [Parent77].

Dials demonstrate three important user interface ideas.

1) By investing many computational cycles to present information to the user, we can simplify the actions that the user must perform to express a design parameter. In this case, the action required is turning a dial.

2) Smooth motion gives the user a good sense of the shapes of objects, particularly three-dimensional ones.

3) Because the human eye is good at discriminating smoothly moving objects from stationary ones, smooth motion allows the user to confirm that the correct set of objects is being repositioned.

Unfortunately, dials are inadequate for precise editing because they rely on the user to decide what "looks right." Snap-dragging invests computational cycles to make simple user input possible, and uses smooth motion transformations. However, with snap-dragging, the transformations can be terminated at precise locations.

2.2.2 Transformations Parameterized by a Point

To describe precise translations, rotations, and scaling operations, the user must specify the vector to translate by, the angle to rotate through, or the factor to scale by. While these values can be typed, it is often faster and more intuitive to specify them indirectly in terms of points on scene objects. For instance, one might translate an object through the displacement vector from a vertex on one object to a vertex on another object, or scale by the ratio of the lengths of two scene line segments. This idea is used in Solidviews [Bier83, Bier87], GRIN [Fitzgerald81, Wolfe81], and is described in Nielson's article on manipulation techniques [Nielson87].

If the points that are used to parameterize the transformations are just different positions of the software cursor, then smooth dragging, cursor positioning, and transformation operations are unified. This technique is used in a number of modern illustrators including MacDraw [MacDraw84], Adobe Illustrator [Adobe87], and Xerox Pro Illustrator [Xerox88].

Gargoyle and Gargoyle3D take this idea a step further by allowing the cursor to be under the control of a sophisticated gravity function during the transformations.

3. Snap-Dragging: A User's View

By the Golden Egg of Faranth
By the Weyrwoman, wise and true,
Breed a flight of bronze and brown wings,
Breed a flight of green and blue.
Breed riders, strong and daring,
Dragon-loving, born as hatched
Flight of hundreds soaring skyward,
Man and dragon fully matched.

– Anne McCaffrey
Dragonflight

This chapter is an extended example of snap-dragging. Section 3.1 illustrates the interface in two dimensions, and section 3.2 illustrates the interface in three dimensions. Sections 3.3, 3.4, and 3.5 describe properties of snap-dragging that apply in both two and three dimensions, namely graphical and textual feedback, selection operations, and unusual construction techniques, respectively.

3.1 Snap-Dragging In Two Dimensions

This section illustrates the notions of gravity, affine transformations and alignment objects in two-dimensional snap-dragging. In addition, it describes some of the user interface features that work together with snap-dragging—operations that measure the slopes and lengths of scene objects, operations that select arbitrary scene parts, and graphical objects that can be geometrically constrained. Finally, it lists the actual keyboard actions that are used to invoke these operations.

3.1.1 Gravity

During most snap-dragging operations, the user positions a hardware cursor over a drawing area, the *scene*, by moving a pointing device such as a mouse or tablet. The hardware cursor, called the *cursor* henceforth, is under direct user control; it is unaffected by the current scene contents and gravity function. The cursor, in turn, controls a software cursor, called the *caret*. Placing the caret precisely is the key to making drawings with snap-dragging. Figure 3-1 shows the cursor as a circle and the caret as a wedge shape.

During the "Caret Placement" operation, the caret's position is determined both by the

position of the cursor and by the graphical objects that are currently in the scene. When the cursor is far away from all scene objects, the tip of the caret is centered on the cursor (Figure 3-1(a)). However, when the cursor gets near to a scene object, the caret leaves the cursor and snaps its tip onto the scene object (Figure 3-1(b)). The function that computes the caret position and orientation from the cursor position and the scene objects is called *gravity*. The caret changes its orientation to match the local slope of the scene object that it snaps to. Those objects that the caret can snap to are called *gravity-active.*

The "Caret Placement" operation is invoked by holding down a combination of keyboard keys and a mouse button and continues until the mouse button is released. Except where noted, all of the operations discussed in this section are invoked directly from the keyboard and mouse.

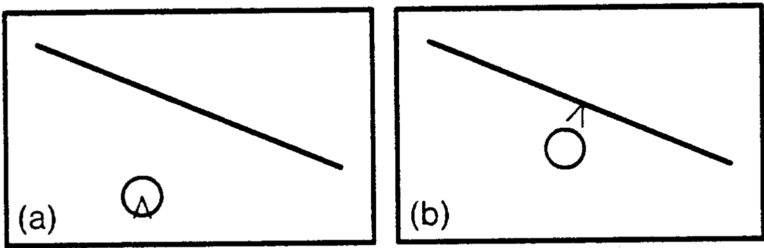


Figure 3-1. During a "Caret Placement" operation. When the cursor gets close enough to a line segment, and gravity is on, the caret snaps onto the line segment precisely.

The gravity function that we have found most useful for snap-dragging is the *points-preferred* mapping. With points-preferred gravity, the caret has a slight preference for line segment endpoints, spline control points, and intersection points over other types of scene objects. For instance, in Figure 3-2, when the cursor is far from the intersection point, the caret snaps to the nearest line (Figure 3-2(a)), but when the cursor is close enough, the caret snaps to the point of intersection even though the cursor is closer to the horizontal line than it is to the intersection point (Figure 3-2(b)).

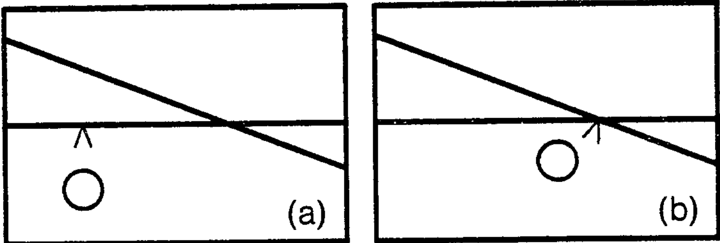


Figure 3-2. Points-preferred gravity. (a) The cursor is far from the intersection point. (b) The cursor is close enough to the intersection point that the caret snaps to it.

Because the caret can be placed precisely, as described above, other operations can be performed precisely by using the caret as a parameter. For instance, when the "Add Line Segment" operation begins, a line segment is added from the previous caret position to the current caret position, which is computed from the current cursor position. As the "Add Line Segment" operation proceeds, the line segment is adjusted in real time using a technique called *rubber-banding* [Newman79], so that one endpoint continues to follow the tip of the caret. Since the caret still obeys the gravity mapping, the caret, and hence the new line segment endpoint, can be snapped to scene objects. In Figure 3-3, one "Caret Placement" operation and one "Add Line Segment" operation are used to construct the diagonal of a square.

Notice that the caret only follows the cursor during specific operations. At other times, as shown in Figure 3-3(b), the caret retains its placement while the cursor moves independently. This independence makes it possible to use the cursor to press menu buttons or point at an application in another window, without disturbing the progress of a geometric construction.

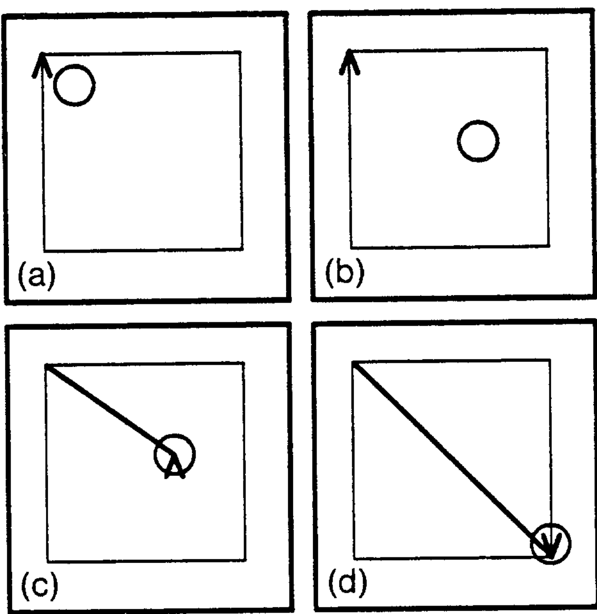


Figure 3-3. Constructing the diagonal of a square. (a) The caret is snapped to one corner using the "Caret Placement" operation. (b) The cursor is moved, leaving the caret behind. (c) The "Add Line Segment" operation begins. (d) The caret is snapped to the opposite corner.

Gravity can be turned on and off. When gravity is off, the caret position simply tracks the cursor position.

3.1.2 Precise Transformations

Snap-dragging composes objects using interactive affine transformations, such as translation, rotation, scaling, and skewing, where the amount by which to translate, rotate, scale or skew is a simple function of the caret position. This section describes how translation, rotation, and scaling depend on the caret position and shows how they are used to compose shapes. Skewing is described in section 3.1.4.

During the translation operation, the selected objects are translated by the vector from the initial caret position to the final caret position. In Figure 3-4, the user translates a selected square until its corner touches the vertex of a hexagon. This construction proceeds in four steps. The "Caret Placement" command is used to put the caret on a corner of the square (Figure 3-4(a)). When the "Caret Placement" command is completed, the cursor moves independently of the caret (Figure 3-4(b)). When the "Translate" command begins, the caret rejoins the cursor,

bringing any selected objects with it (Figure 3-4(c)). Since the caret still obeys the gravity function, the caret can be snapped onto scene objects, achieving a precise translation (Figure 3-4(d)).

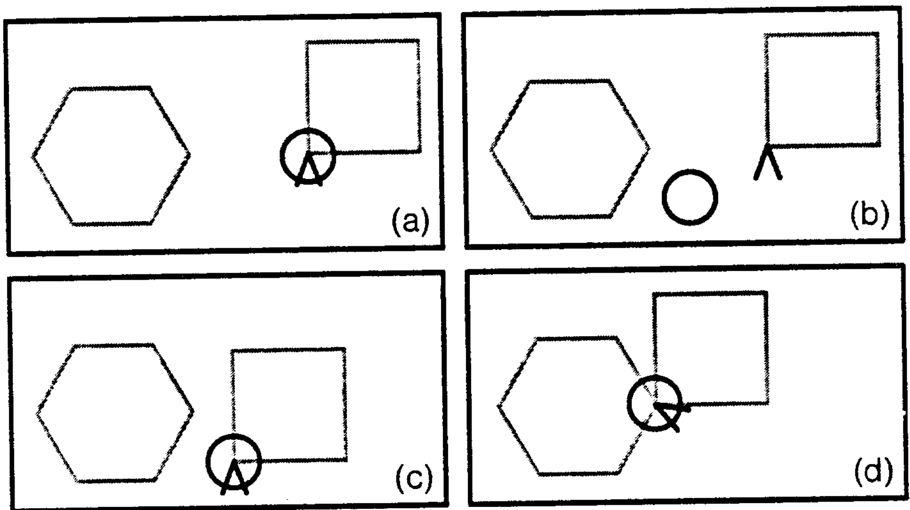


Figure 3-4. (a) The caret is snapped onto the selected square with the "Caret Placement" command. (b) The cursor is moved, leaving the caret behind. (c) When the "Translate" command begins, the caret and square move together. (d) The caret and square snap to the hexagon.

To reduce clutter, the cursor is omitted in the remainder of the figures in this section. The reader should remember that the cursor is always involved in placing the caret.

A distinguished object called the *anchor* is used as a center of rotation, a center of scaling, or a gravity-active position that the user wishes to remember. As shown in Figure 3-5, when the user invokes the "Drop Anchor" command, the anchor takes its position and orientation from the caret, permitting the anchor to be placed precisely. The anchor icon was designed to look somewhat like a nautical anchor. The barbed lines on the right and left of icon indicate the positive and negative x axes of the anchor's coordinate system respectively. The origin of the anchor's coordinate system is at the center of the black-bordered square.

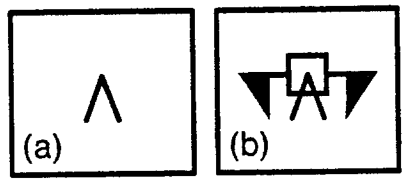


Figure 3-5. The caret is placed (a) and the "Drop Anchor" command is invoked (b).

During the rotation operation, the selected objects are rotated about the anchor point through the angle between the original line determined by the anchor and the caret and the current line determined by the anchor and the caret. Figure 3-6 shows the use of rotation to make the left edge of the square collinear with the upper right edge of the hexagon. In Figure 3-6(a), we drop the anchor at the vertex shared by the hexagon and square. In Figure 3-6(b), we snap the caret to the upper left corner of the square. Throughout the rotation, the caret remains on the line determined by the left edge of the square, as is apparent in Figure 3-6(c). Finally, when we snap the caret to the hexagon's vertex, the hexagon and the square have collinear edges, as shown in Figure 3-6(d).

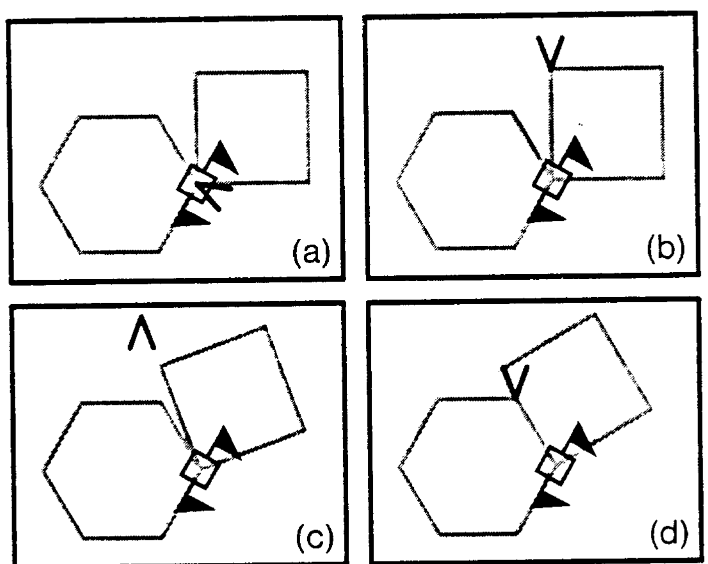


Figure 3-6. (a) The anchor is placed at the caret position. (b) The caret is placed on the upper left corner of the square. (c) During the "Rotate" command the left edge of the square points towards the caret. (d) The caret is snapped to a hexagon vertex.

In a similar fashion, we can scale the square to have sides of the same length as the hexagon's sides. In Figure 3-7(a) we snap the caret to the upper left corner of the square. During the scaling

operation, the selected object (the square) is scaled by the ratio of the current anchor-caret distance to the original anchor-caret distance. Thus, the left edge of the square remains congruent to the anchor-caret distance during scaling. When we snap the caret to the hexagon vertex, the side of the square becomes congruent to the side of the hexagon (Figure 3-7(b)). The "Lift Anchor" command is given and the caret is moved out of sight to produce Figure 3-7(c).

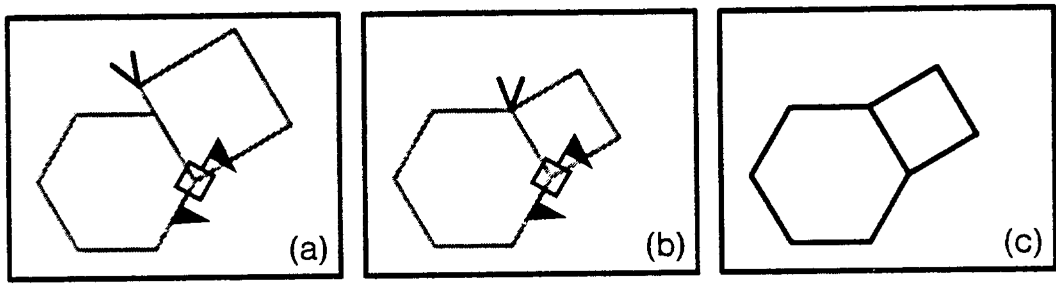


Figure 3-7. (a) The caret is placed on the upper left corner of the square. (b) The "Scale" command is invoked and the caret is snapped to the hexagon. (c) The resulting configuration.

3.1.3 Alignment Objects

To perform a richer set of constructions, we can take advantage of alignment objects—groups of lines and circles used as intermediate steps in a geometric construction, as they are in traditional compass and straightedge constructions. Four types of alignment objects have proven to be useful: lines of known slope, circles of known radius, lines making a known angle to a scene edge, and lines parallel to a scene edge at a known distance from it. These alignment objects are called *slope lines*, *circles*, *angle lines*, and *distance lines*, respectively. Unlike traditional constructions, where alignment objects are drawn one by one, snap-dragging draws many alignment objects in response to a single user action. For example, if the user activates circles of radius 1 inch, circles will be constructed at many sites in the illustration as described below.

The user expresses an interest in a particular slope, radius, angle, or distance by selecting its value in the appropriate menu. Values are selected by pointing at them with the mouse and clicking a button. Selected values appear in bold face. Figure 3-8 shows the four menus as they appear when Gargoyle is first invoked. Slopes are in degrees from horizontal, angles are in degrees, radii and distances are in scale units, where the scale unit can be set to inches, centimeters, or any other value.

Values can be added to these menus and deleted from them. The "Add!" buttons allow the user to select a number from any text region on the screen and add it to the associated menu. "Get!" adds to the menu the slope, length, or angle of a selected scene object. New entries are inserted into the chosen menu in sorted order and can be selected thereafter in the same fashion as the original entries. The "Delete!" buttons delete from the menu all values that are currently selected. If any menu contains more values than can be displayed the values wrap onto additional lines, accessible via a scrollbar.

Slope: Get! Add! Delete! 150 135 120 90 60 45 30 0
Angle: Get! Add! Delete! 90 60 45 30 0 -30 -45 -60 -90
Radius: Get! Add! Delete! 1/18 1/9 1/8 1/4 1/3 1/2 2/3 3/4 1 2 4
LineDistance: Get! Add! Delete! 0 1/18 1/9 1/2 1

Figure 3-8. The default menus of slopes, angles, radii, and parallel line distances.

When the user *activates* a value in the slope menu by selecting it, a line of the specified slope is constructed at each point in the scene that has been designated as a *trigger point.* Both the vertices of objects and the anchor can be trigger points. The anchor is a trigger point by default. Thus, if we drop the anchor (Figure 3-9(a)) and activate 30 degree slope lines (Figure 3-9(b)), a line of slope 30 degrees is constructed through the anchor point. Likewise, if we activate 2/3 inch circles, as shown in Figure 3-9(c), a circle with radius 2/3 of an inch is constructed, centered on the anchor point. In Gargoyle, alignment lines and circles are drawn with a distinctive grey color to help the user distinguish them from scene objects.

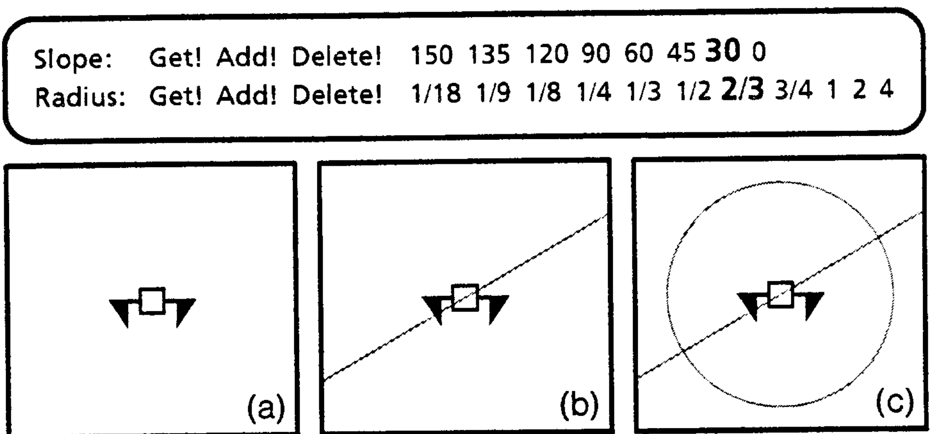


Figure 3-9. (a) The anchor with no alignment objects turned on. (b) 30 degree slope lines are activated. (c) In addition, 2/3 inch circles are activated.

The caret will snap to lines, circles and their intersection points. Hence, we have an easy way to construct a line segment of length 2/3 inches at 30 degrees to horizontal, as shown in Figure 3-10. Notice that the anchor point is one of the points that are preferred by points-preferred gravity, as is the intersection point of the slope line with the alignment circle.

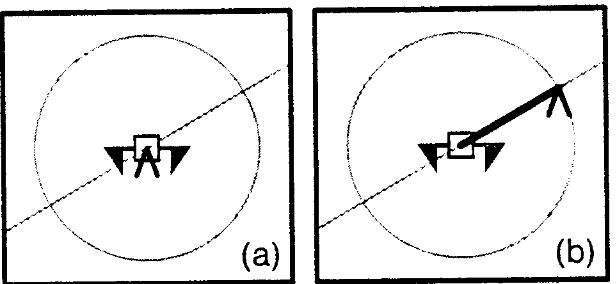


Figure 3-10. (a) The caret is snapped to the anchor point. (b) A line segment is added from the anchor point to the intersection of the slope line and the alignment circle.

The vertices of scene objects can be designated as triggers by making them *hot*. Vertices and edges, when they are both hot and stationary, trigger alignment objects. For instance, we can select the upper vertex of the line segment in Figure 3-10(b) and invoke the "Make Hot" command. A white square is drawn over the vertex to show that it is hot, and the system automatically constructs a new slope line and a new circle through the vertex (the slope line is not actually added because it is redundant), as shown in Figure 3-11(a). By adding a second line

segment from the caret to an intersection point of the two circles (Figure 3-11(b)), and a third segment back to the anchor point, we can complete an equilateral triangle (Figure 3-11(c)).

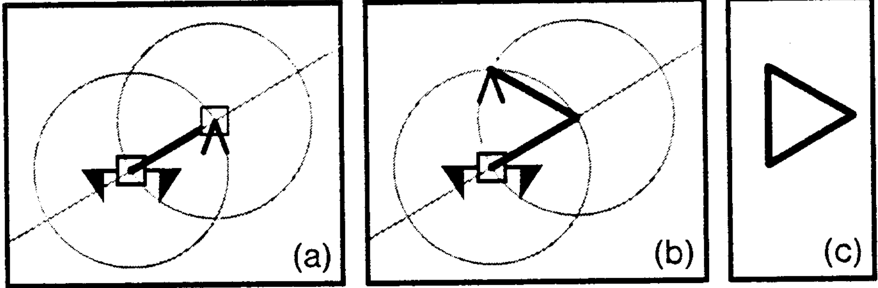


Figure 3-11. (a) The upper vertex of the line segment is made hot. (b) A new segment is added and snapped to an intersection point. (c) Adding a third segment completes an equilateral triangle.

The above construction was a bit tedious. Not only did we add three line segments and select 30 degree slopes and 2/3 inch circles, but we had to drop the anchor and make one of the vertices hot. These last two operations can be avoided by having the system make the vertices hot for us. Gargoyle has a mode, called *automatic*, during which all stationary parts of a shape are made hot automatically while some parts of that shape are moving. This rule captures the intuitive idea that users will often want to align the vertices of an object with other vertices of that same object. Figure 3-12 shows the equilateral triangle constructed again in automatic mode. Assuming that automatic mode was on to begin with, the whole construction takes only six keystrokes: (1) activate circles of radius 2/3 inches, (2) activate lines of slope 30 degrees, (3) place the caret, (4)-(6) add the three line segments. Note that four keystrokes would be needed to rough in an arbitrary triangle. For this example, precision costs only two keystrokes.

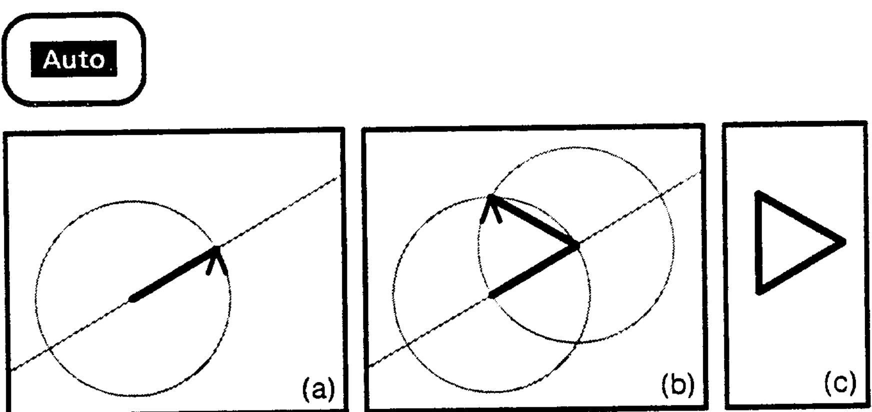


Figure 3-12. A quick equilateral triangle construction in automatic mode.

While slope lines and alignment circles are triggered by hot vertices, distance lines and angle lines are triggered by hot *line segments.* In Figure 3-13(a), all three line segments of a triangle have been made hot, as indicated by the white squares with black borders on the segment endpoints, and the distance "0.2 inches" has been activated. The system automatically constructs two alignment lines next to each hot segment at a distance of 0.2 inches on each side. When the triangle is rotated or translated, the alignment lines rotate and translate with it, as shown in Figure 3-13(b).

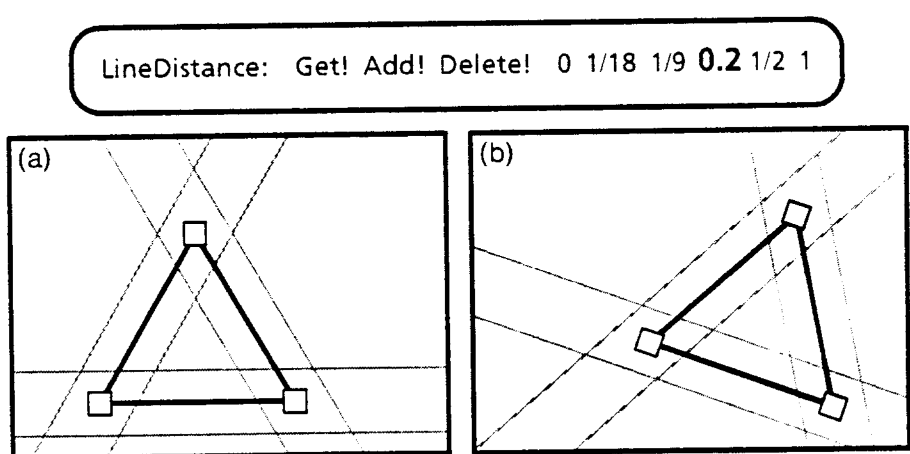


Figure 3-13. Distance lines. (a) All three edges of a triangle are made hot and 0.2 inch distance lines are activated. (b) The triangle is translated and rotated.

One use for distance lines is to create offset shapes. Figure 3-14(a) shows an intermediate step in the process of constructing a triangle offset from the original by 0.2 inches. Figure 3-14(b)

shows the resulting offset triangle. Distance lines were used often in the preparation of this dissertation section to place the figure numbers at a constant offset from the frame borders (e.g., the labels "(a)" and "(b)" in Figure 3-14) and to space the frames at a constant offset from each other (all frames in this section are separated by 1/8 inches).

Note that the white squares that indicate hot segments are not shown in Figure 3-14. During operations that move the caret, such as "Add Line Segment" and all interactive transformations, selection and hotness feedback are suppressed to make it easier for the user to identify scene objects and snap the caret to them. The figures in this dissertation adhere to this convention, except where noted.

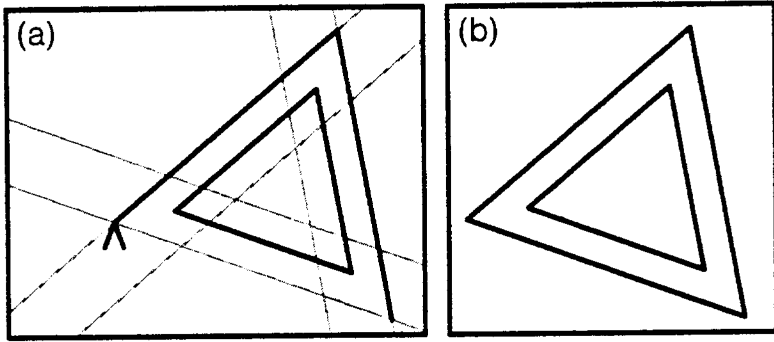


Figure 3-14. (a) Using distance lines to create an offset triangle. (b) The finished product.

When an angle is activated, the system constructs two alignment lines, one at each end of the line segment. The lines make the specified angle with the line segment, where angles are measured counter-clockwise from the segment about the endpoint in question. Figure 3-15 illustrates 45 degree angle lines triggered by a single hot segment.

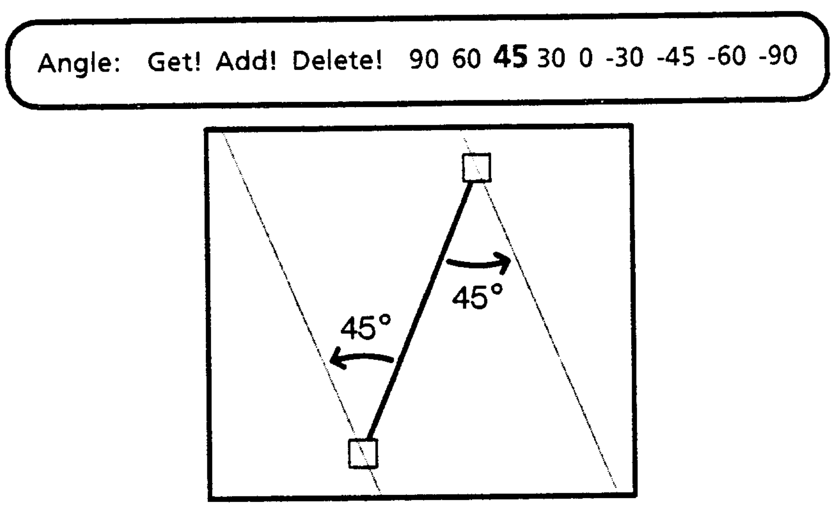


Figure 3-15. A hot segment with 45 degree angle lines activated.

One use of angle lines is to construct a shape with known angles at an arbitrary orientation. In Figure 3-16, the user has turned on automatic mode and activated 90 degree angle lines. He begins to draw the letter F. No angle lines appear as he draws the first segment because there are no stationary segments to act as triggers (Figure 3-16(a)). However, when the second segment is added in Figure 3-16(b), the first segment is now stationary and triggers two angle lines. Continuing in this fashion, the user completes an F (Figure 3-16(c) and (d)).

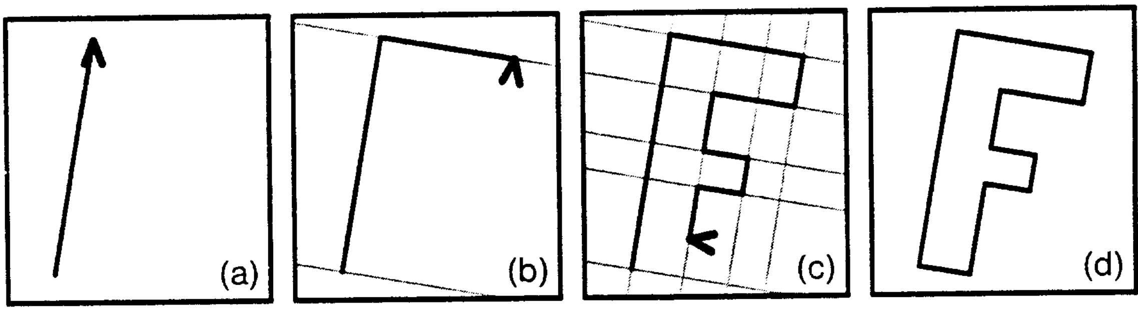


Figure 3-16. Angle lines of 90 degrees. (a) Adding a line segment. (b) Adding a second segment. (c) Adding more segments. (d) The resulting F shape.

3.1.4 Advanced Snap-Dragging Operations

Previous sections illustrated the fundamental principles of snap-dragging: placing the caret with gravity, using affine transformations parameterized by the caret, and activating classes of

alignment objects that augment the set of gravity-active objects. The examples in this section show some of the machinery that goes hand in hand with snap-dragging. In particular, they illustrate the use of multiple slope values to produce a skewed grid, the selection of arbitrary collections of object parts, the skewing transformation, the measurement of slopes and distances, gravity-active midpoints, the extension of alignment menus, constrained objects, curve editing operations, and the lines-preferred gravity function.

Snap-dragging works particularly well in pictures where the same precise relationships hold between a large number of neighboring parts. In Figure 3-17, with 0 and 60 degree slope lines and the automatic rule on, we can sketch in a slanted letter G as fast as we can specify its vertices; there is no further need to worry about precision.

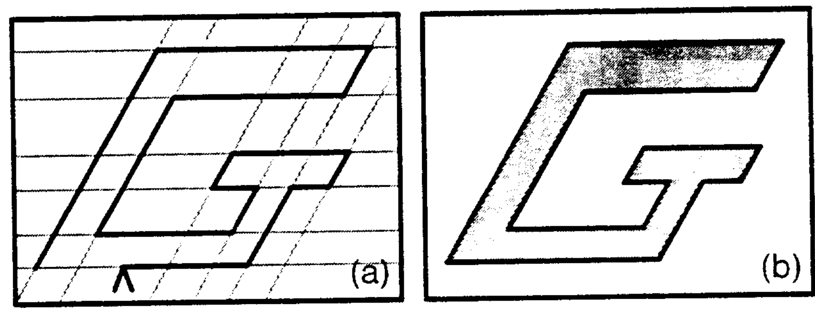


Figure 3-17. (a) Adding the last line segment. (b) The resulting slanted letter G.

The snap-dragging approach benefits greatly from a fine-grained selection mechanism. In Gargoyle, arbitrary collections of vertices and edges can be selected. In this section, selected vertices are shown by drawing a black square over them and selected edges by making them thicker. In Figure 3-18(a), three edges of our slanted G have been selected. If we translate the selected edges, and snap the caret to the slope lines generated, due to the automatic rule, from the non-moving vertices, we can make the G taller as shown in Figure 3-18(b). By selecting other combinations of edges, we can change other properties of the G, including its width, or the width of its strokes, all the time keeping its edges at 0 and 60 degrees.

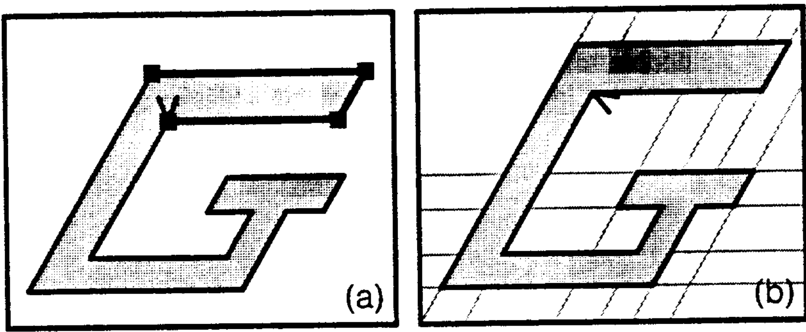


Figure 3-18. Making the G taller. (a) Three selected segments and four selected joints are highlighted. (b) Translating the selected parts.

If we decide that we don't like the way the G looks at a 60 degree slant, we can use interactive skewing to improve it. In Figure 3-19(a), the G has been selected, the anchor has been placed on the lower left corner of the G, and a line segment, *l*, has been added coincident with the top of the letter G. The horizontal arms of the anchor indicate the line of invariance of the skew operation. The line segment *l* will help us keep the G at the same height during skewing. In Figure 3-19(b), the "Skew" operation is in progress.

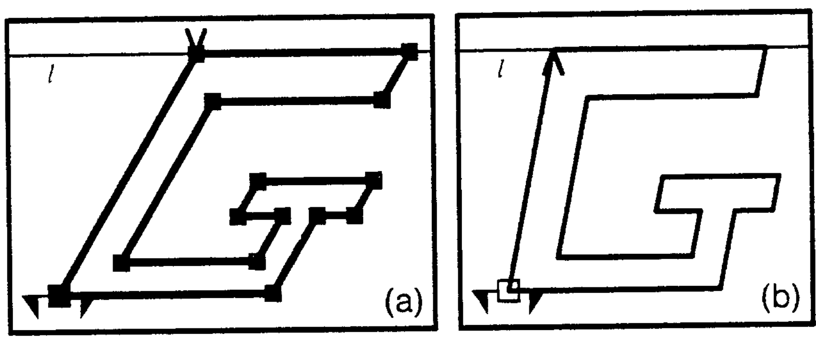


Figure 3-19. (a) The G is selected, the anchor is placed and a guideline *l* is added in preparation for skewing. (b) The G is skewed interactively.

Gargoyle provides a way to measure slopes, distances, and angles and to add these measured values to the alignment object menus. Like most snap-dragging operations, the measuring operations use the caret position as a parameter. Actually, there is no measuring operation *per se*; at the end of each "Caret Placement" operation or interactive transformation, Gargoyle compares the new final caret position with the final caret positions from the previous two "Caret

Placement" or transformation operations. It uses the last two positions to measure slope and radius, and the last three positions to measure an angle and the distance of a point from a line. The measured values are displayed in four control panel regions that are constantly updated, called "SlopeValue", "AngleValue", "RadiusValue", and "LineDistanceValue".

We can edit the G at its new slope by augmenting the slope menu. In Figure 3-20(a) we move the caret to the base of the leftmost line segment of the G. Since the skew operation ended with the caret at the top of this same segment (in Figure 3-19(b)), the value in the "SlopeValue" region is the new slope of the G, 78.84225 degrees. We click the "Add!" button of the slope line menu to add this value to the menu and activate it. In Figure 3-20(b) we use the 78.84225 degree slope lines to lower the lip of the G. Figure 3-20(b) is another example of fine-grained selection – five consecutive segments of the G are being translated down.

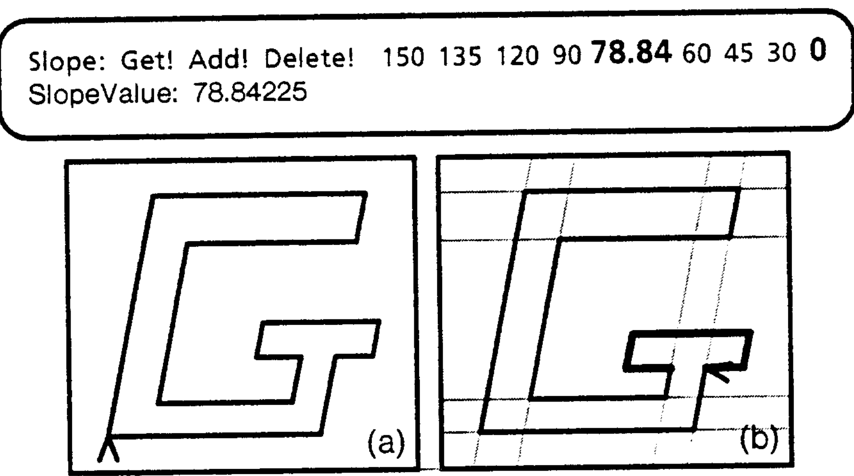


Figure 3-20. (a) Measuring a line segment, (b) Editing the G at its new slope. The five selected segments are shown thicker. (This highlighting would not appear on the screen.)

As a final example of fine-grained selection, consider the circle and four line segments in Figure 3-21(a). If we select the circle (shown by highlighting the five circle control points) and drag it, we get the result in Figure 3-21(b). However, if we select the endpoints of the four surrounding segments as well, as shown in Figure 3-21(c), the "Translate" operation will rubber-band the four line segments and drag the circle as a whole, as shown in Figure 3-21(d). This sort of operation is useful for rearranging box-and-pointer or network diagrams. Notice that there are no constraints on the picture to keep the line segments attached to the circle; the possibilities in Figure 3-21(b) or Figure 3-21(d) are both available to the user with comparable ease.

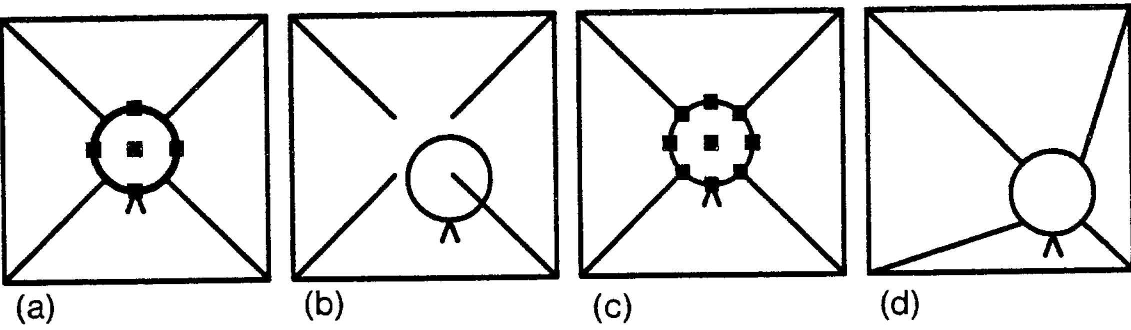


Figure 3-21. A circle is selected (a) and dragged (b). A circle and four line segment endpoints are selected (c) and the collection is dragged (d).

When the midpoints mode is on in Gargoyle, all of the midpoints of line segments become preferred points. In other words, if the midpoint is close enough to the cursor, points-preferred gravity will prefer a midpoint to other points on a line segment. In Figure 3-22, the user activates midpoints mode to help him construct a rectangle whose lengths form the golden ratio. Given square ABCD, he measures from the midpoint M of AB to C (Figure 3-22(a) and Figure 3-22(b)). He adds the measured length, 1.12579 inches, to the menu of alignment circle radii and activates that value. With the anchor on M, he activates lines of slope 0 degrees and completes the construction by dragging the right edge of the rectangle to the new intersection point.

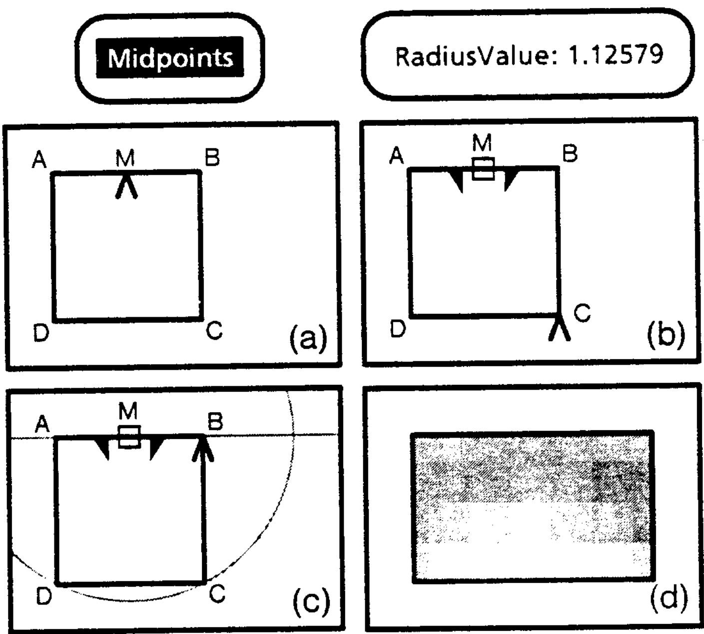


Figure 3-22. The golden ratio rectangle. (a) Find the midpoint of AB, call it M. (b) Place the anchor at M and measure MC. (c) Activate circles of length MC and lines of slope 0. (d) Translate edge BC to the circle/slope line intersection point.

A mathematician would probably find it odd to have the measurements reported in inches. It would be much more helpful to have them in units of the side of square ABCD. Gargoyle will allow the units to be reset to a measured radius. If we measure side AB and set the unit to this value, the measured value from M to C would be 1.1180, which is $\sqrt{5}/2$ to 5 significant figures (the golden ratio = $(\sqrt{5}+1)/2$ = 1.6180...).

One primitive graphical object class implemented in Gargoyle is the Box class. Instances of this class are constrained to be rectangles (unless skewed, whereupon they remain parallelograms). Figure 3-23 contrasts the dragging behavior of a regular (unconstrained) quadrilateral with the dragging behavior of a constrained rectangle when a single control point is selected. Both quadrilaterals are snapped onto a rectangle that is shaded grey. Figure 3-24 shows the different behaviors when an edge is selected. Notice that for the constrained rectangle, selecting its edge and snapping the caret onto the shaded rectangle during a Drag operation leaves the left edges of the two rectangles collinear. This construction is a quick way to get the left edges of boxes to line up without disturbing the right edges.

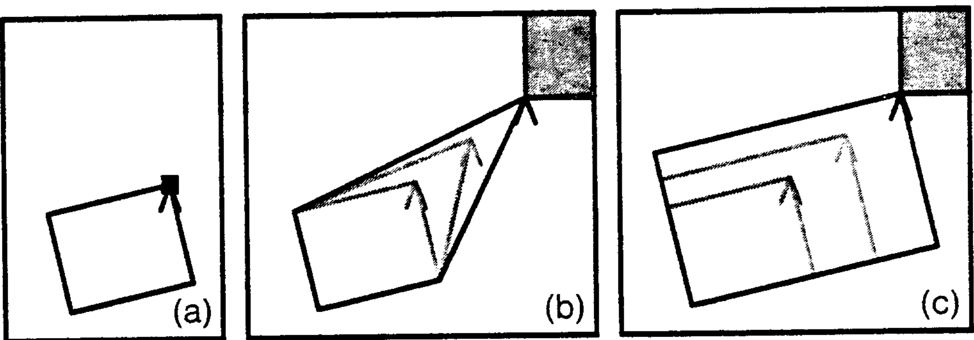


Figure 3-23. Translating the upper right corner of a rectangle. (a) The corner is selected. (b) An unconstrained quadrilateral. Intermediate shapes are shown in grey. (c) A constrained rectangle.

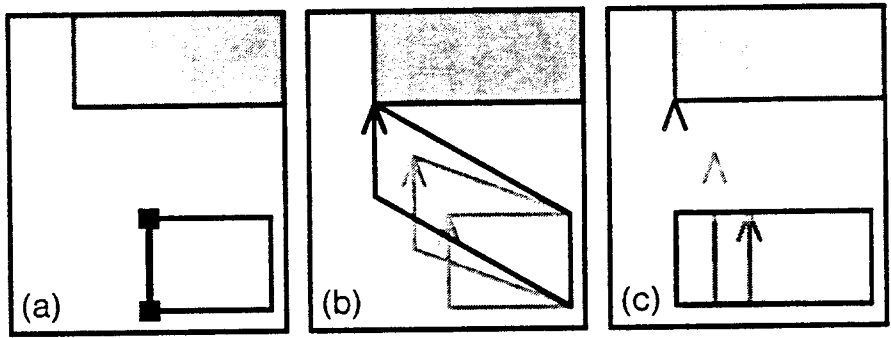


Figure 3-24. Translating the left edge of a rectangle. (a) The edge is selected. (b) An unconstrained quadrilateral. (c) A constrained rectangle.

Snap-dragging can be used to edit any curve that is parameterized by control points. It works particularly well with Bězier curves [Faux79] since precise placement of Bězier control points leads to precise determination of the tangent direction at the ends of each parametric cubic piece. In Figure 3-25, an artist has decided to use a Bězier spline as an arc from a box to a circle. He has also decided that the arc should begin vertically down from the box and should end, at the circle, at a 45 degree slope. To achieve these constraints, the artist places control point 2 on a 90 degree slope line and control point 3 on a 45 degree slope line, *l*, as shown. The figure shows the spline curve for two different positions of control point 3 on *l*.

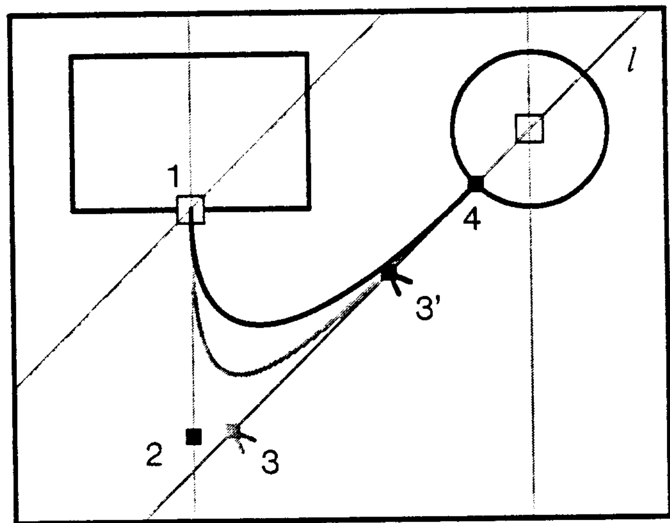


Figure 3-25. Editing a Bézier spline with snap-dragging.

Because the orientation of the caret changes to match the local slope of the curve that it is snapping to, we can perform constructions where that local slope is important. For instance, to construct the line tangent to a curve at a point **p**, we can snap the caret to the curve at **p**, as shown in Figure 3-26(a), and drop the anchor. Because the anchor takes its orientation from the caret, the x axis of the anchor, shown barbed, follows the curve tangent at **p**. The x axis of the anchor is treated by Gargoyle as a special hot edge. So, if we activate the value 0 from the distance menu (or the value 0 from the angle menu) a line is constructed that is parallel to the anchor's x axis and passes through the anchor point (Figure 3-26(b)). The user can now construct a line segment tangent to the curve by snapping both endpoints of the segment to the alignment line, as shown in Figure 3-26(c).

While it might seem preferable to provide a special command to construct tangents, very little would be gained. In particular, if there are many tangents to construct, each new tangent segment takes only four keystrokes to specify – one to place the caret, one to drop the anchor, and two to place the endpoints of the new line segment. To set up this sequence of constructions, the user would activate distance line value 0 at the beginning of the sequence and de-activate it at the end.

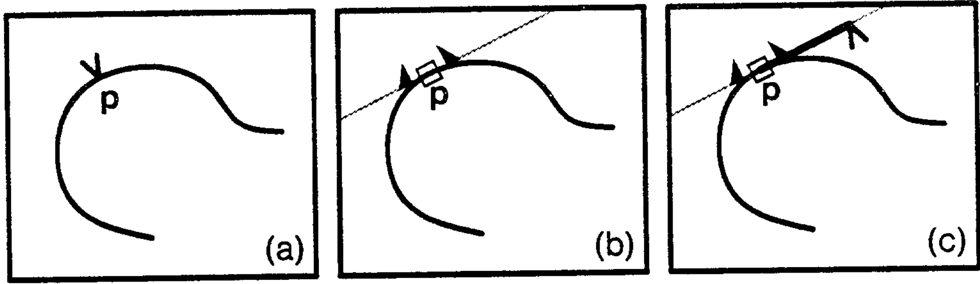


Figure 3-26. Constructing a line tangent to a curve at a point, p. (a) Placing the caret at p. (b) The anchor triggers a tangent alignment line. (c) Adding a tangent line segment.

In addition to the points-preferred gravity used in all of the examples above, it is useful to have a second kind of gravity, lines-preferred gravity, that snaps to the nearest point or curve in the scene giving no special consideration to points. Figure 3-27 illustrates a situation where lines-preferred gravity comes in handy. The user wishes to position a piece of text on top of the horizontal alignment line and near the edge of a box, but wishes to judge by eye how it should be positioned left to right. Using points-preferred gravity, he begins dragging the text towards the corner of the box, and the caret snaps to the intersection point of the alignment line with the box, as shown in Figure 3-27(b). In order to get close to this point without snapping to it, the user switches to lines-preferred gravity as shown in Figure 3-27(c). The keyboard commands for switching gravity type are arranged so that this can be done without interrupting the interactive translation that is in progress.

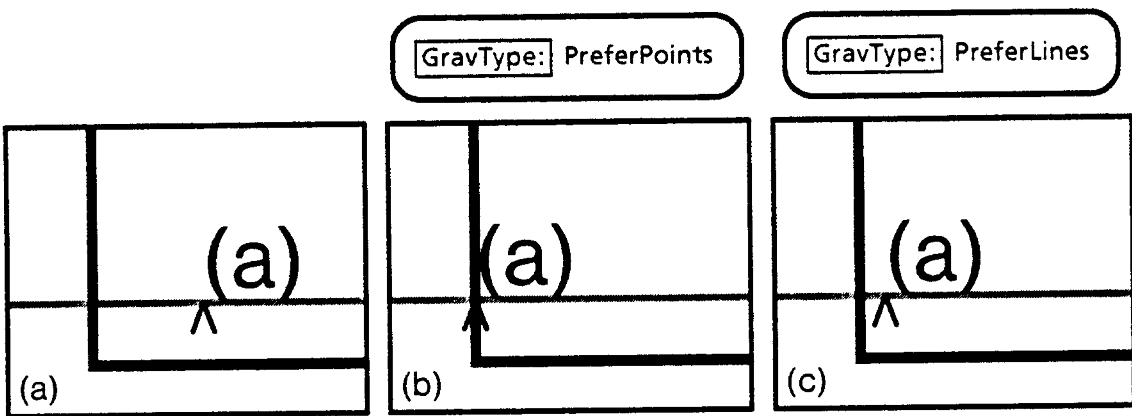


Figure 3-27. Lines-preferred gravity. (a) Dragging a text string along a horizontal line near a black box. (b) The caret snaps to the intersection point of the line and the box edge. (c) With lines-preferred gravity, the caret can get close to the intersection without snapping.

3.1.5 User Interface Summary

The operations described in section 3.1 can be invoked by pressing combinations of keys on the keyboard and mouse or by using extensible menus. These operations are summarized in this section.

The Dorado personal computer has a mouse with three buttons, call them Left, Middle, and Right. The keyboard includes the special keys "Ctrl" and "Shift". Table 3-1 shows the 19 combinations of mouse buttons and special keys used to invoke those snap-dragging operations that are performed at a particular cursor position. In the first four rows, commands are invoked by clicking the mouse button once. In the second four rows, the mouse button is clicked twice in quick succession. Blank entries in the table indicate combinations of buttons that currently have no function.

Single	Left	Middle	Right
Plain:	Select Vertex	Select Segment	Extend Selection
Shift:	Caret Placement	Add Line Segment	Add Bézier
Ctrl:	Translate	Rotate	Scale
Ctrl-Shift:	Deselect Vertex	Deselect Segment	
Double	**Left**	**Middle**	**Right**
Plain:	Select Cluster	Select Region	Select Objects in Box
Shift:			Add Constrained Rectangle
Ctrl:	Copy & Translate		Skew
Ctrl-Shift:	Deselect Cluster	Deselect Region	

Table 3-1. Key and mouse button combinations for snap-dragging operations that are performed at a particular cursor position.

Table 3-2 shows the 12 combinations of alphabetic keys along with "Ctrl" and "Shift" that are used to invoke those snap-dragging commands that are independent of the current cursor position. Some of these commands operate while nothing is moving. For instance, "Drop Anchor" places the anchor at the current, stationary, caret position. Other commands can be used while the caret is moving. For instance, "Gravity On/Off" can be used to toggle gravity on and off in the middle of an interactive transformation, during caret placement, or while a line segment or Bézier spline is being added. Likewise, "Forward to next Gravity Type" switches back and forth between points-preferred gravity and lines-preferred gravity, during these operations. The underlined letters in Table 3-2 indicate a mnemonic to help users remember the command.

Keyboard	Ctrl	Ctrl-Shift
A:	Drop Anchor	Lift Anchor
C:	Cycle Selection Forward	Cycle Selection Backward
F:	Forward to Next Gravity Type	
Q:	Quit Making Alignments	Quit Having Active Alignments
S:	Make Hot (Steam)	Make Cold
Double S:	Make All Hot (Steam Scene)	Make All Cold
Space Bar:	Gravity On/Off	

Table 3-2. Keyboard operations that are performed when the mouse is stationary.

Several commands in Table 3-2 have not been mentioned previously. The command "Quit Making Alignments" temporarily prevents Gargoyle from generating alignment objects, but

leaves the alignment menus as they are. "Quit Having Active Alignments" turns off all of the alignment values that are active in any of the alignment menus. "Make All Hot" and "Make All Cold" make all of the objects in the scene hot or cold respectively. The "Cycle Selection" operations select, in turn, each of the objects that are near the cursor. They are discussed further in section 3.4.

Users control the rest of the snap-dragging parameters from extensible menus and buttons. The four alignment menus were shown above in Figure 3-8. Five other buttons exist:

1) *Gravity Extent.* This button controls how far the cursor must be from an object before the caret will snap to it. Clicking this button with the Left mouse button, halves the value of the gravity extent, clicking with the Right button doubles the extent, clicking with the Middle button resets the extent to its default value.

2) *Gravity.* Clicking this button turns gravity off. Clicking again turns gravity on.

3) *Midpoints.* Clicking this button turns midpoints mode on and off.

4) *Auto.* Clicking this button turns the "automatic rule" on and off.

Altogether there are 23 snap-dragging commands, not counting selection operations, plus the four alignment menus, each with 3 buttons to add and delete values from the menus.

Of the 23 snap-dragging commands, some would be found in other systems in one form or another. For instance, many systems have commands for interactive translation, rotation, scaling, and skewing, for adding a line segment, Bêzier spline, or rectangle, for copying, or for placing a center of rotation (one of the uses of "Drop Anchor"). In general then, 9 of these 23 commands would be provided whether or not snap-dragging was in use. The remaining 14 commands ("Caret Placement", "Lift Anchor", "Forward to Next Gravity Type", "Quit Making Alignments", "Quit Having Active Alignments", "Make Hot" , "Make Cold", "Make All Hot", "Make All Cold", "Gravity On/Off" , "Gravity Extent", "Gravity", "Midpoints", and "Auto") and the alignment menus represent the user interface complexity that is present solely to support snap-dragging. Some of the 14 commands are for convenience only. Users need not learn all of them to be proficient.

3.2 Snap-Dragging In Three Dimensions

Snap-dragging in three dimensions is similar to its two-dimensional variant. It involves the interaction of gravity, interactive transformations, and alignment objects. It works with a two-dimensional pointing device in a single perspective view.

Gravity and alignment objects are even more important in three dimensions than in two. In two dimensions, it is possible to rough in a shape with gravity turned off. In three dimensions, snapping to scene objects and alignment objects becomes the only convenient way to place three-dimensional points with a two-dimensional pointing device.

In this section I will introduce three-dimensional snap-dragging with a set of examples, showing this technique as it appears to the user. A summary of the user interface appears at the end.

3.2.1 Gravity

In three dimensions, gravity snaps a software cursor, called the *skitter* (to distinguish it from the *caret* in two dimensions), to scene objects and alignment objects. A two-dimensional pointing device, a mouse, is used to place the skitter, in a single perspective view of the scene. While this approach does not permit the skitter to be moved along an arbitrary path in 3-space, the skitter can be moved on the surfaces, curves, and points of scene objects and alignment objects, as described below.

As in two dimensions, the mouse controls a hardware cursor, the *cursor*, that moves in the plane of the screen. A ray is shot from the eyepoint (center of projection), through the cursor. If this ray passes near a scene object, the skitter snaps to that object. If there are no scene objects near the ray, the skitter snaps to a plane, the *default plane*, that is parallel to the screen plane and passes through the origin of the reference coordinate frame, WORLD. In Figure 3-28, a scene consisting of a single cube is being edited. The default plane is shown for this particular eyepoint and viewing direction. This scheme is similar to the "cylinder mode" of Sketchpad III [Johnson63a, Johnson63b].

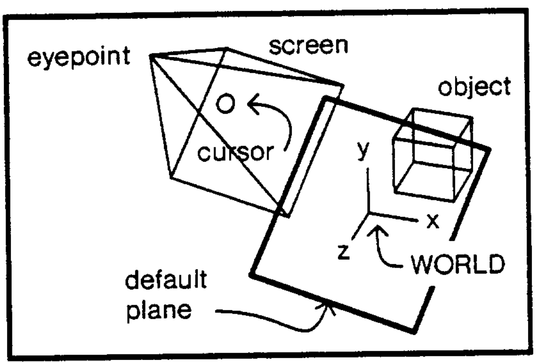


Figure 3-28. The default plane is parallel to the screen and passes through the origin of WORLD.

The three-dimensional gravity function is similar to the function used in two-dimensional snap-dragging. In both mappings, the software cursor is placed on an object that appears behind the hardware cursor. The difference is that in Gargoyle, the software cursor is moving on objects with discrete depths in a 2 1/2-dimensional ordering, while in Gargoyle3D, the depths are continuous.

Figure 3-29 shows skitter placement as seen by the user. The skitter is drawn as a set of three mutually perpendicular axes; the x and y axes are labelled with text and the z axis is drawn as a triangle. The skitter is shown moving on the default plane in Figure 3-29(a) and snapping to the edge of a block in Figure 3-29(b). In addition to snapping the origin of the skitter to the nearest object point, the gravity routines also orient the skitter to match the surface orientation at the snapping point; the z axis of the skitter is chosen to be perpendicular to the line or surface, the x axis of the skitter is chosen to be parallel to the curve's tangent direction if the surface point is on a line or curve and to be horizontal otherwise, and the y axis is computed from the cross product of x and z. For instance, in Figure 3-29(a), the z axis is perpendicular to the default plane, while the x axis is horizontal; in Figure 3-29(b), the z axis is perpendicular to one of the faces that determines the edge, while the x axis follows the edge tangent. The "RotateX" transformation, described below, indirectly uses the x axis of the skitter as an axis of rotation.

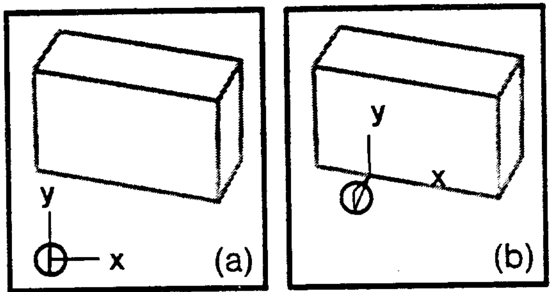


Figure 3-29. (a) The skitter on the default plane. (b) The skitter snapping to a block's edge.

In three dimensions, the points-preferred gravity function performs a three-way arbitration. The skitter can snap to a surface, a curve, or a point. Points-preferred gravity is illustrated in Figure 3-30. If the cursor is near to a vertex or intersection point, the skitter snaps to that point (Figure 3-30(a)). Otherwise, if the cursor is near to an edge, the skitter snaps to that edge, as shown in Figure 3-30(b) and Figure 3-29(b). Otherwise, if the cursor is over a face, the skitter snaps onto that face Figure 3-30(c). Two other gravity functions are used: lines-preferred and faces-preferred. Lines-preferred gravity is like points-preferred gravity, but gives no special treatment to vertices or intersection points—it will snap to them only if they are nearer to the cursor ray than any other point on an edge or alignment line. Similarly, faces-preferred gravity gives no special treatment to edges—it will snap to them only if they are nearer to the cursor ray than any other point on a scene object or alignment object.

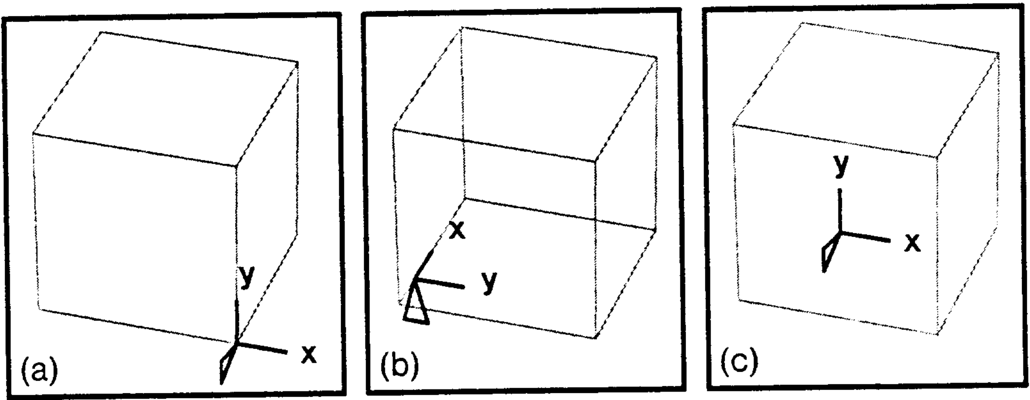


Figure 3-30. Snapping the skitter (a) to a vertex, (b) to a (back-facing) edge, and (c) to a face.

As in two dimensions, the "Add Line Segment" operation adds a line from an initial software cursor position to a final software cursor position, rubber-banding the segment while the operation is in progress. In Figure 3-31(a), the skitter is placed on the corner of a cube. The

cursor can then be moved independently of the skitter (Figure 3-31(b)). When the "Add Line Segment" operation begins, a segment is added from the original skitter position to the new skitter position. In Figure 3-31(c), the line goes from the near lower left corner of the cube to a point on the top face. In Figure 3-31(d), the new line is snapped to the opposite corner of the cube. It is interesting to note that although the line endpoint is moving in a non-trivial manner through three dimensions, its projection moves quite smoothly. In fact, the projection moves just as it would in two-dimensional snap-dragging if we added a line segment in a scene with 12 line segments positioned where the 12 cube edges project onto our viewing plane.

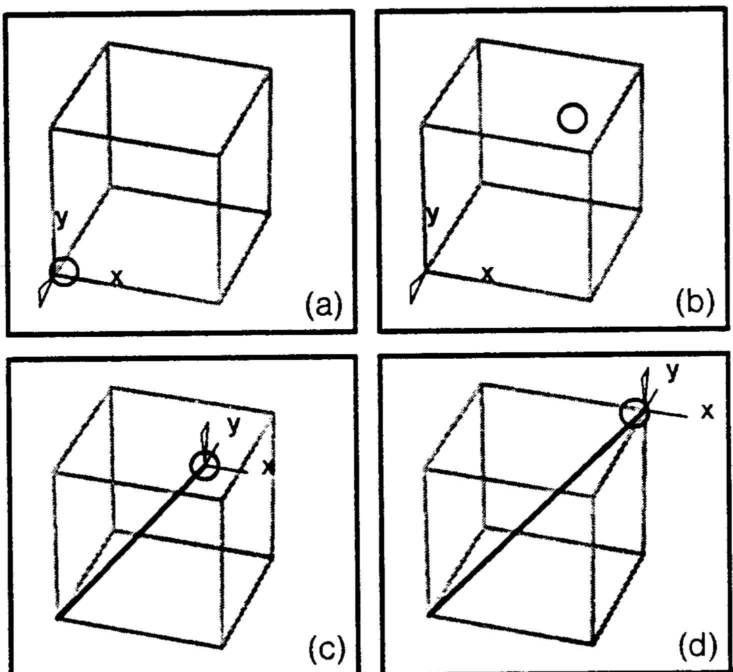


Figure 3-31. Constructing the diagonal of a cube. (a) The skitter is snapped to one corner using the "Skitter Placement" operation. (b) The cursor is moved, leaving the skitter behind. (c) The "Add Line Segment" operation begins. (d) The skitter is snapped to the opposite corner.

3.2.2 Precise Transformations

In three dimensions, translation, rotation, and scaling are parameterized by the skitter position. This section describes, for each of these operations, how the transformation applied depends on the skitter position and shows how to use each operation to compose shapes.

During the translation operation, the selected objects are translated by the vector from the

initial skitter position to the final skitter position. In Figure 3-32, a selected tetrahedron is translated until its corner snaps onto the corner of an octahedron. First, the skitter is placed on the corner of the tetrahedron with the "Skitter Placement" command (Figure 3-32(a)). When the "Skitter Placement" command is completed, the cursor moves independently of the skitter (Figure 3-32(b)). During the "Translate" operation, the skitter tracks the cursor, bringing any selected objects with it (Figure 3-32(c)). Since the skitter still obeys the gravity function, the skitter can be snapped onto scene objects, achieving a precise translation (Figure 3-32(d)).

Note that in Figure 3-32(c) there are no objects near enough to the cursor ray to snap to; the skitter is on the default plane. When the scene is viewed in perspective, the projection of the tetrahedron changes size as the tetrahedron moves from its original depth to the default plane, and again when it moves from the default plane onto the octahedron. If the default plane is roughly the same distance from the viewer as the scene objects, this effect is small and dragging appears smooth. Sometimes it is useful to place the skitter on the default plane during an interactive transformation. For instance, if a translation begins and ends with the skitter on the default plane, the selected objects will be moved parallel to the screen.

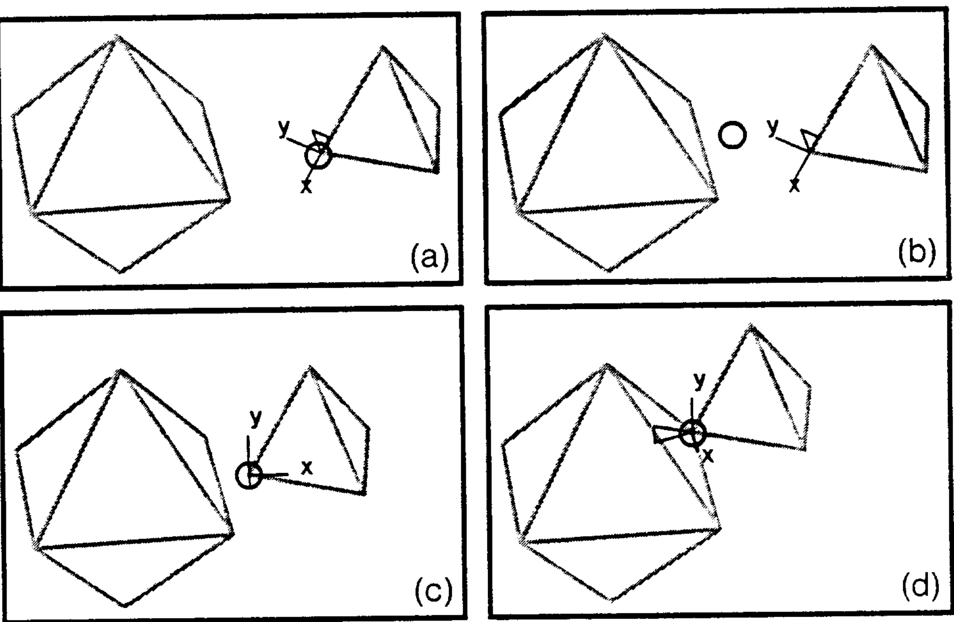


Figure 3-32. Translation. (a) The skitter is snapped onto a selected tetrahedron with the "Skitter Placement" command. (b) The cursor is moved, leaving the skitter behind. (c) The "Translate" command begins. (d) The skitter and tetrahedron snap to the octahedron.

To reduce clutter, the cursor is omitted in the remainder of the figures. The reader should remember that the cursor is always involved in placing the skitter.

A distinguished object called the *anchor* is used as a center of rotation, an axis of rotation, a center of scaling, or a gravity-active position that the user wishes to remember. When the user invokes the "Drop Anchor" command, the anchor takes its position and orientation from the skitter, permitting the anchor to be placed precisely (Figure 3-33). The square in the middle of the anchor faces in the z direction, the barbed line segments lie along the x direction (with an extra arrowhead showing the positive x direction) and the undecorated line segments lie along the y direction.

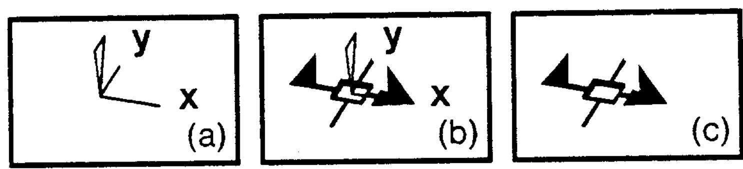


Figure 3-33. Placing the anchor in three dimensions. (a) The skitter is placed. (b) The anchor is created with the same position and orientation as the skitter. (c) The skitter is removed.

During rotation about a point (the "Rotate" command), the selected objects are rotated about the anchor point. This rotation occurs through the angle between the original line determined by the anchor and the skitter, and the final line determined by the anchor and the skitter. The axis of rotation is the line that passes through the anchor point and is perpendicular to the plane determined by three points—the original skitter, the anchor, and the final skitter. Hence, the axis of rotation changes as the skitter moves. This rotation operation is ideal for rotating two edges to be coincident.

Figure 3-34 shows the "Rotate" operation being used to rotate the tetrahedron of Figure 3-32 until it shares an edge with the octahedron. The steps are the same ones used in Figure 3-6 to rotate a square into contact with a hexagon. The anchor is placed at the shared vertex (Figure 3-34(a)) and the skitter is placed on the other end of the edge that we want to align (Figure 3-34(b)). Throughout the rotation, the skitter remains on the line determined by the tetrahedron edge (Figure 3-34(c)), so when we snap the skitter onto the octahedron edge, the two edges are coincident (Figure 3-34(d)).

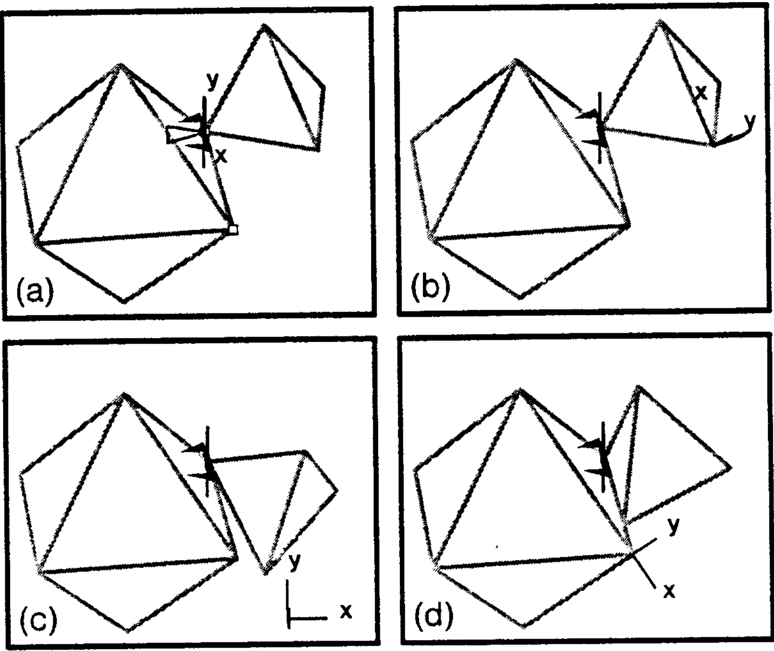
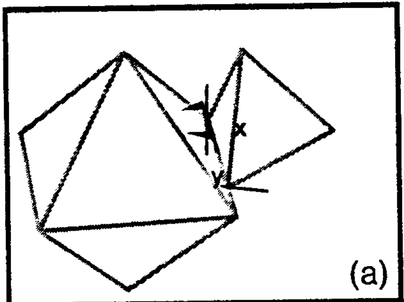


Figure 3-34. Interactive rotation about a point. (a) The anchor is dropped at the skitter position. (b) The skitter is placed at the end of a tetrahedron edge. (c) The "Rotate" operation begins. The skitter is on the default plane. (d) The skitter is snapped to an octahedron edge.

We can scale the tetrahedron so that its edge is congruent with the octahedron's edge. Scaling follows the same steps as in two dimensions. The skitter is placed at the end of the edge of the tetrahedron (Figure 3-35(a)). The "Scale" operation scales the tetrahedron by the ratio of the current skitter-to-anchor distance to the original skitter-to-anchor distance. When we snap the skitter to the end of the octahedron's edge (Figure 3-35(b)), this ratio becomes the ratio of the octahedron's edge to the original tetrahedron's edge, so the two edges are now congruent.



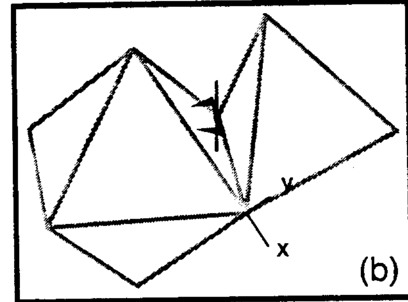


Figure 3-35. Scaling. (a) The skitter is placed on the nearest vertex of a selected tetrahedron. (b) The tetrahedron is scaled until the skitter snaps to a vertex of the octahedron.

During rotation about an axis, the selected objects are rotated about the x axis of the anchor by the angle through which the skitter moves about this axis; any motion of the skitter parallel to the axis is ignored. In Figure 3-36, we rotate the tetrahedron until it shares a face with the octahedron. In Figure 3-36(a), we place the skitter on the top vertex of the tetrahedron. The anchor already has its x axis aligned with the shared edge; this is one of the benefits of having the gravity function align the x axis of the skitter with edges and of having the anchor take its orientation from the skitter. Throughout the "RotateX" operation, the left face of the tetrahedron remains coplanar with the skitter (Figure 3-36(b)), so when the skitter is snapped onto the top vertex of the octahedron the two faces become coplanar (Figure 3-36(c)). The resulting scene is shown in Figure 3-36(d). Figure 3-36(b), (c), and (d) are shown in wireframe for clarity.

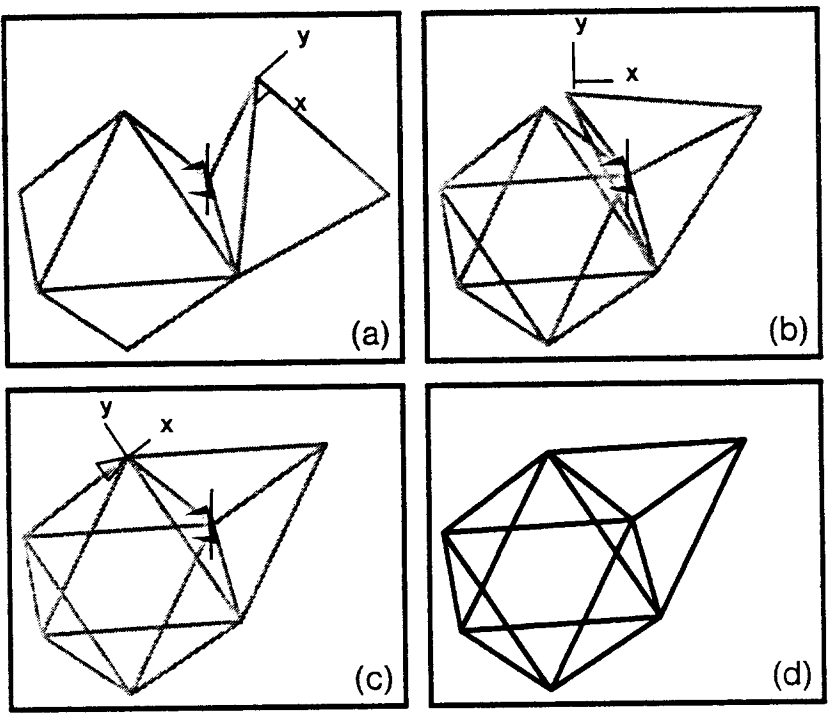


Figure 3-36. Rotating about an axis. (a) The anchor is placed with its *x* axis on the common edge. The skitter is placed on the tip of the tetrahedron. (b) The "RotateX" operation begins. (c) The skitter is snapped to the top vertex of the octahedron. (d) The resulting scene.

3.2.3 Alignment Objects

Each of our two-dimensional alignment types suggest one or more three-dimensional alignment types. In place of slope lines, we will want lines of known direction and planes of known surface normal. In place of alignment circles, we will want alignment spheres. The spheres will produce circles when intersected with planes or with other spheres, so we need not provide a separate mechanism for circles. In place of distance lines, we will want cylinders (the set of all lines at a known distance from a line) and distance planes (the two planes of known distance from a given plane). In place of angle lines, we will want cones (the set of all lines making a given angle with a line at a given point, and angle planes (a plane constructed on each edge of an oriented polygon making a known angle with the plane of the polygon). This would give us seven alignment types in three dimensions. Three of these alignment types have been implemented: lines of known direction, planes of known normal, and spheres of known radius. Even this small set is adequate for many constructions. The alignment lines, planes, and spheres are constructed at the

anchor and at all vertices that the user has declared to be hot.

The direction vectors of alignment lines and the surface normals of alignment planes can both be specified by choosing two angles: an azimuth angle and a slope (or elevation) angle. In Gargoyle3D, azimuth is the angle (-180 to 180 degrees) measured counter-clockwise from the positive *x* axis in the horizontal (*x*-*z*) plane of WORLD. Slope is the angle (-90 to 90 degrees) between the direction vector and the horizontal plane. It would be adequate to let azimuth vary between 0 and 180 degrees. However, providing the larger range allows users to think of directions in whichever of two azimuths is most convenient.

The alignment menus are shown in Figure 3-37. A desired direction of alignment lines is activated by selecting from a menu, called the "Line" menu, of azimuth-slope pairs. Likewise, a desired direction of alignment planes is activated from the "Plane" menu, and a desired radius of alignment sphere from the "Radius" menu. Each of these menus initially contains a set of default values. The "Line" and "Plane" menus initially contain the default values (0 0), (0 90) and (90 0) corresponding to the *x*, *y* and negative *z* axes of WORLD respectively. The "Radius" menu initially contains the values 1/8, 1/4, 1/3, 1/2, 2/3, 3/4, 1, 2, 3, and 4, where the units can be set to inches, centimeters, or any other value.

Azimuth:		Add!	Delete!	150 135 120 90 60 45 30 **10** 0
Slope:		Add!	Delete!	90 60 45 30 **0** -30 -45 -60 -90
Line:	New!	Add!	Delete!	(0 0) (0 90) **(10 0)** (90 0)
Plane:	New!	Add!	Delete!	(0 0) (0 90) (90 0)
Radius:		Add!	Delete!	1/8 1/4 1/3 1/2 2/3 3/4 1 **2** 3 4

Figure 3-37. The alignment menus in Gargoyle3D.

To help the user extend the "Line" and "Plane" menus, two other menus are provided: "Azimuth" with a choice of azimuths and "Slope" with a choice of slopes. To add a new azimuth-slope pair to the "Line" menu, the user activates one item from the azimuth menu and one from the slope menu and clicks the "New" button near the "Line" menu. Pairs may be added to the "Plane" menu in a similar fashion. Figure 3-37 shows the alignment menus after (10 0) lines have been added and activated.

As in two dimensions, all of the alignment menus can also be extended by typing new values and by measuring values from the scene. Whenever an operation is completed, Gargoyle3D

measures the displacement vector through which the skitter has moved since the last operation was completed and reports the length and direction angles of this vector. Any or all of these values may then be added to the alignment menus.

The construction of a tetrahedron will illustrate the use of alignment lines and spheres. In Figure 3-38(a), the user draws a line segment on the default plane. He makes one end of it hot and activates lines with direction angles (10 0) and spheres of radius 2 inches. An alignment line and an alignment sphere appear, centered on the hot vertex (Figure 3-38(b)). Spheres are drawn as a single circle (an ellipse, when perspective is used) representing the silhouette of the sphere. In Figure 3-38(c), the user drags the skitter around on the near surface of the sphere while rubber-banding the line segment. The sphere draws itself darker to show that it is involved in positioning the skitter. As in two dimensions, the intersections between objects are gravity-active. In Figure 3-38(d), the skitter snaps to the intersection point of the sphere with the alignment line. In this example, the projection of the intersection point is near where the projections of the sphere silhouette and the alignment line meet, but this is not true in general.

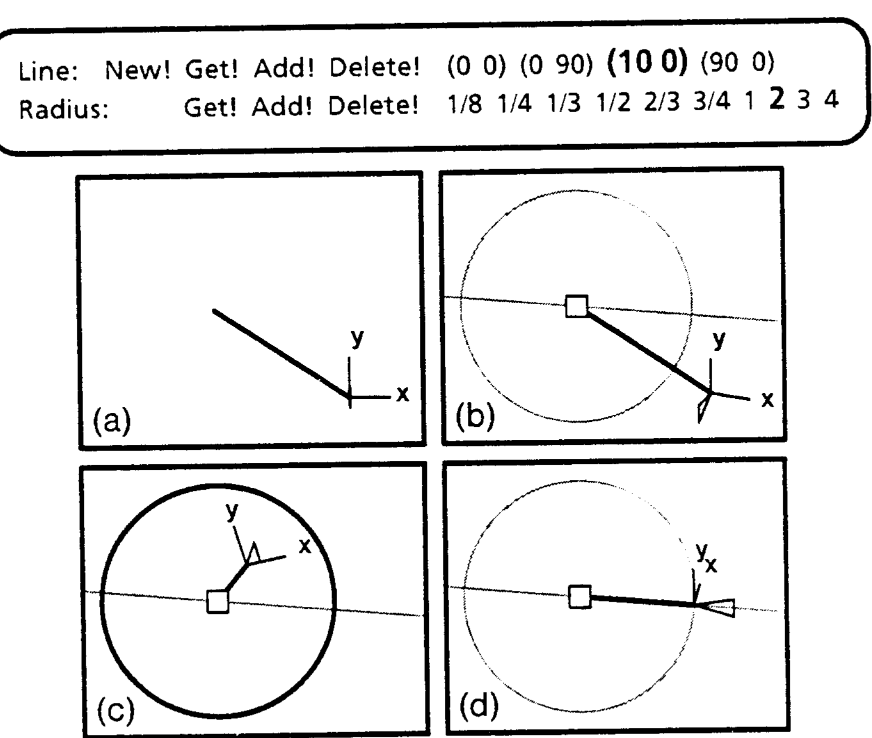


Figure 3-38. Alignment line and sphere.

In Figure 3-39(a), the user has made both endpoints of the line segment hot. Spheres of

radius 2 inches are still active, so the system constructs two alignment spheres centered on the two vertices. In addition, the system draws the circle of intersection of the two spheres. The circle of intersection is drawn in black to distinguish it from the grey silhouettes of the spheres. A segment is stretched from the original segment's endpoint to the circle of intersection. In Figure 3-39(b), a third segment has been drawn to complete an equilateral triangle and the third vertex has been made hot. Gargoyle3D constructs a third alignment sphere and computes its circles of intersection with the two existing spheres. The user stretches a new line segment from a vertex to one of the two points where all three intersection circles meet. All vertices are made cold and two segments are added between existing vertices to complete a tetrahedron (Figure 3-39(c)).

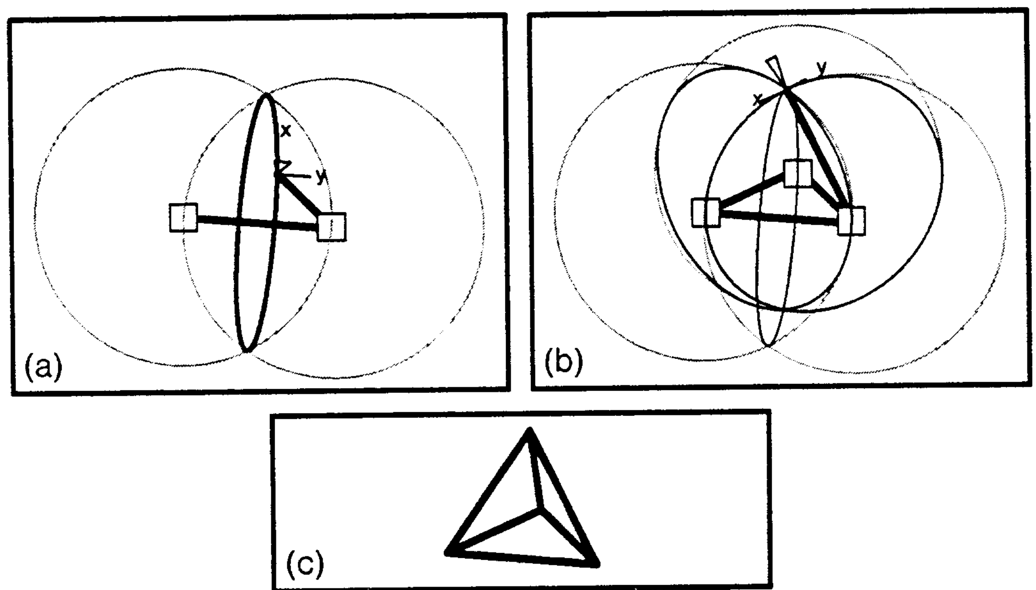


Figure 3-39. Tetrahedron construction. (a) A new line segment is stretched to the circle at which the spheres intersect. (b) A fourth segment is added to a point where all three spheres meet. (c) A fifth and sixth segment are added, completing a tetrahedron.

An alignment plane is drawn as a set of grey-bordered transparent squares, one centered on each vertex that triggers that alignment plane. In Figure 3-40(a), we see a plane with normal direction (0 90) (a horizontal plane) triggered by the anchor. As more plane directions are activated, more planes are constructed and their lines of intersection are computed and drawn as shown in Figure 3-40(b) and Figure 3-40(c).

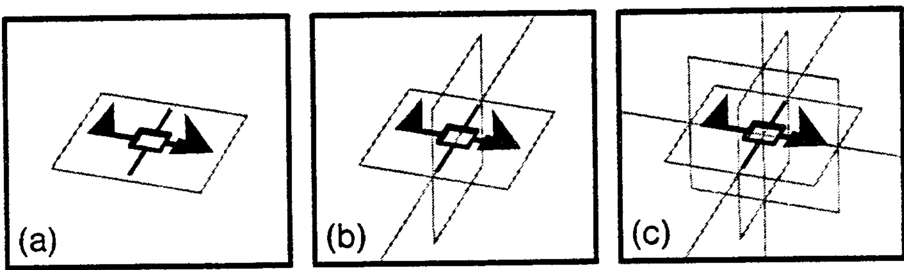


Figure 3-40. Alignment planes triggered by the anchor. (a) A horizontal alignment plane. (b) Two perpendicular alignment planes. (c) Three perpendicular alignment planes.

The user may place the skitter on whichever plane is closest along the ray from the eyepoint through the cursor. In Figure 3-41(a), the skitter is placed on the horizontal plane, in Figure 3-41(b) the left-hand vertical plane. When the skitter snaps to an alignment plane, the feedback square for that plane darkens to provide an extra cue as to the skitter's position in space. With points-preferred or lines-preferred gravity, lines are preferred to surfaces, allowing the user to place the skitter on an intersection line even though it is behind an alignment plane (Figure 3-41(c)). When the skitter snaps to an intersection line, both planes that intersect at that line highlight.

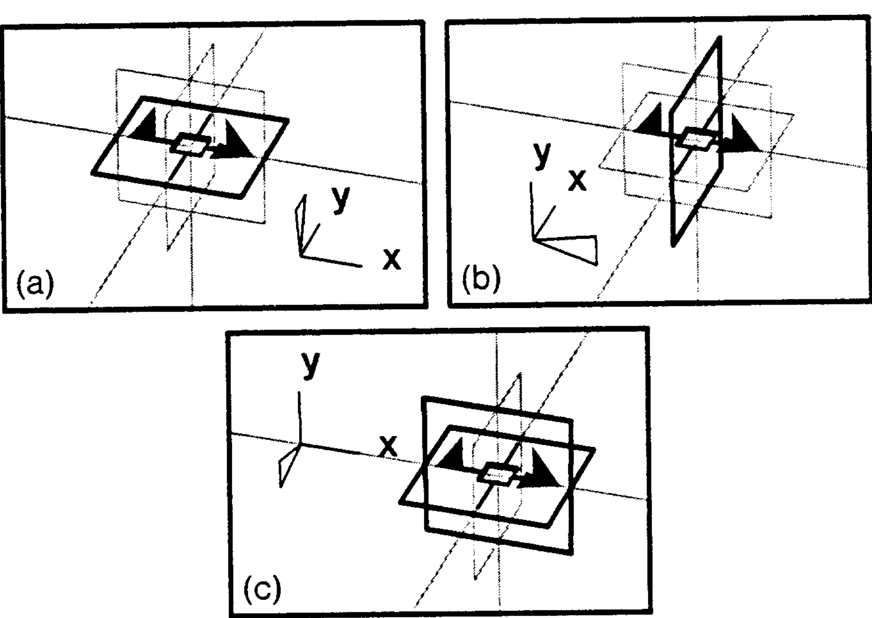


Figure 3-41. Snapping to alignment planes. (a) The skitter moves on the horizontal plane. (b) The skitter moves on a vertical plane. (c) The skitter snaps to a line behind the grey vertical plane.

Alignment planes and alignment spheres can work together. Figure 3-42(a) shows the anchor

triggering an alignment sphere and three mutually perpendicular planes. The intersection circles of the planes with the sphere are automatically drawn and made gravity-active. The intersection lines produced by the planes intersect with the sphere in 6 points. Figure 3-42(b) and (c) shows an octahedron constructed by stretching line segments between these points.

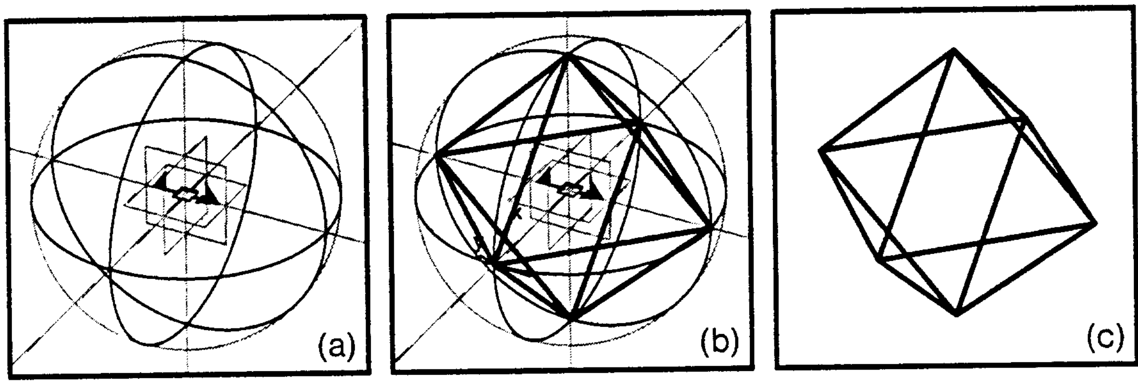


Figure 3-42. Constructing an octahedron. (a) An alignment sphere, four alignment planes, and their intersection curves. (b) An octahedron is constructed. (c) The result.

3.2.4 Advanced Snap-Dragging Operations in Three Dimensions

The same operations that support snap-dragging in two dimensions also support it in three. This section shows heterogeneous selections, different gravity functions, and constrained objects at work in three dimensions. In addition, this section illustrates the use of alignment objects to facilitate rotations.

Measurement of slopes and distances and the extension of alignment menus are also interesting in three dimensions, but these facilities are similar enough to their two-dimensional counterparts that no further examples are given. Also, as curved surfaces have not yet been implemented in Gargoyle3D, no examples of curved surface editing are available to complement the curve editing examples of section 3.1.4.

In Figure 3-43, a mixed selection is used to make a ladder narrower in a single dragging operation. The user selects the far ends of the three blocks that make up the steps of the ladder and selects the left leg of the ladder in entirety. When he translates this collection of objects, the left leg moves and the steps shrink along their long dimension to make the ladder narrower. The alignment line is parallel to the steps and is used to keep the ladder square.

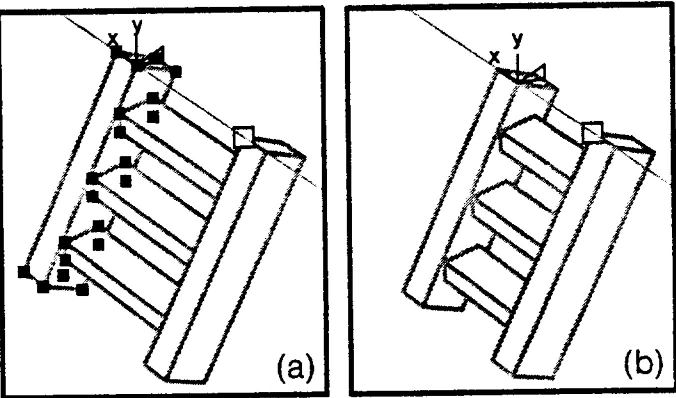


Figure 3-43. Heterogeneous selection. (a) The user selects a collection of parts. A hot point triggers an alignment line. (b) The user drags the selected parts along the alignment line.

Faces-preferred gravity is useful when the user wishes to snap an object to a face, but there are points or edges nearby that the user does not want to snap to. For instance, in Figure 3-44(a), the user has selected a truck (drawn as two blocks) and wishes to place it on the roadbed of a trestle bridge. He does not want the truck to snap to the edges of the supporting structure. With faces-preferred gravity, the truck can be placed anywhere along the roadbed as shown in Figure 3-44(b).

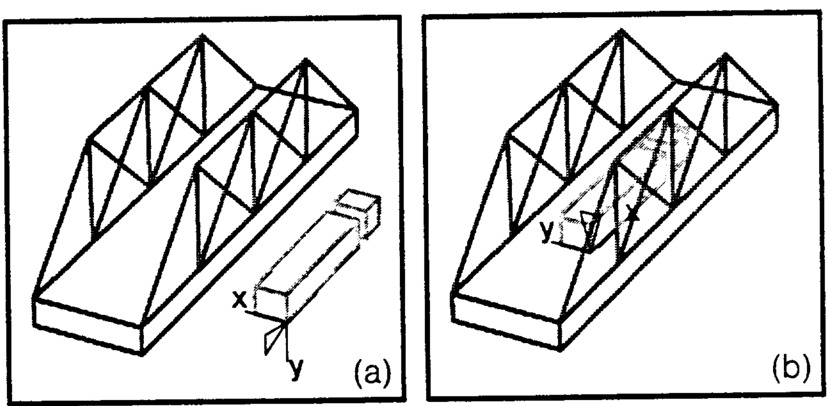


Figure 3-44. Truck and trestle. (a) The skitter is placed on the selected truck. (b) The truck is snapped onto the roadbed using faces-preferred gravity.

Just as constrained rectangles were useful in two dimensions, constrained blocks are useful in three dimensions. For constrained blocks, we can change combinations of height, width, and depth by selecting either a vertex, an edge, or a face. Figure 3-45(a) shows a block being stretched between two walls. If a face is selected and a "Translate" operation is performed, then the width

of the block changes to keep the skitter coplanar with that face, as shown in Figure 3-45(b). If an edge is selected, "Translate" changes both the width and height of the block to keep that edge collinear with the skitter as shown in Figure 3-45(c). Finally, if a vertex of the block is selected, "Translate" changes width, height, and depth to keep the selected vertex coincident with the skitter as shown in Figure 3-45(d).

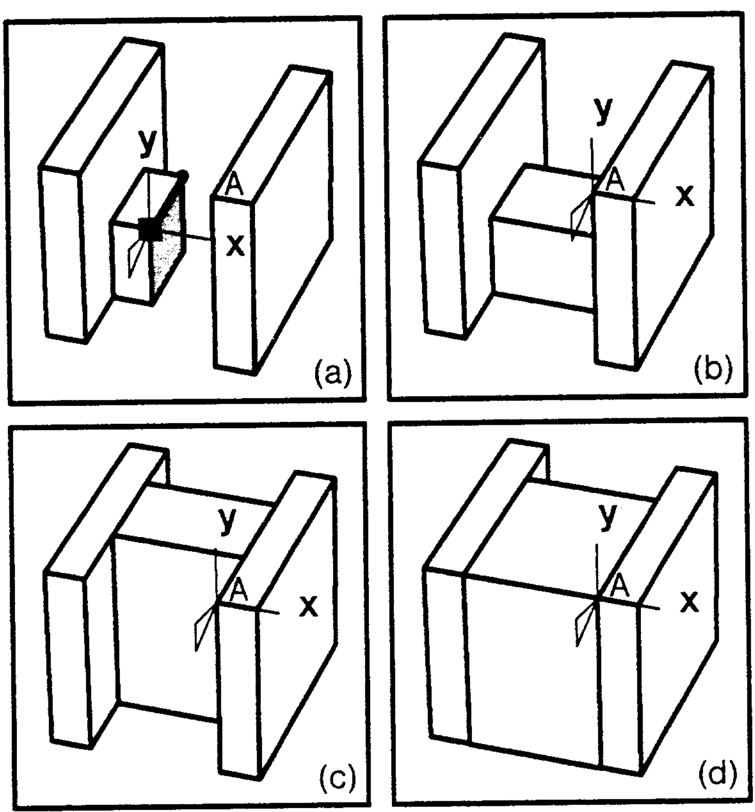


Figure 3-45. Stretching a constrained block. (a) The user selects either a joint, an edge, or a face and places the skitter at the highlighted vertex. Next, the "Translate" operation is performed until the skitter snaps to point A. (b) With the gray face of (a) selected. (c) With the thickened edge of (a) selected. (d) with the highlighted vertex of (a) selected.

When used with alignment objects, the "Rotate" operation can be used to perform free-form rotations. For instance, in Figure 3-46(a), the user has placed the anchor at the centroid of a cube and activated (0 0), (0 90), and (90 0) alignment planes and an alignment sphere. By moving the skitter along one of the intersection circles during a "Rotate" operation, the user rotates the cube around an axis that is perpendicular to the plane of the circle and passes through the origin of the circle (Figure 3-46(b)). By performing several rotations using two or more circles, the user can

arbitrarily orient the cube in a manner similar to that provided in other systems with three sliders or three dials.

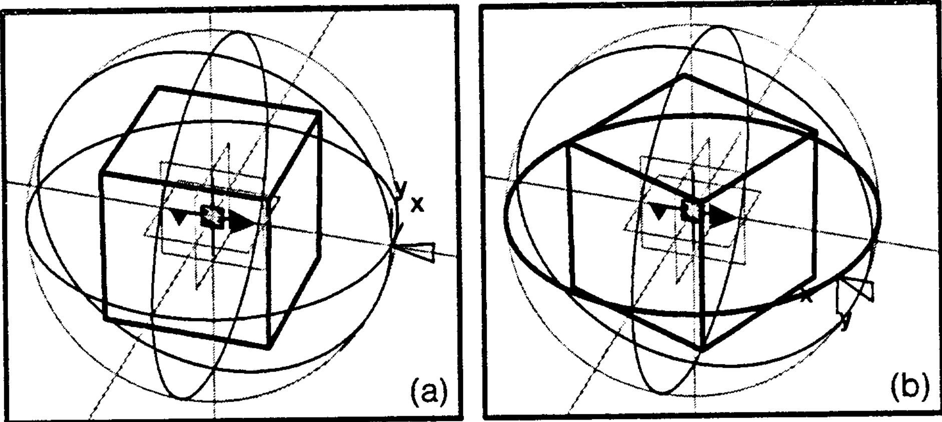


Figure 3-46. Freeform rotation. (a) The skitter is placed at the intersection of two alignment planes and an alignment sphere. (b) The "Rotate" operation is performed while the skitter is moved along a horizontal intersection circle.

Rotations about an arbitrary axis can be achieved using the alignment sphere by itself. If the skitter is placed on the sphere at the beginning of the rotation and moved along the surface of the sphere, the selected objects are rotated about an axis perpendicular to the great circle that passes through the beginning and ending skitter positions. The objects rotate as though embedded in a transparent two-degree-of-freedom track ball, where the skitter defines the starting and ending positions of a single point on the track ball's surface. This rotation method is similar to the "virtual sphere" technique described by Chen et al [Chen88] except that it uses absolute cursor positions as input instead of instantaneous cursor motion directions and the rotation is about a distinguished point (the anchor) instead of the focal point of viewing.

3.2.5 User Interface Summary

The user interface for Gargoyle3D is only slightly more complicated than that of Gargoyle. The same mouse actions are provided, except that "RotateX" is used instead of "Skew", and "Add Constrained Block" is provided instead of "Add Constrained Rectangle".

Likewise, the same keyboard actions are available. In three dimensions, "Forward to Next Gravity Type" cycles through the three types points-preferred, lines-preferred, and faces-preferred instead of the two types available in Gargoyle. The "Gravity Extent", "Gravity",

"Midpoints", and "Auto" buttons work just as they do in two dimensions. As discussed above, Gargoyle3D has five alignment menus compared to four in Gargoyle. Furthermore, the Line Menu and Plane Menu have an extra button, "New!", that is not found in Gargoyle.

3.3 Feedback: Graphical and Textual

In snap-dragging, the user must know which objects are selected, which object part the caret has currently snapped to, where the control points of the spline segments are, and where the alignment objects are. In order to keep the user's attention focused on the picture itself, Gargoyle and Gargoyle3D provide graphical feedback for these types of information. In a WYSIWYG ("What you see is what you get") illustration system, all stationary graphical feedback is potentially ambiguous. The user can draw a picture that looks just like the caret or the anchor or our highlighted control points. Furthermore, for detailed work, the feedback can get too cluttered in the work region to be of any use. For these situations, we also provide textual feedback in a region distinct from the drawing area.

In order to show the user what object parts are selected, Gargoyle and Gargoyle3D both draw small black squares on the vertices of selected parts. This feedback is effective even on bi-level displays. In addition, the vertices of the non-selected parts are highlighted with small white squares. This additional feedback serves two purposes. First, it identifies the object whose part is being selected. Second, it forces the display of off-curve and off-surface control points, which are usually suppressed to reduce screen clutter. In Figure 3-47(a), a single vertex of a Bézier spline is selected. The white squares show that the selected vertex does not belong to the adjacent grey rectangle, and also indicate the locations of the control points of the spline. Figure 3-47(b) shows a scene of three blocks in which a vertex has been selected. The vertex could belong to any of the three blocks. The white control squares show that the vertex in fact belongs to the upper block, and also indicate the locations of the vertices and the centroid of that block.

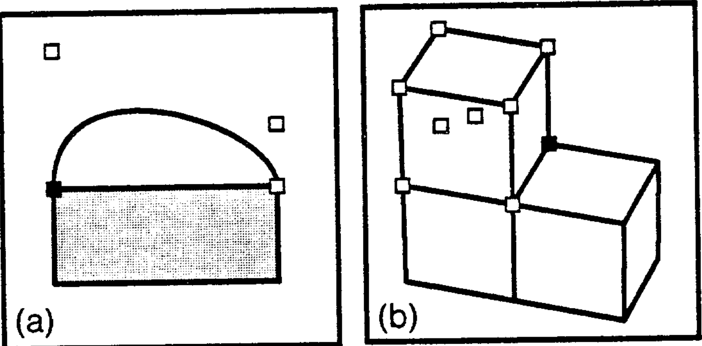


Figure 3-47. Selecting a vertex. (a) The vertex of a Bézier spline is selected. (b) The vertex of the upper block is selected.

The object that the caret or skitter is snapping to is called the *attractor.* The two Gargoyle editors highlight with white squares those parts of the attractor nearest to the caret or skitter. This feedback helps the user place the software cursor in cluttered or hidden scene regions. Figure 3-48(a) shows the caret snapping to a line segment in a cluttered region. Gargoyle places white squares at the end of this segment, effectively indicating to the user that the caret is on the centermost of the three parallel lines. In Figure 3-48(b), the user is snapping the skitter to the hidden vertex of a block. All of the adjacent vertices of the block highlight as does the block centroid. This quickly shows that the skitter is snapping to a vertex, that the vertex is part of the top cube, and that the vertex is hidden.

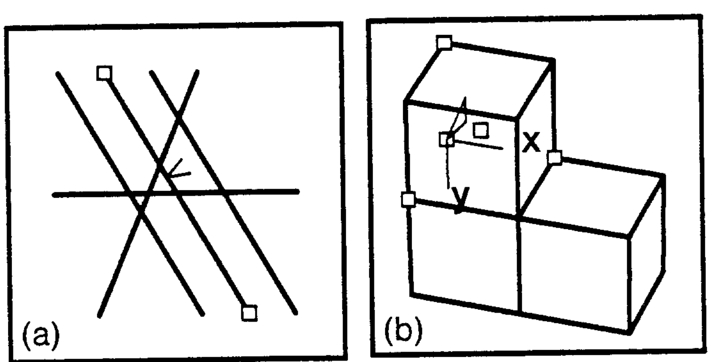


Figure 3-48. Attractor feedback. (a) The caret snaps to a line segment in a cluttered region. (b) The skitter snaps to a hidden vertex.

Table 3-3 shows the attractor feedback that is provided in each of four situations in two dimensions. When the caret is on a vertex, the vertices and control points of the two neighboring segments are highlighted as well as the vertex itself. This helps the user determine if the correct

vertex is being picked in cluttered scene regions. When the caret snaps to a Bézier spline, the off-curve control points are displayed. Because of this, no special commands need to be provided to turn the display of control points on and off.

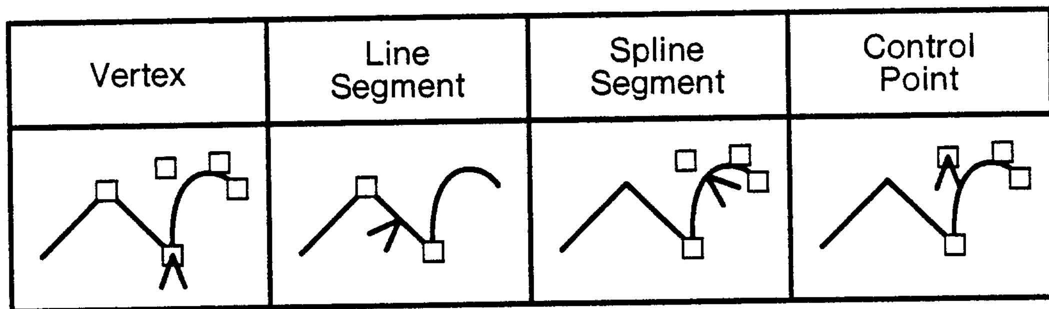


Table 3-3. Attractor feedback in two dimensions. Different vertices are highlighted depending on whether the caret snaps to a vertex, segment, or control point.

Table 3-4 shows the attractor feedback when the skitter snaps to each of the parts of a Block object. For vertices, the three neighboring vertices are highlighted along with the centroid point. For edges, all of the vertices of the neighboring faces are highlighted along with the centroid. For a face the vertices of the face are highlighted along with the centroid. For the centroid itself, all of the block vertices and the centroid are highlighted. These different types of feedback allow the user to distinguish between selecting a face, selecting an edge, and selecting a vertex. All of these types of feedback show the position of the centroid in case the user wants to snap to that next.

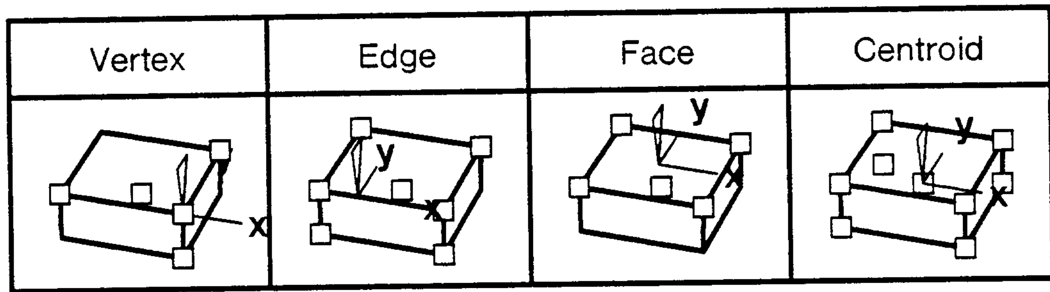


Table 3-4. Attractor feedback in three dimensions. Different vertices are highlighted depending on whether the skitter snaps to a vertex, edge, face, or centroid, as shown.

When many alignment objects are present, it becomes more difficult for the user to tell if the software cursor is snapping to the correct one. Gargoyle3D provides extra feedback to reduce ambiguity. When the skitter snaps to an alignment object, the vertices or edges that trigger that

alignment object are highlighted. For instance, in Figure 3-49, alignment lines with directions (0 0) and (90 0) are active and all of the corners of the two blocks are hot. The skitter is snapping to the intersection point of two alignment lines. All of the hot corners that are triggering those two lines are highlighted with an asterisk shape. If the skitter were on a line, but not on an intersection point, only the vertices that trigger that line would be highlighted.

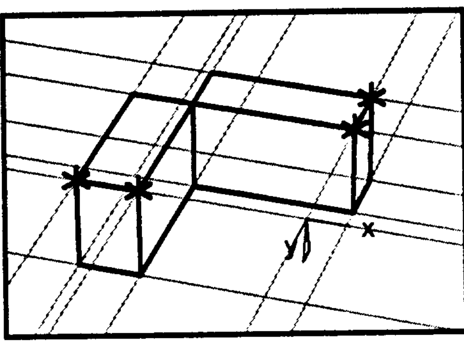


Figure 3-49. The skitter snaps to the intersection point of two alignment lines. The vertices that triggered those two alignment lines are highlighted with an asterisk.

During all snap-dragging operations, a line of text, called the *feedback line* describes the operation in progress. This textual feedback supplements all three kinds of graphical feedback described in this section. When part of an object is selected, or when the software cursor snaps to part of an object, the part type and object type are described textually (e.g., "skitter on the front face of a block"). Likewise, when the software cursor snaps to an alignment object that object is described (e.g., "caret on a slope line/slope line intersection point").

3.4 Selecting Objects

As we have already seen, much of the power of snap-dragging comes from the ability of the user to select arbitrary collections of vertices, segments, and faces. Often, we need to be able to select objects in cluttered parts of the picture or parts of objects that are underneath other objects. Many of these problems can be solved using indirect selection or cycling selection.

Consider a scene such as the one in Figure 3-50(a). Three line segments end at the same point. If we wish to select the right-hand vertex of the leftmost segment, we may have difficulties. If Gargoyle's only criterion for selecting vertices were their distance from the cursor, it would have to arbitrarily choose one of the three endpoints. Instead, Gargoyle uses points-preferred

gravity to find objects near the cursor. If we place the cursor directly over the vertex, we encounter the ambiguity just discussed. However, if we move away from the vertex and place the cursor near one of the edges, the gravity function describes that edge to the selection algorithm. Since we are performing a vertex selection operation, Gargoyle computes the nearest vertex of the edge to the cursor and selects that. Using this indirect selection technique, we can easily select the vertex that we want. Figure 3-50(b) shows the result of translating the selected vertex.

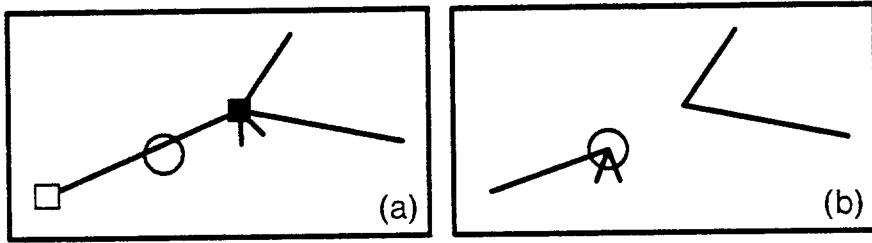


Figure 3-50. (a) Selecting the endpoint of a segment by pointing at the segment. The hardware cursor is shown as a circle. (b) Dragging the selected endpoint.

The trick used above may fail if the scene is cluttered near the line segment as well as near the vertex. In this case, the user can fall back on the ability to cycle through a number of selections. After every selection is made, the Gargoyle editors remember the nearest object to the cursor and all of the objects that are almost as close as the nearest object. The "Cycle Selection" command, available from a menu or keyboard combination, selects each of these in turn.

In three dimensions, we often want to select an object that is behind something else. For instance, Figure 3-51 shows a pair of cubes placed one behind the other. We would like to select the rightmost face of the far cube. To do this, we place the cursor as shown in Figure 3-51(a) and invoke the "Select Face" command. This command selects the nearest face under the cursor, which turns out to be the front face of the near block. If we use the "Cycle Selection" command, Gargoyle3D deselects this face and selects the next farthest face, the leftmost face of the right-hand block. We repeat the "Cycle Selection" command one more time to select the rightmost face of the left-hand block, as shown in Figure 3-51(b).

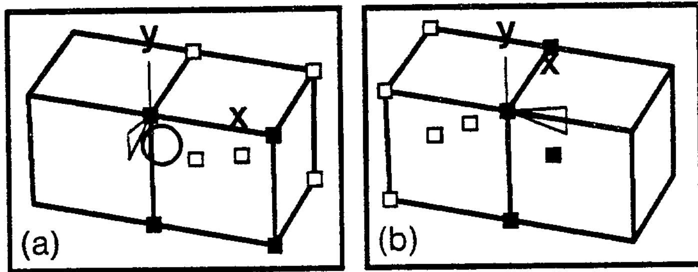


Figure 3-51. Selecting a hidden face. (a) The "Select Face" operation selects the nearest face. (b) By cycling through the nearest faces behind the cursor, the user selects a hidden face.

3.5 Unusual Constructions

While snap-dragging shares some ideas with traditional ruler and straightedge construction, the availability of affine transformations parameterized by the caret changes the way one thinks about constructing shapes. Objects need not be constructed in place; they can be constructed at any convenient orientation and scale, and then translated, rotated, and scaled into place. Furthermore, the caret need not remain near the object that is being edited. It can be placed on objects in one region while remotely reshaping objects in another. This section shows examples of this principle at work.

To construct the largest square that fits in a hexagon, we can construct a square at any convenient size and scale it up to fit, using a 45 degree slope line to identify the point where the square should touch the hexagon, as shown in Figure 3-52.

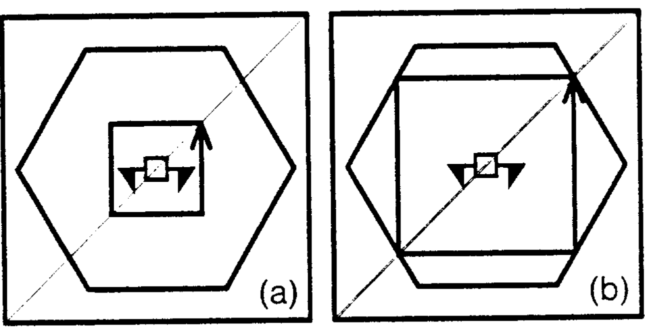


Figure 3-52. (a) The caret is placed on a corner of the square. (b) The square is scaled until the caret snaps to the intersection of the 45 degree slope line with the hexagon.

The construction of the square in the hexagon was discovered by using snap-dragging as a

tool for learning about geometry. The square was scaled and rotated about the anchor, with gravity turned off, to discover those angles at which the square could be made the largest while still fitting in the hexagon. Once an approximate configuration had been constructed, it was a relatively easy matter to prove that the construction of Figure 3-52 produces the largest square.

In Figure 3-53, we rotate an arrow to point to the center of the circle, B. The rotation begins with the caret on the arrow at point A (Figure 3-53(a)) and finishes with the caret on B (Figure 3-53(b)). As a result, the anchor point, point A and point B are collinear.

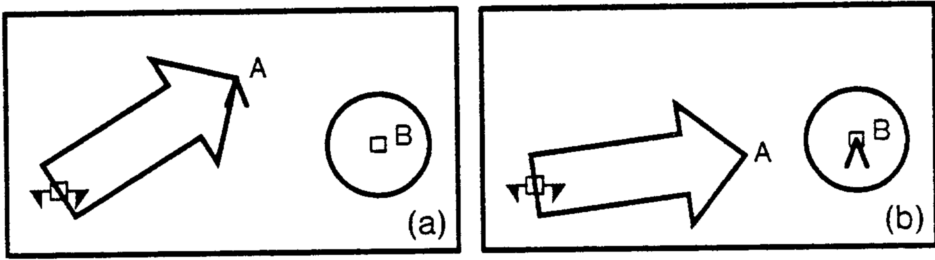


Figure 3-53. A rotation operation where the caret leaves the rotating object.

Translations where the caret leaves the object being moved are also useful. For instance, In Figure 3-54, the user has placed a house in one frame of a comic strip. If he wishes the house to appear at the same position in the other frames of the strip, he can copy the shape in place and position the caret on a corner of the first frame (Figure 3-54(a)), and then translate the copy until the caret snaps to the corresponding corner of the second frame (Figure 3-54(b)).

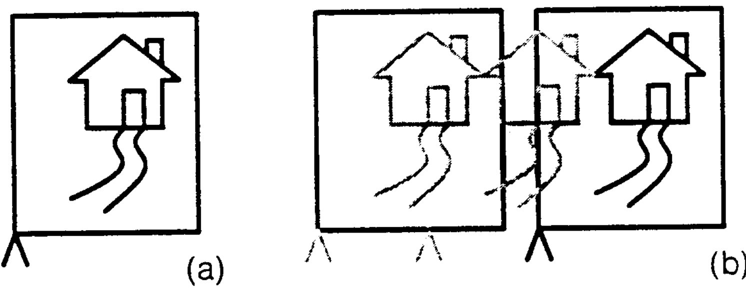


Figure 3-54. Copying a displacement. (a) The house is copied and the caret is snapped to the lower left corner of the first frame. (b) The copy is translated until the caret snaps to the lower left corner of the second frame. Intermediate positions are shown in grey.

4. Snap-Dragging Implementation

Sow seed in flats from late summer to early spring for later transplanting, or buy flat-grown plants at nursery. Set out plants in early fall in mild-winter areas, spring in colder sections. If snapdragons set out in early fall reach bud stage before night temperatures drop below 50°F, they will start blooming in winter and continue until weather gets hot.

– Sunset New Western Garden Book

Snap-dragging reduces the human time needed to construct precise scenes by performing many computations, only a few of which directly aid the user. In particular, all objects participate in the gravity computation, even though the user will only point to one at a time, many alignment objects are computed and drawn, even though only two or three will contribute to the user's next constructed point, and many intermediate positions of objects are displayed during interactive affine transformations, even though the final position is the only one that is important in the end. Snap-dragging, then, is likely to be computationally expensive.

In this chapter, I describe the algorithms used to compute the gravity function, select objects, update the set of alignment objects, and interactively transform objects. These algorithms use on-the-fly calculations, incremental calculations, caching, and hashing to achieve good performance. Section 4.1 describes the implementation of gravity in two and three dimensions. Section 4.2 describes the computation of the active set of alignment objects. Section 4.3 describes the computations that take place during interactive transformations.

Gargoyle and Gargoyle3D are both written in the Cedar programming language, an extension of the Mesa language [Mitchell79], which in turn is based on Pascal [Jensen74]. Both editors run in the Cedar programming environment [Swinehart86], on the Dorado high-performance personal workstation [Lampson81, Pier83]. Not counting comments, the Gargoyle editor is implemented in 35,557 lines of code, of which the gravity algorithm takes 973 lines, the algorithm for maintaining the active set of alignment objects and gravity-active scene objects takes 1305 lines, and implementation of interactive transformations, caret positioning, and interactive creation of line and rectangle objects takes 954 lines. The Gargoyle3D editor is 43,921 lines of code, of which gravity takes 1420 lines, updating the alignment objects takes 836 lines, and interactive transformations, etc. take 1085 lines. The implementation of alignment objects in Gargoyle3D does not yet include all of the optimizations of the two-dimensional version.

4.1 Gravity

4.1.1 Two-Dimensional MultiMap

In the user's view, gravity is a user interface option that affects the behavior of the caret, causing it to snap to an object that is near to the cursor if some object is near enough. From an implementor's point of view, gravity is a routine that takes the current cursor position and a description of the scene objects and alignment objects and returns information about those objects that exist in a small region around the cursor. In Gargoyle, the gravity routine, MultiMap, is used to implement both caret placement operations and object selection operations. In this section, I describe what MultiMap does, how it is used to implement caret placement and object selection and how it is implemented.

To a first approximation, MultiMap behaves as follows: Given the set of gravity-active objects (scene objects and alignment objects), a position of the hardware cursor **q**, and a tolerance distance, t, MultiMap finds all of the objects that are within distance t from **q**. From this set, MultiMap extracts the nearest object to **q**, o_{min}, and all objects that are almost as near to **q** as o_{min} (within floating point round-off). For each of these "nearest" objects, MultiMap computes the nearest point, **p**, on that object to **q**, the curve normal of that object at **p**, and a description of the object sub-part at which **p** occurs (e.g., the segment or vertex that **p** is on). MultiMap returns the list of "nearest" objects and their corresponding points **p**, normals, and sub-part descriptors.

In addition to describing the scene objects and alignment objects that are near **q**, MultiMap also computes the pair-wise intersections of these objects on the fly, and reports them just as it would report a vertex. In particular, if an intersection point is close enough to **q**, MultiMap will return the coordinates of the point of intersection, the curve normal at that point of one of the curves that intersect there, and the sub-part descriptors of both of the curves that intersect there. MultiMap will also compute the midpoints of all straight line segments on the fly. The intersection and midpoints options can be turned off if MultiMap's caller is not interested in these quantities.

Chapter 3 described two different gravity functions for placing the caret. *Points-preferred* gravity makes it easy to place the caret on vertices and points of intersection. *Lines-preferred* gravity simply places the caret on the nearest curve; it does not treat points specially. A parameter to

MultiMap tells it which of these two gravity functions to perform as it is narrowing down the set of objects that are within t from $\mathbf{q}$. A third function, *faces-preferred* gravity will also be described here, even though it is more useful in three dimensions than in two dimensions, because it allows two-dimensional MultiMap to be defined nearly identically to three-dimensional MultiMap.

Gargoyle is a 2 1/2-dimensional picture editor. A Gargoyle picture is made of a set of planar objects, that are drawn in a specified order, back to front. If we number the objects from back to front, we can think of these numbers as discrete depth coordinates. Some of these objects are opaque closed shapes. These objects have non-zero interior areas (faces). If one or more faces are behind a particular position of the cursor, then shooting a ray starting at the cursor in the direction perpendicular to the screen plane will cause the ray to skewer these faces.

Faces-preferred MultiMap returns all of the faces that are hit by the cursor ray, sorted by depth, near to far. If the ray hits nothing, MultiMap instead returns the nearest edge or vertex o_{min} that is within screen distance t of $\mathbf{q}$ and all edges and vertices that are almost as close to $\mathbf{q}$ as o_{min} (within floating point round-off). Notice, that if any object is within t of $\mathbf{q}$ than faces-preferred MultiMap will return something. It prefers to return faces, if any are under the cursor.

Faces-preferred MultiMap returns other information as well. If faces are hit by the cursor ray, MultiMap returns a list of these faces, pairing each element with the point $\mathbf{q}$ and a default normal direction. Otherwise, if edges or vertices are within t of $\mathbf{q}$, MultiMap associates with each edge in the list the nearest point on that edge to $\mathbf{q}$ and the curve normal at that point. The sign of the curve normal is chosen so that the normal points to the same side of the curve that $\mathbf{q}$ is on. Associated with each vertex in the list, is the coordinates of the vertex and the curve normal of one of the edges that meet at that vertex, namely the edge whose tangent direction is angularly closest to the line from the vertex to $\mathbf{q}$. Figure 4-1 shows the point and normal direction that MultiMap returns for several cursor points $\mathbf{q}$ in a scene consisting of a single triangle.

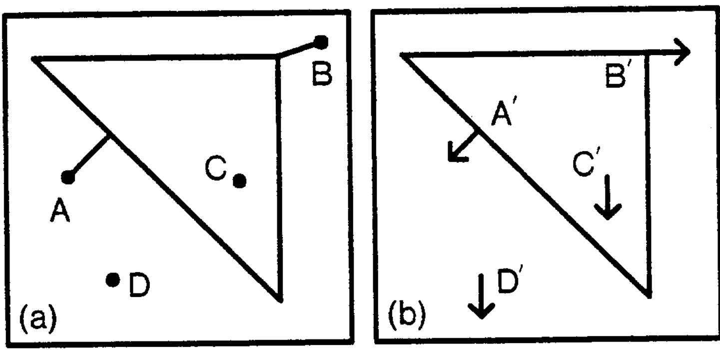


Figure 4-1. Faces-preferred gravity examples. (a) For four hardware cursor positions A, B, C, and D a line shows where these points would map. A, B, and C are within distance *t* of the triangle, but D is not. C and D map to themselves. (b) A′, B′, C′ and D′ are the images of the original points. The arrows show the curve normal at each point. C′ and D′ are associated with the default normal.

With a gravity-field diagram, we can summarize the results of MultiMap for all possible positions **q** in a given scene. Such a diagram is shown in Figure 4-2(b) for faces-preferred gravity around the triangle shown by itself in Figure 4-2(a). The tolerance radius in this figure is about 0.2 inches. If **q** is within the light-grey region, MultiMap returns the the point **q**, a default normal, and a description of the interior of the triangle. If **q** is within the dark-grey region, MultiMap returns the nearest point on the closest edge to **q**, the normal direction to the edge at that point, and a description of the edge. If **q** is within an arc-bounded white region, MultiMap returns the coordinates of the nearest vertex to **q**, the normal direction to one of the edges that meet at that vertex (the edge whose normal direction is angularly closest to the direction from the vertex to **q**), and a description of the vertex. In all other regions, MultiMap returns **q**, a default normal, and a description of empty space.

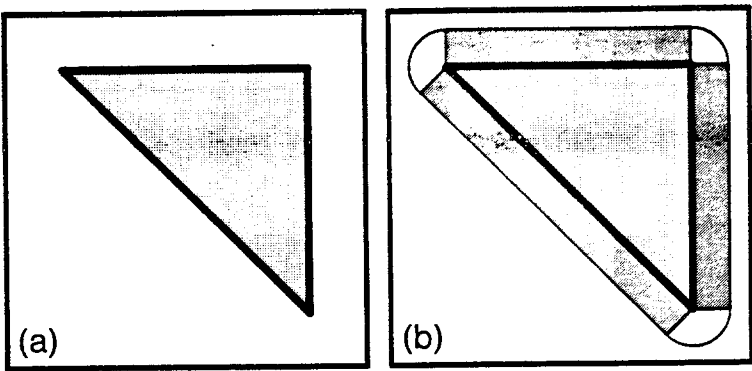


Figure 4-2. Faces-preferred gravity. (a) The triangle as it appears to the user. (b) The faces-preferred gravity field around the triangle.

Lines-preferred MultiMap gives first priority to edges. If any edge or vertex is within distance t of the cursor, MultiMap returns a list of edges and vertices that includes the nearest edge or vertex and all edges and vertices that are almost as near as the nearest (within floating point round-off), pairing them with points and normals as described for faces-preferred gravity; in this case, none of the descriptors returned by MultiMap describe faces. Otherwise, the result of faces-preferred MultiMap is returned. Figure 4-3 shows the lines-preferred gravity fields for two intersecting lines and for the triangle of Figure 4-2(a). Notice in Figure 4-3(b) that, MultiMap still returns a description of the triangle's interior if **q** is positioned in the light grey region in the middle of the triangle. However, MultiMap will no longer map **q** to itself when **q** is positioned over interior points that are within t of an edge.

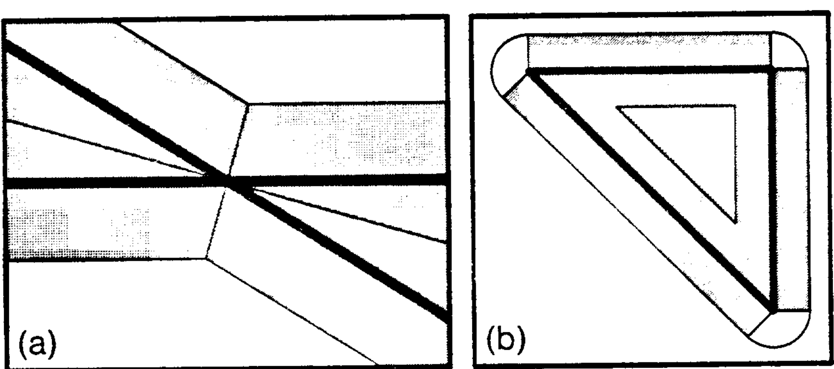


Figure 4-3. Lines-preferred gravity. (a) The field around two intersecting heavy black line segments. (b) The field around the triangle of Figure 4-2(a). The arcs have radius *t*.

The main shortcoming of lines-preferred gravity is that points of intersection are difficult to

snap to. As shown in Figure 4-3(a), the only point in the plane that maps to the point of intersection is the point of intersection itself. The *points-preferred* gravity function remedies this problem by giving vertices and points of intersection first priority; if **q** is within distance r from a vertex or intersection point (where r is a constant and $r \leq t$), then MultiMap returns a list of points that includes the nearest point and all points that are almost as near as the nearest (within floating point round-off). Otherwise, the result of the lines-preferred gravity function is returned. Figure 4-4 shows the gravity field diagrams for points-preferred gravity. Points **q** in the white circle of Figure 4-4(a) map to the intersection point. Comparing this function to lines-preferred gravity, we see that we have given up our ability to map to some of the points on the lines near the intersection point in order to make the intersection point itself easy to map to.

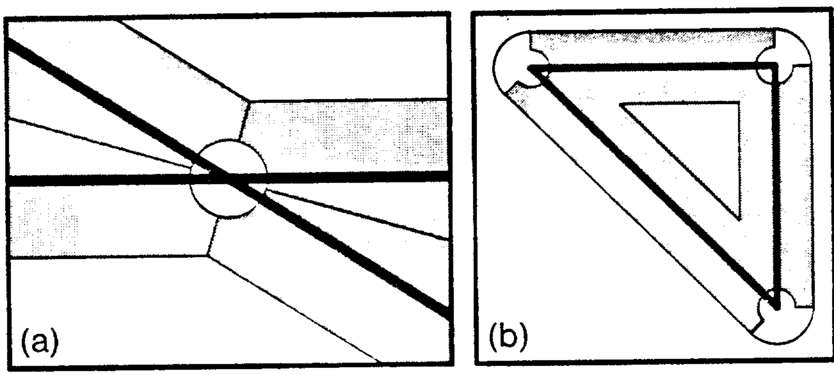


Figure 4-4. Points-preferred gravity. (a) The field around two intersecting line segments. The circle has radius r. (b) The field around the triangle of Figure 4-2(a). The large arcs have radius t, the small arcs radius r, where $r = t / 2$.

4.1.2 Caret Placement and Selection

We can build the snap-dragging caret placement operations for points-preferred and lines-preferred gravity using MultiMap as a first step. For instance, to implement points-preferred gravity, we look at the set of objects that points-preferred MultiMap returns. If the set is empty, we place the caret at **q** using a default orientation. If the set contains a single object, we place the caret at the point and orientation that MultiMap associated with this object. Otherwise, we must arbitrate among several objects. We choose the nearest scene object, if any are in the set, or the nearest alignment object otherwise. The caret is placed at the point and orientation associated with this object. To implement lines-preferred gravity, we begin with the set of objects that *lines-*

preferred MultiMap returns and narrow them down as just described.

One of the reasons MultiMap returns sub-part descriptors is to support attractor feedback. When the caret is placed, the sub-part descriptor returned by MultiMap determines the type of attractor feedback that is displayed (recall Tables 3-3 and 3-4).

We can also build a set of selection operations using MultiMap as a first step. In this case, we pass scene objects, but not alignment objects, as arguments to MultiMap and ask MultiMap not to compute intersections. To implement vertex selection, we use MultiMap with points-preferred gravity and a small tolerance distance (say 1/4 inches). After we narrow the results down to a single object, we look at the sub-parts descriptor for the object that MultiMap returns. If this descriptor describes a vertex, we select that vertex. If this descriptor describes a segment or interior, we return the nearest endpoint of that segment or the nearest corner of that interior. This is useful for selecting an endpoint in a cluttered picture region, as was demonstrated in section 3.4.

Similarly, to select segments, we use lines-preferred gravity and pick one object from the set that MultiMap returns. If this object describes a segment, we select that segment. If this object describes a vertex, we select the segment that ends at that vertex and whose normal direction was returned by MultiMap. If this object describes the interior of a path, we choose the nearest segment of that path to **q**.

4.1.3 *Implementing Two-Dimensional MultiMap*

In the Mesa programming language [Mitchell79] MultiMap is defined as

MultiMap: PROCEDURE [q: Point, *t*: REAL, alignBag: AlignBag, sceneBag: SceneBag, gravityType: GravityType, intersections: BOOLEAN, midpoints: BOOLEAN] RETURNS [nearVEF: NearVertexEdgeAndFaces, count: INTEGER],

where **q** is the cursor point, *t* is the tolerance radius, "alignBag" describes the alignment objects, "sceneBag" describes the gravity-active scene objects, "gravityType" is one of {preferPoints, preferLines, preferFaces}, intersections is TRUE if MultiMap should compute intersections on the fly, and midpoints is TRUE if MultiMap should compute midpoints of segments on the fly. The return value "nearVEF" is a fixed-length array of records, where each record has a point, normal, and sub-part descriptor for one of the objects near **q**. Each element also stores the distance of its associated object from **q**. Finally, "count" tells how many elements of "nearVEF" have valid data.

In order to compute MultiMap, we must be able to perform three basic sub-tasks:

1) Find the region interiors (faces) that are directly behind **q**,

2) Find the line and curve segments that are within t of **q**,

3) Find the vertices, control points, and intersection points that are within t of **q**.

These tasks are performed by three routines FacesBehind, CurvesInNeighborhood, and PointsInNeighborhood. I will define these routines precisely and show how to combine them to implement lines-preferred MultiMap and points-preferred MultiMap, with on-the-fly intersections. The implementation of faces-preferred MultiMap is described in section 4.1.5. In my implementations, MultiMap and all three supporting routines return arrays of length 20. If there are more than 20 "nearest" objects, then only the nearest 20 of these are returned. The consequences of this limit are discussed in Appendix A.1.

FacesBehind takes as arguments a set of scene objects and the cursor point **q**. It returns the set of objects that are hit by the cursor ray, sorted in near to far depth order.

CurvesInNeighborhood takes a set of scene objects, a set of alignment objects, the cursor point **q**, the tolerance radius t, and the inner tolerance radius r (recall section 4.1.1) used for points-preferred gravity. It returns the union of two sets of curves:

1) All curves that are within radius r of **q**.

2) Any of the following curves so long as they are within t of **q**: the nearest curve o_{min} to **q** and all curves that are no more than s farther from **q** than o_{min}, where s is a small quantity of the order of floating-point round-off.

These curves are ordered by distance from **q**, nearest to farthest.

PointsInNeighborhood takes the curves returned by CurvesInNeighborhood, a set of scene objects, a set of alignment objects, the cursor point **q**, the tolerance radius t, the inner tolerance radius r, a boolean that is TRUE if intersections should be computed, and a boolean that is TRUE if midpoints should be computed. It returns a list of points, vertices and intersection points, all of which are within t of **q**. If this list is non-empty, the list includes the nearest point o_{min} to **q** and all points that are no more than s farther from **q** than o_{min}, where s is a small quantity of the order of floating-point round-off. These points are ordered by distance from **q**, nearest to farthest.

FacesBehind, CurvesInNeighborhood, and PointsInNeighborhood all use bounding-box

culling to quickly reject objects that are not within t of **q**. Each scene object includes a description of the smallest rectangle, aligned with the viewport axes, that includes the entire object (or its projection, in three dimensions). This rectangle includes the points $x_{lo} \leq x \leq x_{hi}$ and $y_{lo} \leq y \leq y_{hi}$, where x_{lo}, x_{hi}, y_{lo}, and y_{hi} are the bounds of the rectangle. If **q** is not within the (larger) rectangle $x_{lo} - t \leq x \leq x_{hi} + t$ and $y_{lo} - t \leq y \leq y_{hi} + t$, then the object can be rejected without further computation.

Lines-preferred MultiMap can be built on top of PointsInNeighborhood, CurvesInNeighborhood and FacesBehind in five steps:

Algorithm: Lines-Preferred MultiMap

Step 1. Perform CurvesInNeighborhood on all of the curves in the scene, where the argument r is set to 0. This produces an ordered list of curves.

Step 2. Perform PointsInNeighborhood on all points in the scene. (Do not compute intersection points.) This produces an ordered list of vertices.

Step 3. Merge the two lists maintaining distance order.

Step 4. If the resulting list is non-empty, we are done. Otherwise, call FacesBehind and return the nearest face.

Step 5. If no faces were returned in step 4, return a descriptor of empty space, the point **q**, and the default normal direction.

Points-preferred MultiMap can be built on top of PointsInNeighborhood, CurvesInNeighborhood and FacesBehind in five steps. Unlike lines-preferred gravity, points-preferred gravity does not set $r = 0$ in CurvesInNeighborhood. Also, for points-preferred gravity, the intersection points of gravity-active objects must be computed.

Algorithm: Points-Preferred MultiMap

Step 1. Perform CurvesInNeighborhood on all of the curves in the scene, where the argument r is set to the inner tolerance radius r of points-preferred gravity.

Step 2. Compute the pairwise intersections of all curves computed in Step 1 (if requested) and add the intersections to the set of gravity-active points. Find the midpoints of the curves computed in Step 1 (if requested) and add these to the

set of gravity-active points. Perform PointsInNeighborhood on this enlarged set of gravity-active points.

Step 3. If Step 2 returns any points within distance r of **q**, return these points. Otherwise, merge the curves from Step 1 with the points from Step 2, maintaining distance order and remove all points and curves that are more than s farther from **q** than the closest point or curve. Return the resulting collection of points and curves.

Step 4. If the resulting list is non-empty, we are done. Otherwise, call FacesBehind and return the nearest face.

Step 5. If no faces were returned in Step 4, return a descriptor of nothing, the point **q**, and the default normal direction.

The implementations of PointsInNeighborhood, CurvesInNeighborhood, and FacesBehind appear in Appendix A.1.

4.1.4 On-the-fly Intersections

Step 2 of the points-preferred MultiMap algorithm involves computing the pair-wise intersections of all of the curves computed by CurvesInNeighborhood in Step 1. For a reasonably small tolerance radius in a scene of moderate complexity, there will not be very many curves within tolerance, so the computation of all pair-wise intersections will not be expensive. However, a simple trick allows us to get good performance in a larger range of circumstances.

The trick relies on two observations. (1) Points-preferred gravity need only return the nearest object and any objects that are only slightly farther away than that. (2) The intersection point of two lines cannot be any closer to **q** than either of the lines. Hence, if the closest intersection point found so far is at a distance d from **q**, we can eliminate from further consideration any lines that are farther from **q** than $d+s$. Because curves are considered from closest to farthest, near intersection points tend to be found first, removing from consideration most of the curves. In fact, if we are careful to consider all near pairs before trying any farther pairs, as is accomplished by the looping structure

```
for i = 1 to N
    for j = 1 to i
        Consider the intersection of curve i with curve j
    endloop
endloop,
```

then it can be shown that, once t is larger than $d_{min} + s$, where d_{min} is the distance from **q** of the closest intersection point, additional increases in t do not increase the number of intersection points that are computed. See Appendix A for more details about on-the-fly intersections.

This same trick can be used to reduce the time spent considering vertices of shapes as well. There is no need to compute the distance of a vertex from **q** unless one or more of the segments that end at that vertex are within distance $d + s$ of **q**, where d takes on the value r at the beginning of points-preferred MultiMap. However, caution is required; when a segment is being rubber-banded one of its vertices may be stationary, and thus gravity-active, even though the segment is not. Such lonely vertices and all off-curve control points must still be compared to **q**, independent of what curves are near **q**.

Finally, the idea of keeping track of the closest object seen so far can be used to reduce the time spent considering the curves themselves. For instance, with lines-preferred MultiMap, once a curve at distance d has been found, curves farther from **q** than $d + s$ can be rejected immediately. See Appendix A for more details.

4.1.5 Three-Dimensional MultiMap

The gravity implementation in three dimensions is almost identical to the two-dimensional implementation. There are a few notable differences.

1) The skitter "position" is a complete coordinate frame. Hence, MultiMap must compute, for each object that projects near the cursor, the point on the surface of that object whose projection is nearest to the cursor and any surface normals and curve tangents at that point. This point and these vectors are used to construct a coordinate frame. Recall that to orient the caret in two dimensions it is adequate to return just a curve normal.

2) A third type of gravity, *faces-preferred* gravity, is available to the user. This gravity function snaps the skitter to a default plane when the cursor is not near any shapes.

3) Objects can intersect in curves as well as in points. The intersections that result in curves must be computed as soon as they exist, so that the curves can be drawn. Thus these intersections are not computed on the fly by the gravity algorithm.

These issues are discussed in this section.

MultiMap must return with each object, o, the point at which to place the skitter's origin if that object turns out to project closest to **q**. Ideally, this point would be that point of o that projects nearest to **q**. Computing this ideal point is difficult in general. Conceptually, it requires projecting vertices, edges, and faces onto the screen, computing the closest point from each projection to **q**, and projecting back to find the corresponding point in 3-space. Fortunately, an approximation will suffice. Gargoyle3D finds the closest three-dimensional point **p** on the object to the cursor ray, projects this point onto the screen and measures the screen distance of this projected point from the cursor. These screen distances are compared to determine the closest objects to the cursor.

This approximation works poorly for edges and faces that are viewed nearly "edge on". For instance, Figure 4-5 shows a two-dimensional analogy of the problem. A' and B' are the projections of points A and B respectively. While B' is closer to **q** than A', A is closer to the cursor ray than B, so MultiMap will incorrectly return A instead of B. However, if the user moves the cursor closer to the line segment, B becomes closer to the ray than A, and all is well. In fact, when the ray points directly at the segment, both the approximate distance and the screen distance are zero, so the approximation converges to the correct value as the cursor approaches a shape.

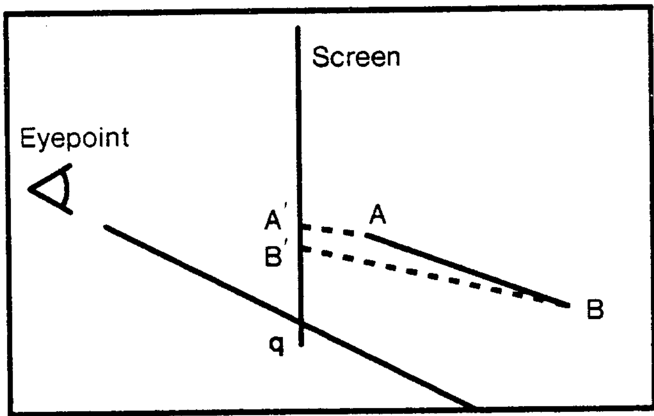


Figure 4-5. Computing distances in 3-space for shapes viewed "edge on."

The coordinate frame returned by MultiMap is orthogonal and its orientation is chosen so as to align its axes with some of the differential quantities at **p**, such as surface normals and curve tangents, where the quantities to use depend on whether the returned object is a surface, edge, curve, vertex, or intersection point. The orientation of the skitter is important, because it determines the orientation of the anchor, and the axes of the anchor are used as axes of rotation. Below, I describe, for each type of object that the skitter can snap to, how its orientation is computed.

For a surface, the z axis is the surface normal, and the x axis is horizontal (or parallel to the x axis of WORLD if horizontal does not determine it uniquely).

If **p** is on a curve, the x axis is parallel to the edge tangent. If this curve is the edge of a polyhedron, the z axis is chosen from the surface normal of one of the polygons that meet at that edge; Gargoyle3D chooses the polygon that more nearly faces the eyepoint. For a curve that is not the edge of a surface (e.g., an alignment line), the z axis is the curve normal (and thus in the osculating plane) on the convex side of the curve; if the curve is a straight line, the z axis is chosen to be horizontal (or parallel to the x axis of WORLD if horizontal does not determine it uniquely).

For **p** on a vertex, MultiMap chooses from the edges that end at that vertex, the one whose projection is nearest to **q** and computes the x and z axes as described above for **p** on an edge. For the intersection point of two curves, the z axis is the cross product of the tangent directions of the curves, and the x axis is the curve tangent of the curve that projects closer to **q**. For the intersection of a surface with a curve, the z axis is chosen from surface normal of the surface, ignoring the

curve, and the x axis is horizontal (or parallel to the x axis of WORLD if horizontal does not determine it uniquely).

In three dimensions, faces-preferred gravity is important for placing the skitter, both as a third type of gravity available to the user and as a basis for the other two types. Faces-preferred gravity uses ray casting to find the objects that are directly behind the cursor [Roth82]. Unlike the ray casting used for rendering applications, however, this ray-casting must find all of the objects that the ray hits, rather than just the one nearest to the eyepoint. Once we have a list of all of these objects, we can allow the user to cycle through the list to place the skitter on or select objects that are hidden behind other objects. If no objects are found, the ray is intersected with a default plane that is parallel to the screen. The skitter is placed on the resulting intersection point.

In three dimensions, intersection curves must be computed and drawn. Spheres produce circles when intersected with spheres or planes. Planes intersect planes in lines. Planes intersect polygons in line segments. These curves of intersection should be drawn for the user so the user knows what shapes are available to snap to. In two dimensions, intersections are only computed when the cursor gets near the curves that form the intersection. In three dimensions, intersections are treated in two cases. If the intersection results in a curve, it is computed whenever the scene or alignments change. The resulting intersection curves are placed, along with special sub-part descriptors, in the same data structure with regular alignment objects. The sub-part descriptor for an intersection curve describes whether it is a circle, line, or line segment, and describes the two objects whose intersection it is. If the intersection results in a point, it is computed on the fly as in two dimensions.

4.1.6 Performance of Gravity

The time needed to compute MultiMap is quite reasonable even in fairly large scenes. In this section, I present some of the running times for MultiMap on a small number of benchmarks. While elapsed times are not very useful for comparing algorithms, they do show when an algorithm has become feasible on available hardware. These times were computed by a gravity-testing routine, built into Gargoyle, that chooses screen points at random (using a pseudo-random number generator so the results are reproducible), draws the test points, and computes the average time to compute MultiMap.

To aid the reader in estimating how these algorithms would perform on other computers and operating systems, here is a rough characterization of the Cedar environment running on a Dorado. The production version of the Dorado has a 65 nano-second micro-cycle time and takes 4-5 micro instructions on the average to execute a Cedar macro instruction [Atkinson88]. In Cedar, the basic floating-point operations are implemented in micro-code. When the Dhrystone benchmark [Weicker84] is implemented in the Cedar language and run in the Cedar environment on a Dorado, a performance of 3310 dhrystones/second is achieved. For the VAX™ 11/780 running UNIX™ 4.2 BSD and programmed in C, published Dhrystone times [Richardson87] range from 1243 to 1441 dhrystones/second depending on the compiler. For the SUN 3/160 running UNIX SUN 3.0, times range from 2843 to 3236 dhrystones/second depending on the compiler. While it is always dangerous to compare systems on the basis of a particular benchmark, the Dhrystone benchmark places Cedar language programs running in the Cedar environment on a Dorado at between 2.3 and 2.7 times C programs running in UNIX 4.2 BSD on a VAX 11/780, and between 1.0 and 1.2 times programs running in UNIX SUN 3.0 on a SUN 3/160.

One obvious gravity test is to create a fairly dense rectangular grid and measure computation time for gravity on that grid. A grid can be created in Gargoyle by building an x axis and y axis out of short line segments, making all vertices hot and turning on 0 and 90 degree slope lines as shown in Figure 4-6. Test points are chosen at random as suggested by Figure 4-6. I used an 8 inch by 10 inch grid with 0.1 inch spacing (81 vertical lines, 101 horizontal lines, 8181 intersection points), testing MultiMap 500 times for each result, using test points uniformly distributed in the drawing area (roughly the same size as the grid). The default value of t in Gargoyle is 25 pixels (about 0.347 inches).

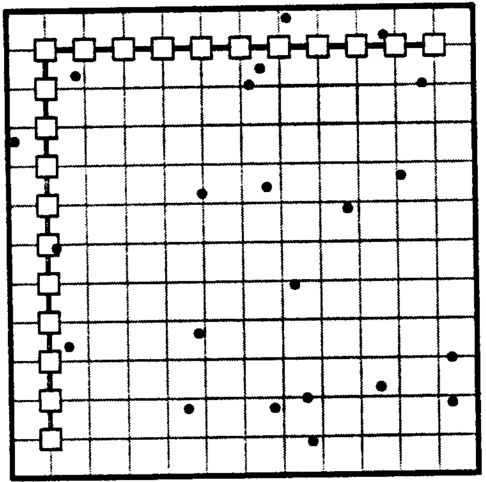


Figure 4-6. Gravity scatter plot on a grid of alignment lines.

Below, I list the times to compute MultiMap on this scene. Because this scene contains line segments (making up the grid axes) as well as slope lines, I also list, in parentheses, the average time to compute MultiMap with the axes alone. Notice that adding alignment lines sometimes *reduces* the time to compute gravity! This happens because alignment lines, which are inexpensive to examine, are processed before scene objects, which are relatively expensive. Thus, with the alignment lines present, $d_{min} + s$, the distance from the cursor beyond which objects can be trivially rejected, is quite small by the time the scene shapes are examined, reducing the number of scene shapes that need to be processed.

For points-preferred MultiMap:

1) With t=0.347 inches: 16 to 40 msec, 19 msec avg. (6 msec avg. with no lines.)

2) With t=11.11 inches: 40 to 171 msec, 113 msec avg. (183 msec avg. with no lines!)

For lines-preferred MultiMap:

1) With t=0.347 inches: 14 to 32 msec, 17 msec avg. (6 msec avg. with no lines.)

2) With t=11.11 inches: 51 to 115 msec, 84 msec avg. (123 msec avg. with no lines!)

Note that lines-preferred MultiMap does not compute intersections, but its time is nearly identical to points-preferred for t=0.347. In fact, only an average of 1.4 intersections per cursor position are computed in points-preferred MultiMap for t=0.347 in this example. The cost of intersections is negligible compared to the cost of finding the nearest curves to the cursor. The

tricks discussed in section 4.1.4 are important for this low cost. Before they were implemented, points-preferred MultiMap computed 87.3 intersections per cursor position on the average instead of 1.4 on this scene with t=0.347.

Also note that for t=11.11, all of the objects in the picture are within tolerance of all of the test points. In this case, the tricks discussed in section 4.1.4 become important. For instance, keeping track of the nearest curve seen so far reduced the average time for lines-preferred MultiMap from 212 msec to 84 msec.

Similarly, placing 8 line segments, each 1 inch long, end to end to make a row of segments, and placing 9 such rows at 1 inch spacings, we have a grid with 81 vertices. Making all vertices hot and activating circles of radius 1 inch, we have a regular array of 81 circles. Figure 4-7 shows a small part of such an array. The resulting circles intersect at 1088 points, are tangent to other circles at 102 points and tangent to the straight lines at 162 points.

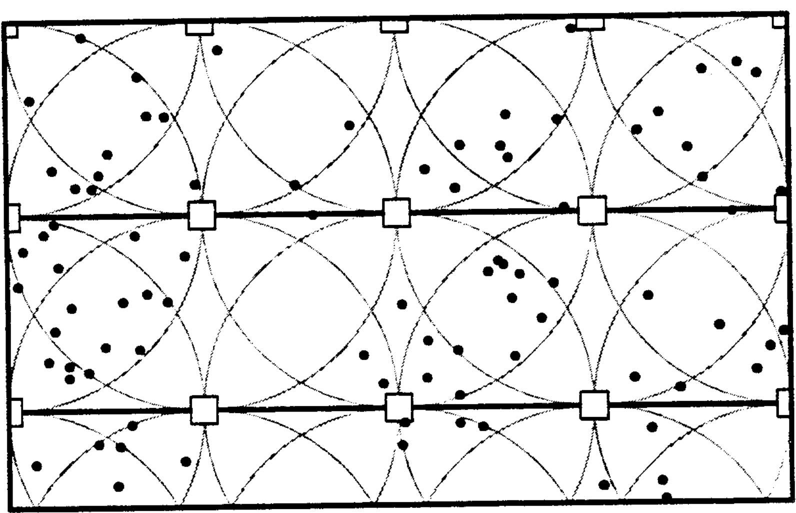


Figure 4-7. Gravity scatter plot on a grid of 81 circles.

Below, I list the times to compute MultiMap on this scene. As above, I also list, in parentheses, the average time to compute MultiMap with the line segments alone. Notice that adding alignment circles sometimes *reduces* the time to compute gravity, for the reasons described above.

For points-preferred MultiMap, the times were:

1) With t=0.347 inches: 14 to 40 msec, 21 msec avg. (8 msec avg. with no circles.)

2) With t=11.11 inches: 48 to 99 msec, 75 msec avg. (83 msec avg. with no circles!)

For lines-preferred MultiMap:

1) With t=0.347 inches: 13 to 24 msec, 18 msec avg. (8 msec avg. with no circles.)

2) With t=11.11 inches: 21 to 74 msec, 49 msec avg. (64 msec avg. with no circles!)

In this example, an average of 2.97 intersections are computed per cursor position.

These results indicate that gravity for alignment lines and alignment circles can be computed rapidly. Higher gravity times are observed in scenes with a preponderance of spline curves for which the cost of computing the nearest point of the curves to the cursor is higher. However, for most practical scenes, including those with splines, the cost of computing gravity is fast enough to allow the cursor to be updated several times per second. In a collection of benchmarks used to test Gargoyle performance, average times for points-preferred MultiMap range from 5 msec for a scene with 12 line segments and 24 alignment objects to 37 msec for a picture of a gargoyle, shown in Figure 4-8. All of these times are computed with t=0.347 inches.

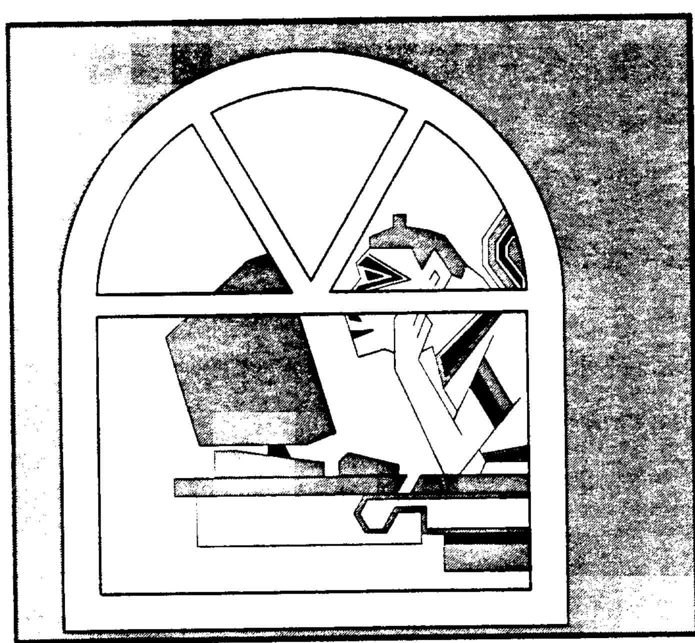


Figure 4-8. "The Gargoyle in the Window" Maureen Stone and Eric Bier, 1986.

The gravity algorithm described in this section is O(n) where n is the number of vertices and segments in the scene. Every scene object is considered each time the cursor position changes. While objects far from the cursor are quickly rejected by a bounding box test, this asymptotic

complexity can be a problem in very large scenes. However, hierarchy can be used to improve this bound. For many reasons, the user may cluster the scene objects into assemblies and sub-assemblies. Failing that, the system can cluster objects automatically into a user-transparent hierarchy based on object proximity. By performing bounding box tests at each level in the hierarchy, we can reduce search complexity to O(log n) in those case where the number of objects within t of **q** is small compared to n. So far, neither of these schemes has been implemented in the Gargoyle editors. It is also possible to use spatial sub-division schemes such as EXCELL [Tamminen81] or the "grid file" [Nievergelt84]. However, since scenes change frequently during snap-dragging, the overhead of keeping these structures up to date might well be prohibitive.

4.2 Alignment Objects and Gravity-Active Scene Objects

During a snap-dragging session, alignment objects and their intersections are constantly being computed and drawn as the user activates and deactivates alignment types, makes scene objects hot or cold, and moves objects or stops moving them. Likewise, the set of gravity-active scene objects changes when objects are added or deleted, and when objects move or stop moving. The system, then, must maintain the consistency of four entities: 1) the set of objects that are "triggering" alignment objects, 2) the set of gravity-active scene objects, 3) the set of alignment objects, and 4) the display.

In this section, I describe how this consistency can be maintained rapidly. I begin with an overview of the rules for computing the four entities from each other. Then, I describe the set of user operations that can change one or more of the entities. Next, I describe the data structures and algorithms used to implement the rules. Finally, I describe the special facilities needed for three-dimensional snap-dragging.

4.2.1 Trigger Bag, Scene Bag, Align Bag, and Display

In a static scene, the set of alignment objects that should be active can be computed from two simple Cartesian product rules. At each hot vertex in the scene, and at the anchor, construct one slope line for each slope that the user has requested, and one circle for each radius that the user has requested. This amounts to the Cartesian product: {triggering points} × {point alignment values}. At each hot edge in the scene, and at the x axis of the anchor, construct two angle lines

for each angle that the user has requested, and two distance lines for each distance that the user has requested. The anchor is treated like an edge with only one end. This amounts to the Cartesian product: {triggering edges} × {edge alignment values}. Each cartesian product can be computed with two nested loops, one iterating over a set of triggering objects, and one iterating over the associated set of alignment values.

Likewise, when the scene is static, the set of gravity-active scene objects is easy to compute: The anchor and scene objects are gravity active.

In a dynamic scene (i.e., while the user is rubber-banding objects, transforming entire objects, or both), the rules are more complicated. Vertices and edges that are moving are not allowed to trigger alignment objects, even if those vertices and edges are hot. This restriction simplifies the gravity algorithm, reduces screen clutter during interactive transformations, and speeds up screen refresh.

Computing the set of gravity-active scene objects is also more complicated. All object parts that are strictly stationary (neither moving nor rubber-banding) are gravity active. Other object parts are not. Like the rule for alignment objects, this rule ensures that the user does not have to worry about snapping the software cursor to a moving target.

The set of vertices, edges, and special objects that are currently eligible to trigger alignment objects is stored in a data structure called the *trigger bag*. The current set of alignment objects is stored in the *align bag*. The current set of gravity-active scene objects is stored in the *scene bag*. The sequence of steps required to compute these bags from scratch is depicted in Figures 4-9 and 4-10.

The trigger bag includes the anchor (if it has been placed), the scene object parts that the user has explicitly made hot, and the scene objects that are made hot by the automatic rule. These sets are unioned together, and all moving parts are removed to produce the trigger bag (Figure 4-9).

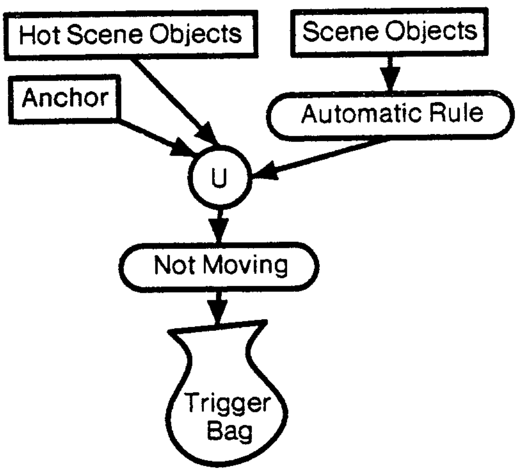


Figure 4-9. Computing the trigger bag.

The scene bag is computed simply by taking all of the objects in the scene and removing those parts that are moving, as depicted in Figure 4-10(a).

Finally, the align bag is built by combining two sets as shown in Figure 4-10(b). The first set results from taking the Cartesian product of trigger points with point alignment values combined with the Cartesian product of trigger edges with edge alignment values as described above.

The special operations function computes intersection curves in Gargoyle3D, and allows the basic alignment scheme to be extended in both Gargoyle editors. The scene bag is used as an input to this function, instead of the entire scene, because only stationary objects (and their intersections) are of interest. The special operations function can be used to construct alignment objects that are tailored to a particular operation. For example, during scaling, we may wish to compute the line that will be swept out by the caret if it maintains the current slope from it to the center of scaling. This would have simplified the construction of the largest square that fits in a hexagon (recall Figure 3-52). The "mode" information mentioned in the diagram is a description of the user interface operation ("Caret Placement", "Translate", and so forth) that is in progress.

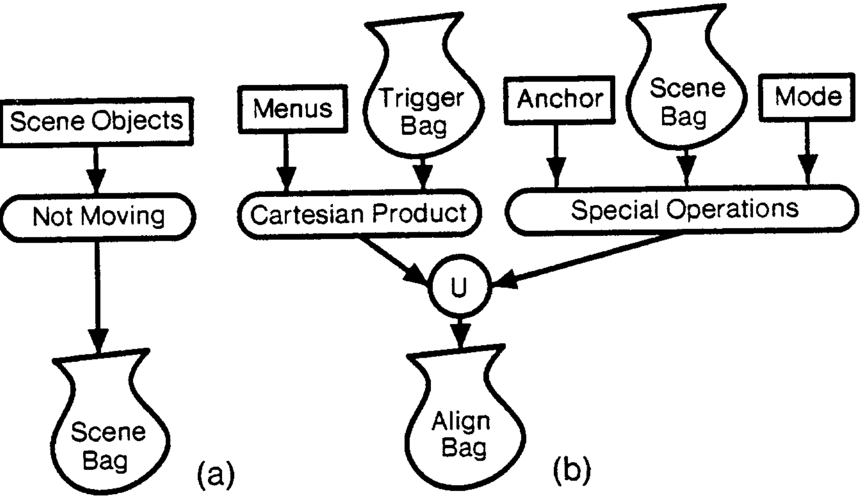


Figure 4-10. Computing the gravity-active bags. (a) Computing the scene bag. (b) Computing the align bag.

In early implementations of snap-dragging, the special operations function was used to compute the midpoints of scene edges and the intersection points of scene edges. However, the code became both simpler and faster when these computations were performed on the fly as part of the gravity computation, as described in section 4.1.4.

Several triggering points may produce the same alignment object. For instance, in Figure 4-11, the four vertices marked with an asterisk trigger the same slope line. All of these vertices are highlighted when the user snaps the caret to the slope line to give the user confidence that these vertices are indeed collinear. In order to produce this feedback, and to reduce the load on the gravity algorithms, duplicate alignment objects are detected and coalesced. In the example of Figure 4-11, only one alignment line is placed in the align bag to represent the geometric line that the caret is snapping to. However, the data structure representing this line contains pointers back to all of the objects that trigger it.

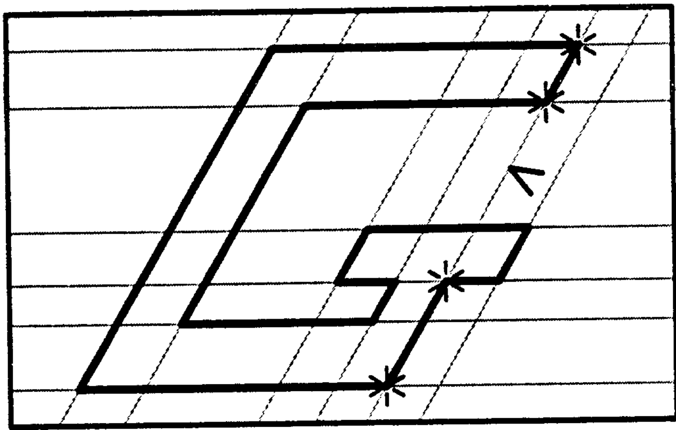


Figure 4·11. Several hot points that trigger the same alignment line.

Finally, once the set of alignment objects is computed, these objects must be drawn on the display. If we think of the display being composed back to front, the picture is built up in five layers: 1) the stationary scene objects, 2) the moving scene objects, 3) the selection feedback and attractor feedback, 4) the alignment objects, and 5) the caret and anchor. The alignment object layer contains renderings of lines and circles (or lines, ellipses, and parallelograms for three-dimensional snap-dragging) corresponding to each alignment object in the align bag. These lines and circles are drawn in a grey half-tone pattern and ORed into the display so they only partially obstruct the scene objects behind them. The other display layers are discussed in section 4.3.

4.2.2 Incrementally Updating the Bags and the Display

Above, I described the process of computing the alignment objects and gravity-active scene parts from scratch. During an editing session, however, small changes are made to the set of objects that are hot, to the scene, and to the set of selected alignment values. For performance reasons, it is important to be able to update the three bags and the display incrementally in response to these changes. There are eight cases to consider:

1) *More hot.* When vertices or edges become hot, they are added to the trigger bag, the new alignment objects that they trigger are added to the align bag and drawn on the display. The scene bag is unchanged.

2) *More cold.* When hot vertices or edges become cold, they are removed from the trigger bag, the alignment objects that they trigger are removed from the align bag

and cease to be drawn on the display. This requires some care because other vertices and edges may trigger the same alignment objects. In this case the alignment objects are not removed.

3) *Objects added.* When objects are added to the scene, they are added to the scene bag. If any parts of the new objects are hot, we must also perform step 1 on the hot parts.

4) *Objects deleted.* When objects are deleted from the scene, they are removed from the scene bag. If any parts of the deleted objects are hot, we must perform step 2 on the deleted hot parts.

5) *Alignments on.* When a new alignment value is activated, the Cartesian product of this value with triggering points or edges must be computed, the resulting alignment objects must be added to the align bag and drawn on the display.

6) *Alignments off.* When an alignment value is deactivated, all of the alignment objects of that value must be removed from the align bag and cease to be drawn on the display.

7) *Motion begins.* When objects begin to move, their moving parts are removed from the trigger bag and the scene bag. If the automatic rule is on, their stationary parts are added to the trigger bag. Any alignment objects that were triggered by the moving parts are removed from the align bag and cease to be drawn on the display. Any alignment objects triggered because of the automatic rule are added to the align bag and drawn on the display.

8) *Motion ends.* When objects stop moving, the process described in step 7 must be reversed. Parts that were moving are returned to the scene bag. Parts that were hot and moving are returned to the trigger bag. Parts that were placed into the trigger bag because of the automatic rule (and for no other reason) are removed. Alignment lines that were triggered by these automatically active parts are removed from the align bag and cease to be drawn on the display. Alignment lines triggered by hot points that were moving are computed at their new positions, placed in the trigger bag, and drawn on the display.

4.2.3 Data Structures and Algorithms

Efficient implementation of the incremental updates described in the last section was achieved by applying three familiar techniques:

1) caching – saving values that may be needed again,

2) hashing – accessing data by one of its properties,

3) case analysis and coherence – identifying those situations where one or more data structures do not change or are returning to a very recent state. In these cases, existing data structures are used instead of building new ones.

In this section, I describe the representations of the trigger bag, the scene bag, the align bag and the alignment object layer of the display and show how they are used in the eight incremental updates.

The trigger bag. The trigger bag is implemented in two parts. The first part is a pointer to the anchor, if it has been dropped. The second part is a hash table, where each bucket contains a linked list of descriptors. Each descriptor describes, for a single scene object, which of its parts are hot and not moving. The address of the object in virtual memory is used as the hash table key. When more parts of an object are made hot (manually or automatically), the hash table is searched for a descriptor for that object. If a descriptor is found, the descriptor for the new hot parts is unioned with the old descriptor and the result is stored in the hash table in place of the old descriptor. Otherwise, the new descriptor is stored. Furthermore, the difference of new and old descriptors is computed to discover which parts are newly hot (to aid in updating the align bag). Likewise, when parts become cold, the old descriptor is found, the difference of old and new descriptors is computed and the result is substituted for the old descriptor. The intersection of new and old descriptors is computed to discover which parts have become cold (again, to aid in updating the align bag).

Non-destructive list operations are used to maintain the linked list of descriptors that resides in each bucket of the trigger bag's hash table. Thus, the contents of the trigger bag can be copied simply by copying the address of each linked list from the bucket of the original hash table to the corresponding bucket in a new hash table. Non-destructive list operations ensure that changes to the original will not affect the copy and vice-versa.

The scene bag. The scene bag consists of a hash table much like the one used in the trigger bag. Each descriptor describes those parts of a scene object that are stationary. When an interactive transformation is about to begin, the hash table is searched for each object that is moving. If the entire object is moving, its descriptor is removed from the hash table. Otherwise, a descriptor describing the stationary parts is substituted for the existing one. The buckets of the scene bag are maintained using non-destructive list operations, so they can be copied in the same manner as described for the trigger bag.

The align bag. The align bag consists of four parts, the slope lines, the angle lines, the distance lines, and the circles. Each of these parts must support three operations: 1) addition of a new alignment object (with duplicate suppression), 2) deletion of an existing alignment object, and 3) enumeration of all of the alignment objects in that part. Each of the four parts is represented by a hash table, where the hash table key of each object is derived from its position in space. Each bucket of the hash table contains a linked list of the alignment objects that hash to that bucket.

For instance, the hash table for slope lines is implemented as follows. Two slope lines are considered to be "duplicates" if they are parallel (were generated by the same slope angle in the alignment menus) and are closer together than some value ε. Notice that the distance between two parallel lines is just the difference between their signed distances from the origin. In particular, if

$$c\,y - s\,x - d_1 = 0, \text{ and}$$

$$c\,y - s\,x - d_2 = 0$$

are the equations of two parallel lines, where c and s are the cosine and sine respectively of their common angle from horizontal, then the distance between the two lines is just $| d_1 - d_2 |$. If we use the signed distance d_i as the basis for a hash table key, we can arrange it so that duplicates fall in the same bucket or in closely related buckets.

In particular, let the hash table key for slope line i be

$$\text{key(i)} = \lfloor d_i / \varepsilon \rfloor,$$

where $\lfloor\ \rfloor$ denotes the largest integer smaller than the bracketed quantity. Then any duplicates of line i will be in the buckets whose keys are key(i)-1, key(i), or key(i)+1.

When a line, i, is added to the align bag, the hash table is searched for duplicates. If a duplicate, j, is found, the line is not added to the hash table. Instead, a pointer back to i's trigger is added to the record for j. Otherwise, i is added to the bucket whose key is key(i).

A line can be deleted from the align bag by describing its slope, its distance d_i from the origin, and the object that triggered it. The buckets key(i), key(i)+1, and key(i)-1 are searched, in that order, to find a reference to the line. Notice that this makes it unnecessary to maintain a list of pointers from scene objects to alignment objects. When a scene object becomes cold, the alignment lines that must be removed from the align bag can be computed from the Cartesian product of the points that have become cold with the active alignment values. These values can then be found in the hash table and removed using list operations.

The values in the hash table are enumerated by enumerating all of the hash buckets.

All of the slope lines, independent of slope are put into a single hash table. Thus, the code that searches for duplicates must check the slope of each line found in a hash bucket as well as its distance from the origin. At the expense of additional space, it would be possible to use a list of hash tables, one for each slope value. This scheme would both make additions faster and would make it trivial to turn off all of the slope lines of a given slope. It has not yet been implemented.

The other three parts of the align bag are implemented in a similar fashion. For distance lines and angle lines the hash key is computed from d_i in the same way as for slope lines. For circles, the distance (computed with the 1-norm) of the center of the circle from the origin of the reference coordinate system is used as d_i. Circles must be compared carefully because circles with the same distance from the origin may be far apart. The buckets of all four parts of the align bag are maintained using non-destructive list operations, so they can be copied in the same manner as described for the trigger bag and the scene bag.

The display data structures. Two forms of bitmap caching and a hash table are used to help update the alignment object layer of the display. The first cache contains scan-converted versions of each size of alignment circle. The second cache is a single screen-sized bitmap, the *alignments bitmap*, into which all alignment objects are drawn before being combined with the other display layers. The hash table keeps track of all of the alignment lines that are drawn into the alignments bitmap. This information can be used in some cases to support incremental erasure of lines.

Whenever an alignment circle is scan-converted for display, the resulting rasterized circle is

stored in a square bitmap that is just large enough to contain it. If the cache is full, previously rasterized circles are thrown out of the cache. The rasterized circle is then copied into the alignments bitmap, by ORing the bits. When all of the alignment objects have been drawn into the alignments bitmap, it is ORed onto the display. Since, in the common case, the user only activates a few sizes of circle, the cache rarely fills up. Thus, most circles can be drawn using hardware RasterOp operations (see Newman and Sproull's discussion of one bit per pixel RasterOp [Newman79]) instead of being scan-converted from scratch.

Alignment lines are drawn directly into the alignments bitmap. The alignments bitmap is flushed whenever the user scrolls (or whenever the viewing parameters change in three dimensions).

The display hash table is much like the hash table used for the slope lines part of the align bag. However, for this hash table, ε is set equal to half a pixel. When a slope line, angle line, or distance line is about to be drawn onto the alignments bitmap, the hash table is searched for an existing line of the same slope and nearly the same position. If such a line exists, the new line is not drawn. Instead a pointer to the line is added to the record for the existing duplicate line.

In some cases, incremental deletion can be performed on the alignments bitmap. In particular, if the only active alignment values are two slope values, and these slope values are more than a few degrees apart, then we can remove lines from the alignments bitmap (e.g., when some vertices become cold) by drawing *in white* those lines that are to be removed. This will degrade the image slightly at the points where the removed lines intersect the remaining lines. However, this degradation is tolerable. Care must be taken not to erase a line if a duplicate for that line remains. The hash table is used to see if such duplicates exist. This trick is only used for slope lines. When circles, distance lines, or angle lines are to be deleted from the alignments bitmap, the bitmap is erased and redrawn, using the circle cache to speed the drawing of circles, as described above.

Now, let's see how these data structures are used to implement the incremental updates. Notice that case analysis is particularly important when motion begins or ends.

More hot. Incremental additions are made to the trigger bag, the align bag, and the display.

More cold. If a majority of objects are made cold, the trigger bag and the align bag are made from scratch and the display is redrawn (since this is faster than making incremental deletions). Otherwise, incremental deletions are performed on the trigger bag and the align bag.

Furthermore, if the align bag contains only slope lines (of reasonably different slopes), the display is updated incrementally. Otherwise, the display is redrawn.

Objects added. Incremental additions are made to the scene bag. If any of the new object parts are hot, updates are made as for "More hot".

Objects removed. If a majority of objects are deleted, the scene bag is made from scratch (since this is faster than making incremental deletions). Otherwise, incremental deletions are performed on the scene bag. If any removed parts were hot, updates are made as for "More cold".

Alignments on. When an alignment value is activated, make incremental additions to the align bag and the display.

Alignments off. If the value that is being turned off is the last active value (or if all values are turned off at once), remake the align bag from scratch (since this is faster than making incremental deletions). Otherwise, delete the obsolete alignment objects from the align bag. If more alignment values are being deleted than remain, or if alignment objects other than slope lines are present, the display is made from scratch. Otherwise, the display is updated incrementally.

Motion begins. Make a copy of the current scene bag for use in "Motion ends" (see below). If more parts are moving than stationary, remake the scene bag from scratch (since this is faster than making incremental deletions). Otherwise, incrementally delete moving parts from the scene bag. Remove moving hot parts from the trigger bag. Make a copy of the resulting trigger bag for use in "Motion ends". Add to the trigger bag those object parts that become hot because of the "automatic rule". Remove alignment objects that were triggered by hot parts that are now moving from the align bag. Make a copy of the resulting align bag for use in "Motion ends". Add to the align bag those alignment objects generated by the "automatic rule". If alignment objects were removed from the align bag (because hot parts are moving) update the display without these lines, just as for "More cold". Make a copy of the resulting alignments bitmap for use in "Motion ends". If any alignment objects were added to the align bag, draw these alignments into the alignments bitmap.

When the automatic rule is off and no hot parts are moving, the trigger bag, align bag, and alignments bitmap need not be copied. The original versions can be used in "Motion ends".

Motion ends. Return to the saved versions of the scene bag, trigger bag, align bag, and alignments bitmap. If some hot parts were moving, compute the new alignment objects triggered by these parts, add these alignment objects to the align bag, and draw them into the alignments bitmap.

4.2.4 Alignment Objects in Three Dimensions

The computation of alignment objects in three dimensions is identical to the computation in two dimensions except that intersection curves must be computed and placed in the align bag along with the alignment objects. This is done at the "special operations" stage that was described in section 4.2.1.

When the intersection curve of two objects is computed, the result is packaged up into a data record along with pointers to the two objects. Later on, when the gravity mapping is performed, if the nearest object to the cursor ray is an intersection curve, Gargoyle3D can highlight the two objects that intersect, as well as the intersection curve, giving the user more confidence that the skitter is snapping to the correct object.

4.3 Interactive Transformations

During interactive transformation operations and during the "Caret Placement" operation, all of the machinery of snap-dragging is working together. The software cursor is snapping under the influence of gravity to scene objects and alignment objects, the automatic rule, if enabled, is adding extra alignment objects to the scene, and the selected collection of scene objects is moving and rubber-banding. From an implementation standpoint, these interactive operations have three stages: set-up, interaction, and completion. The following descriptions apply to Gargoyle and Gargoyle3D.

4.3.1 Set-up

When the user invokes an interactive operation, a number of data structures must be updated from their static state to their dynamic state. If the user is invoking a transformation, then some objects are about to move. The bags must be updated as described in section 4.2. We make note of the initial position of the anchor, call it *anchorPoint*, the initial x axis of the anchor, call it

anchorX, and the initial position of the caret, call it *startPoint*, since all transformations computed during dragging are relative to these initial values.

If the user is starting a "Caret Placement" or "Skitter Placement" operation, no objects are about to move, so the bags can be left in their static state.

4.3.2 Interaction

Each time the cursor moves by a pixel or more, the current gravity function is used to map the cursor to a software cursor position, *mapPoint*, and a closest object, O.

For transformations, we compute a new transformation based on mapPoint. The transformation that is computed depends on whether we are translating, rotating, scaling, or skewing.

For translation, we translate by the vector (mapPoint - startPoint).

For rotation in two dimensions, we rotate about anchorPoint by the angle between the vectors (mapPoint - anchorPoint) and (startPoint - anchorPoint).

For rotation about a point in three dimensions, we rotate about the line that passes through anchorPoint and whose direction is the result of the cross product (mapPoint - anchorPoint) $\times$ (startPoint - anchorPoint), by the angle between (mapPoint - anchorPoint) and (startPoint - anchorPoint).

For rotation about the x axis of the anchor in three dimensions, we project startPoint and mapPoint onto the y-z plane of the anchor, to get points startPoint$'$ and mapPoint$'$ respectively. We rotate about the x axis of the anchor by the angle between (mapPoint$'$ - anchorPoint) and (startPoint$'$ - anchorPoint).

For scaling, we scale about anchorPoint by the ratio $\|$mapPoint - anchorPoint$\|_2$ / $\|$startPoint - anchorPoint$\|_2$, where $\|\cdot\|_2$ denotes Euclidean distance.

For skewing in two dimensions, we use the unique affine transformation that takes the triangle [anchorPoint, anchorPoint+anchorX, startPoint] into the triangle [anchorPoint, anchorPoint+anchorX, mapPoint].

Snap-dragging can be extended to perform other affine transformations without modification so long as the new transformation depends only on startPoint, anchorPoint, anchorX, and mapPoint.

Finally, we apply the transformation to all selected objects and display them in their new positions. An outline of the refresh algorithm appears below.

4.3.3 Completion

We update the scene database to reflect the new positions of the selected objects, and we update the screen to show these objects in their new positions.

Since the selected objects are no longer moving, we restore the bags, as described in section 4.2. Returning to the old state of the bags makes sense because the bags simply contain *pointers* to the scene objects. Interactive transformations change object positions but do not add or delete objects and do not change which objects are hot or cold.

4.3.4 Updating the Screen

Snap-dragging can be used with a variety of refresh strategies including those based on XORing lines, ORing bitmaps, draw white/draw black and so forth. Gargoyle and Gargoyle3D compose each frame in several bitmaps, combine the bitmaps into a single bitmap, called the *chunking bitmap*, and send the chunking bitmap to the screen in a single step to achieve smooth motion.

During set-up, the stationary objects are drawn, in back to front order, into a bitmap called the *background bitmap*, and the alignment objects are drawn into the alignments bitmap. During interaction, the chunking bitmap is repeatedly composed as follows: The background bitmap is drawn into (completely overwrites) the chunking bitmap. The moving objects, including the stationary parts of objects that are rubber-banding are drawn next, directly into the chunking bitmap. Next, any attractor feedback, such as control points and asterisks, is drawn directly into the chunking bitmap. The alignments bitmap is ORed into the chunking bitmap. Finally, the caret and anchor are drawn directly into the chunking bitmap. The chunking bitmap is then drawn into (overwrites) the drawing area of the screen. The user sees a succession of completed frames, resulting in a perception of smooth motion.

To improve performance, each of the bitmaps is constructed incrementally. For instance, in the set-up phase of an interactive transformation, the background bitmap is constructed from the static background bitmap by redrawing only those stationary objects that overlap the bounding

rectangle of the objects that are about to move. Likewise, in the completion phase, the new background bitmap (consisting of all objects since they are all stationary) is constructed from the dynamic background bitmap by updating the bits within the rectangle containing the newly repositioned objects.

5. Productive Scene Composition

In ev'ry job that must be done
There is an element of fun;
You find the fun and snap!
The job's a game;

– Richard M. Sherman and Robert B. Sherman
"A Spoonful Of Sugar"

5.1 The Elements of Productivity

Snap-dragging addresses the problem of composing geometric scenes rapidly and precisely. Speed and precision are both essential to the problem at hand. If speed were unimportant, we might as well describe precise shapes by typing coordinates in a suitable geometric design language. If precision were unimportant, any fast interactive drawing program or painting program would do. If an interactive design technique does not provide both, then either deadlines will slip or designers will settle for less precision than they had hoped for.

Roughly, we can quantify the productivity of the user of a scene composition technique by measuring the number of scene points that are placed at their final precise positions per unit of time. A good scene composition technique will improve the productivity of the user. Productivity can be improved by using a combination of three strategies:

(I) Reduce the number of operations that are needed to place each point.

(II) Reduce the amount of time it takes for the user to invoke each operation.

(III) Reuse old work.

The number of operations can be reduced in several ways including:

(I.1) Provide operations that are powerful and natural to the task at hand so that fewer operations are needed to place each point. Very powerful operations may even place several points at once.

(I.2) Factor out commonly-used operations by providing modes. For instance, if the user is repeatedly invoking a command that forces the most recently added line to be horizontal, provide a mode that constrains all lines to be horizontal to begin with. This technique takes advantage of coherence in the editing session; while

something is being done repeatedly, do it automatically.

(I.3) Reduce manipulation of peripheral state. All drawing systems store some information that is not an essential part of the geometric model being produced. For instance, in a system where the final goal is to print a picture on paper, hierarchical structure and object names are not essential to the final result; two drawing systems could store different peripheral information and still produce identical printed pictures. Reducing the number of operation invocations spent modifying peripheral state (e.g., operations that change or display names) may reduce the total operation count for making an illustration.

Similarly, the amount of time per operation can be reduced in several ways including:

(II.1) Reduce the number of keystrokes that the user must type by taking advantage of the display and mouse. For instance, pop-up menus allow the user to invoke a command with a single keystroke instead of typing its name. Likewise, graphical entities can be selected by pointing at them instead of naming them. The idea is to achieve a good match between the computer, which can draw and type rapidly but cannot see or hear the user, and the user who can interpret pictures quickly but is slow at pressing keys.

(II.2) Reduce the time for mouse motion. Pop-up menus take time to use, particularly if several menu options must be chosen to invoke each operation. If a special key combination is provided for important commands (e.g., CTRL key combinations), these commands can be invoked even more rapidly than with a menu.

(II.3) Reduce the time that the user spends thinking. At the syntactic level, this can be done by designing the user interface carefully—choosing suggestive command names, providing good on-line documentation, and so on. At the semantic level, this can be done by providing operations that are easy to understand, are tuned to the task at hand, or both.

(II.4) Keep the system small. Reducing the total number of operations in the system will reduce the amount of time that the user spends searching for the desired operation. Also, if there are only a few operations, each of them can be invoked from the keyboard (strategy II.2).

(II.5) Reduce clutter on the screen. This will reduce the time that the user spends looking on the screen for relevant information.

Old work can be more easily reused if one or more of these strategies are employed:

(III.1) Reduce archeology. Make it easy for a user to understand geometry created by other users.

(III.2) Support transformation-invariant editing. Make it easy to translate, rotate, scale and skew a collection of scene objects and then continue to edit them at their new position, orientation, size and slant. If this is easy, the designer is encouraged to make a new shape from old shapes instead of starting from scratch.

Unfortunately, the three strategies for increasing productivity conflict with one another. Attempts to reduce the number of operations can increase the time per operation or the difficulty of sharing geometry with other users, and attempts to reduce the time per operation can result in operation proliferation. Furthermore, the different sub-strategies of a single strategy conflict. For instance, some methods for reducing the number of operations needed per task actually increase that number because of unconsidered factors. Here are some examples of conflicts:

1) *Powerful operations vs. thinking time.* Using powerful operations may increase the time per operation. For instance, operations that modify many points at once are often harder to learn, more time-consuming to use, and require more thinking time than operations that modify only a few points.

2) *Factoring vs. thinking time.* When modes are used to factor out common operations, the user requires extra thinking time to keep track of the modes that are currently active. Furthermore, failure to keep track of modes can result in errors that will take time to correct.

3) *Powerful operations vs. reducing peripheral state.* Some of the operations that do not directly contribute to the illustration build up state that powerful operations rely on. For instance, in a constraint-based system, the user must perform operations that create a constraint network before the constraint solver can operate. These operations cannot be eliminated even though they do not directly contribute to the illustration.

4) *Reducing screen clutter vs. pointing and reducing peripheral state.* Reducing screen clutter is difficult. If less system state is displayed (e.g., if the current constraints are not shown in a constraint-based system), then the user will have to spend time asking for this information when it is needed. Furthermore, if the pointing device is used to reduce typing as described in strategy II.1, objects must be displayed on the screen so the user can point to them.

Furthermore, some strategies don't apply in all systems. For instance, special key combinations can be provided to allow accelerated invocation of a few commands. However, the idea does not scale. The number of commands that can be accelerated is limited by the number of keys on the keyboard and by the user's ability to remember the key combination that invokes a given command.

Our goal of increasing the productivity of geometric designers can be thought of in economic terms. The total time needed to produce an illustration is equal to the number of operations, n, times the average time per operation, t. The curve $n\ t = k$, for some constant k gives all combinations of n and t that produce an illustration in the same total time. Figure 5-1 shows the curve $n\ t =$ k, normalized for $k=1$. Our strategies I and II correspond to moving the curve in directions **u** and **v**, respectively.

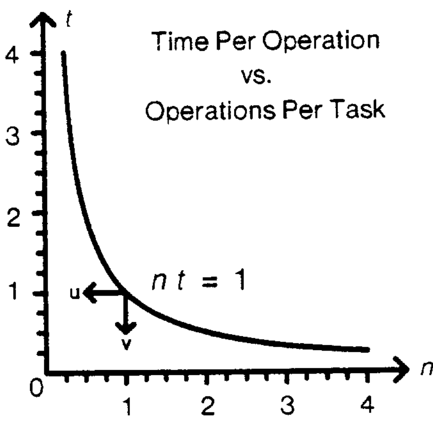


Figure 5-1. A curve of constant total task time.

If we perform a given task with a reference geometric design system, and normalize the total time to do the task to be 1, then we can compare other systems against the reference system on the graph shown in Figure 5-1.

In addition to the trade-off between operation count and operation time, the designer of a scene composition technique must also consider the time it will take a user to become proficient in using a particular technique. A highly-factored system with many accelerators, and many powerful commands may be worthless to an inexperienced user. Likewise, a simple system, with a few very intuitive commands will frustrate a more sophisticated user.

As was argued in section 3.1.5, the snap-dragging user interface possesses 14 commands and 4 alignment menus over and above the commands that would be present in most illustration systems. While more operations are provided for precision in snap-dragging than the 2 or 3 commands needed to turn the grid on and off and change the grid spacing in a grid system, this complexity appears to be within reach of a large community of users. See Chapter 6 for more information on the experiences of users with snap-dragging in its current form.

5.2 Comparison of Several Techniques

Different applications impose a variety of requirements on an interactive geometric design system. Some geometric designers want a system that is easy to use and is capable of producing box-and-pointer diagrams or laying out simple forms. A grid-based system is probably appropriate for this group of users. Animators, simulators, and mechanical engineers need to construct models with behavior. This modelling process includes deciding which parts of the model are fixed, which parts can vary, and what types of variability are allowed. These users want to represent these decisions either in a procedural language or in a declarative form such as a network of constraints. Finally, some geometric designers want to make technical illustrations, graphics for advertisements, thesis drawings, floor plans, mechanical schematics, or animation key frames. They need to be able to make these illustrations quickly and precisely. A technique based on drafting (i.e., on a ruler and compass paradigm) is appropriate for these users.

In this section, I compare grids, constraints, and snap-dragging, considering how effectively each of these techniques employs the productivity strategies described in section 5.1. This comparison is summarized in Table 5-1. Shaded entries in the table highlight areas of difficulty. Boxes with bold text and blackened corners indicate areas of special strength.

		Grids	Constraints	Snap-Dragging
Fewer Ops.	I.1 Use Powerful Operations	No.	Yes. (but much set-up)	Moderate.
	I.2 Factor Common Operations	Yes. Grid on.	In some systems.	Extensively.
	I.3 Reduce Manipulation of Peripheral State	Few. State is visible.	No. Build Constraints & lots of state.	Partly.
Less Time	II.1 Use Pointing	Yes, in 2D	In some systems.	Extensively.
	II.2 Reduce Thinking Time: (Planning & Debugging)	Potential problem.	No. Build constraints & debug constraints.	Yes.
	II.3 Keep the System Small	Yes.	In some systems.	Moderately.
	II.4 Reduce Screen Clutter	Yes, in 2D	In some systems.	Partly.
Re-use	III.1 Reduce Archeology	Yes.	No.	Yes.
	III.2 Support Transformation-Invariant Editing	No.	In some systems.	Yes.

Table 5-1. Summary of the productivity strategies for grids, constraints, and snap-dragging.

For the most part, the strategies that are successfully used by grid systems are also successful for snap-dragging. This similarity between grids and snap-dragging is not surprising. Snap-dragging can be seen as a generalization of the grid approach. Alignment lines provide a sort of tailorable grid. Furthermore, a grid can be added to a snap-dragging system without any modifications to the basic snap-dragging paradigm. I include grids in this discussion because many systems use them as the only mechanism for precise scene composition, and because their simplicity makes them attractive for a large group of users.

5.2.1 Reduce Operation Count

Use powerful operations. The three techniques differ greatly in the power and naturalness of their operations. Grids work well for pictures composed mainly of horizontal and vertical lines but are awkward in general. Many precise configurations of points cannot be achieved by placing all of the points on a regular rectangular grid. For instance, equilateral triangles and some points of intersection are impossible to construct, as shown in Figure 5-2.

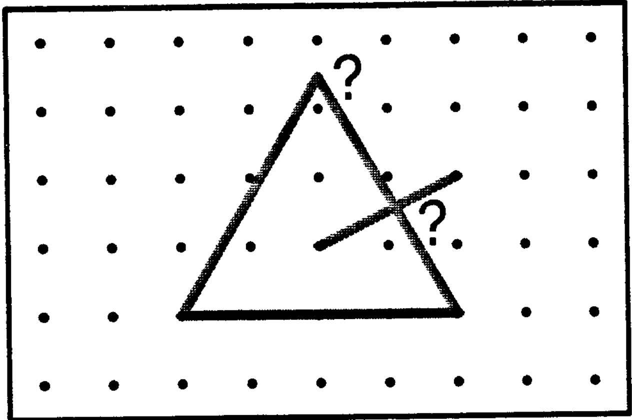


Figure 5-2. Some constructions that do not work out evenly on a regular grid.

Constraint systems and snap-dragging are similar in the relationships that they express directly. These include constraints on the distances and angles between points, lines, and planes. They differ in the way these relationships are used. In a constraint system, the user specifies those relationships that the model must satisfy and those degrees of freedom that can be changed. The user invests time in making these decisions and in building up the constraint network. The payoff comes when the designer uses the constraint solver to modify many point positions simultaneously by varying one or more of the degrees of freedom. In snap-dragging, alignment lines are used to place points in precise relationships. In a sense, the user specifies what constraints to apply to a new point by snapping the point to a particular surface, curve, or point. Because drawing and constraining happen simultaneously, few keystrokes are needed to assemble an illustration. However, because these constraints are not remembered, there is no special support for making global changes later on.

It is interesting to consider that when a designer decides to construct a true three-dimensional model instead of a single projection, he is opting to make an investment similar to the investment made in constraint-based systems. The three-dimensional model will probably take more time to construct, but once it is done, a number of powerful operations are available, including changing the viewing parameters, changing lighting, automatically dashing hidden lines, and so forth, that would be difficult to achieve with a projection. Because snap-dragging is so similar in two and three dimensions, it may reduce the effort to switch between two- and three-dimensional editing, encouraging the designer to use whichever medium is most effective for the task at hand.

One way to describe the power of constraint systems is to note that, instead of defining a

single shape, a constraint network defines a family of shapes, where different members of the family are chosen by varying one or more constraints. The affine transformations in snap-dragging also have this flavor. For instance, during a rotation, the user views a family of objects that are related to each other by a rotation of the selected parts. Constraint systems are more powerful in that a more general set of shape families can be defined. In this view, an affine transformation temporarily imposes a constraint network on the entire scene such that all unselected objects remain fixed and all selected objects translate, rotate, scale, or skew as a unit.

Factor common operations. One can imagine a system with a grid in which the user issues a command when he wants to snap the cursor to the nearest grid point. This operation would be so common that it would be worth introducing a mode such that the cursor point always snaps to the grid. Of course, this is exactly what is done in all grid-based drawing systems. This is an example of factoring common operations. Some constraint systems use this technique as well. In Juno [Nelson85], the user can constrain many objects to be horizontal by activating the horizontal constraint mode and pointing at all of the objects that are to be made horizontal. Factoring is also used extensively in snap-dragging. Rather than requesting the construction of alignment objects one at a time, mimicking the steps in traditional compass and straightedge construction, the user activates entire classes of alignment objects such as lines sloping at 0 and 90 degrees. The system will construct alignment objects of this class at all triggering vertices until the user deactivates the alignment class.

Reduce manipulation of peripheral state. Constraint systems store an unusual form of peripheral state, namely a network of constraints. The operations used to build up this state must be seen as an investment; with the constraints in place, the user will be able to make global changes at a future time. However, if the future changes are not extensive, the constraint-building operations may be a bad investment. Grids and snap-dragging involve a smaller investment. Turning a grid on and off or turning a type of alignment object on and off require fewer operations, in general, than applying constraints.

For instance, in Figure 6 of Nelson's paper on Juno, a block letter A is constructed, similar to the one shown in Figure 5-3. The A has 10 vertices with 2 coordinates each, making 20 degrees of freedom. In Nelson's example, 20 constraints are provided, not counting the constraints on some extra points that are added so that the height, width, and stroke width of the A can be easily

changed. 22 keystrokes are used to place the 10 vertices, and 64 keystrokes are used to constrain these 10 points (not counting the "freeze" constraints placed on the extra points). In Gargoyle, one can construct the A in 25 keystrokes: Turn on 0, 60, and 120 degree slope lines, 1/2 inch circles and 1 inch circles (5 keystrokes), turn on gravity and the automatic hotness rule (2 keystrokes), starting at point S, sketch using the order shown in Figure 5-3(a) (11 keystrokes), select and delete segments 3 and 6 (3 keystrokes), select and close the inner and outer contour (4 keystrokes) for 25 keystrokes total. The segments 1, 3, 6, and 8 are congruent (1/2 inch long), and the segments 2, 4, 5, and 7 are congruent (1 inch long) as in the Juno example.

A precise comparison of keystroke counts is difficult. For instance, by using keyboard combinations, Juno could place the original 10 points in 12 keystrokes, as Gargoyle does, instead of 22. However, even taking this into account, the total minimum number of keystrokes needed to make a picture with Juno is larger than the total needed with Gargoyle. Most of this difference results from the need to specify peripheral state – the constraints. Of course, not all users will be able to build a shape in the minimum number of keystrokes in either system. The user may have trouble converting a mental image of a shape into a sequence of construction operations or into a set of constraints, resulting in mistakes or greater thinking time per keystroke. Reducing the time per operation is discussed below.

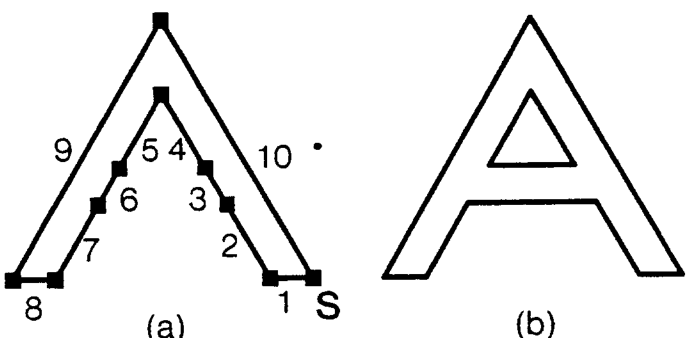


Figure 5-3. A block letter A.

A second cost of peripheral state is the need to make it visible to the user. It is difficult for the user to remember all of the constraints that are currently active. Sketchpad provides graphical representations for the constraints, which are displayed at the user's request. Juno represents the constraints textually. In either case, the user needs to perform operations requesting that the constraints be displayed. In two dimensions, grids and snap-dragging require no special viewing

operations; grids and alignment objects both have natural graphical representations that are sparse enough that they usually can be left on the screen during a session. In three dimensions, grids are more difficult to display without clutter, as suggested in Figure 5-4, making grids much less feasible for solid modeling. Because the user has control over how many alignment objects are displayed, alignment objects continue to be feasible in three dimensions.

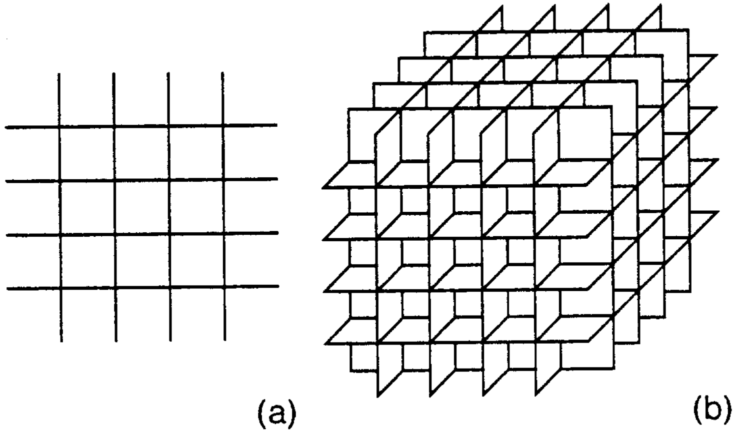


Figure 5-4. Visual representations of grids. (a) In two dimensions. (b) In three dimensions.

5.2.2 Reduce Time Per Operation

Use pointing. Grid systems allow the user to place the cursor by pointing instead of by typing coordinates. With some constraint systems, the user applies constraints to a scene by pointing at the name of a constraint and pointing at the vertices to be constrained. Snap-dragging makes particularly good use of pointing to reduce keystrokes. When the user snaps the cursor to the intersection of three alignment spheres during an "Add Line Segment" operation in Gargoyle3D, he performs four scene composition steps at once: a line segment is added to the scene, and the distance of that line segment from three points in the scene is specified. This is similar to adding a point and three constraints in a constraint-based system, except that the constraint is forgotten as soon as the operation is completed.

In a three-dimensional grid system, it would be hard to point to a desired grid point; there are too many of them. The trick to using pointing successfully is to give the user many choices so that the user can express a lot of information with each keystroke, but not so many choices that he becomes uncertain of which choice he has made. Snap-dragging, by allowing the user to customize the grid, puts this trade-off in the hands of the user.

Reduce thinking time. Constructing a geometric model using an interactive computer program can be thought of as a translation problem. The user takes a geometric idea, be it from paper or from the user's imagination, and communicates it to the computer via a sequence of commands that add new geometry and modify existing geometry. If the geometric idea cannot be naturally expressed in terms of these commands, the user will require extra time for planning and starting over. For instance, in Figure 5-5(a), the user is trying to use a grid to draw an arrow that begins at the midpoint of an edge. Unfortunately, the edge is an odd number of grid units long, so there is no grid point at the midpoint. If the user had planned ahead, he might have constructed the rectangle to be an even number of grid units long, as shown in Figure 5-5(b).

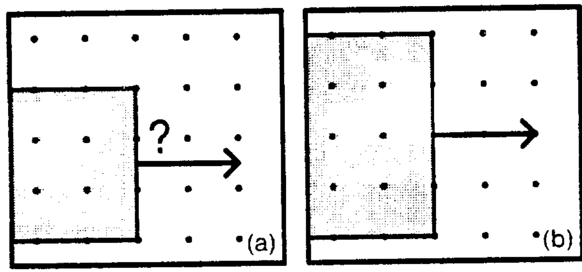


Figure 5-5. Adding an arrow at the midpoint of an edge. (a) The edge is an odd number of grid units high. (b) The edge is an even number of grid units high.

With constraint-based systems, the user describes precise shape relationships in terms of a set of geometric invariants. The user must spend time choosing enough independent constraints to completely constrain the shape and time to make sure that all of these constraints have been applied. Expressing geometry in terms of a particular set of constraints is not always easy. For instance, in Juno, to make two line segments be collinear, the user must learn to apply two parallel constraints, one to make the line segments parallel, and a second to make an imaginary line between the two segments be parallel to one of the segments. This construction might not be obvious to a beginning user. In addition, when the constraint solver reaches an unexpected solution, it takes time to figure out how to modify the constraint set to repair the problem. Debugging geometric constraints amounts to debugging a declarative computer program.

In snap-dragging, the user translates a geometric idea into a sequence of construction operations. Not all constructions will be obvious to the user. For instance, one way to make two segments be parallel is to measure the slope of one and use alignment objects to construct the other at the same slope. This construction might not be obvious to a beginning user. Users can

expect to learn construction tricks and facts about geometry as they become more proficient in using the system. However, since each change to the picture is made by translating, rotating, or scaling parts interactively, when something goes wrong, the user is not generally faced with a difficult debugging task. A human factors experiment would be useful to compare how long it takes to learn to use grid, constraint, or snap-dragging systems and to compare how much thinking time goes into constructions, and how many mistakes are made in each case.

Keep the system small. While scene composition is only a small part of what a complete illustrator or solid modeler must do, reducing the set of commands devoted to scene composition can make a noticeable difference in the size of an entire system. Grids can be controlled with a particularly small user interface. For instance, in the Gremlin picture editor at UC Berkeley [Opperman84], the user can turn the grid on and off by clicking the mouse over a special grid icon (graphically-labelled button), and can make the grid finer or coarser, by factors of two, using either the left or middle mouse buttons with the mouse over a second icon. Icons are also provided to invoke translation, scaling, rotation, horizontal reflection, vertical reflection, and point-moving operations. Constraint-based editors can have a very simple user interface for scene composition as well. Six of the icons displayed by the Juno editor have scene composition functions. They put Juno into modes that cause the mouse to apply a particular constraint.

User interfaces for ruler and compass systems can grow quickly. If there are operations for all of the standard constructions such as constructing the line through two points, constructing the line tangent to a circle at a point, and so on, the system quickly has enough operations that the user can spend noticeable time searching for the right one. In such systems, it is important to organize the operations into sensible categories to reduce this search time.

As described in section 3.1.5, snap-dragging relies on 23 commands, of which 14 are unique to snap-dragging, plus a set of menus that are always displayed (there is no delay waiting for them to pop up). The commands can all be invoked from the keyboard and mouse, except "Gravity Extent" and "Midpoints", which are available from buttons that are always displayed. Numbers in the alignment menus are kept sorted to reduce search time.

Reduce screen clutter. As shown in Figure 5-4, screen clutter makes grids less attractive in three dimensions than in two. Constraint systems can also cause the user to spend time searching the screen for important information, whether the constraints are shown textually or graphically.

In Juno and MathPAK, where the constraints are shown textually, the user must glance back and forth between the text and the graphical display to make the correspondences between textual symbols and geometric parts. In Sketchpad, each constraint is displayed using a graphical symbol showing which scene vertices participate in the constraint. These constraint symbols can overlap with the picture and with each other. They work reasonably well on simple objects such as a rectangle in two dimensions (8 degrees of freedom), but would probably be unmanageable in three dimensions even with a shape as simple as a cube (24 degrees of freedom).

The screen clutter resulting from the snap-dragging alignment objects is reasonably small and is under user control. When there are many hot points and a few active slope and radius values, the screen is still easy to understand, because all of slope lines with a given slope are parallel, and all of the circles of a given radius are the same size. Figure 5-6 shows a moment in the construction of the block letter A of Figure 5-3. The alignment objects are drawn with stipple patterns, as they are on the Gargoyle screen. Even though 3 kinds of slope lines, and 2 kinds of circles are active, it is easy to see that the caret is correctly placing the lower left point of the A.

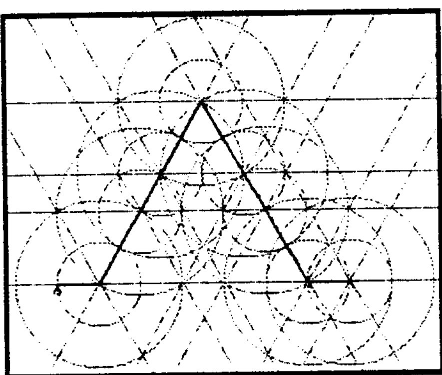


Figure 5-6. During a construction with 0, 60, and 120 degree slope lines and 0.5 inch circles. Taken directly from a bi-level display. Shown 1/2 actual size.

With snap-dragging, the image of alignment objects becomes more difficult to understand in three dimensions than in two dimensions. This difficulty has two sources: (1) The same number of hot vertices and active alignment values can produce more displayed curves in three dimensions than in two. (2) Because the screen image is a projection of three-dimensional space, the orientation of alignment lines is ambiguous.

In two dimensions, with n hot points, and k point-triggered alignment values, as many as $n\ k$

alignment objects will be drawn. The actual number may be less than this because two different points may trigger the same alignment object. For instance, the block letter G construction in Figure 3-17(a) shows 14 vertices and 2 active alignment types, but only 12 alignment lines are drawn of a possible 28.

In three dimensions, n hot points, and k point-triggered alignment plane types may cause as many as $n\,k$ planes, and $\frac{1}{2}\,n\,k\,(n\text{-}1)(k\text{-}1)$ intersection lines. Again, the actual number may be less than this. For instance, if the user activates the three plane directions parallel to the faces of a cube, as shown in Figure 5-7, only 12 out of a possible $\frac{1}{2}\times 8\times 3\times(8\text{-}1)\times(3\text{-}1) = 168$ intersection lines are drawn. Even so, the rectangles drawn to indicate the plane positions are distracting (Figure 5-7(a)). This distraction can be reduced either by making few points hot (two hot corners on the same major diagonal will generate exactly the same lines shown in Figure 5-7(a)) or by using alignment lines instead of alignment planes, as shown in Figure 5-7(b). As a rule of thumb, the user will want to choose hot points and alignment plane types in such a way that the number of planes is no larger than 15 or maybe 20. In those cases where alignment lines can be used instead of alignment planes, three-dimensional snap-dragging produces $O(nk)$ alignment lines, just as two-dimensional snap-dragging does.

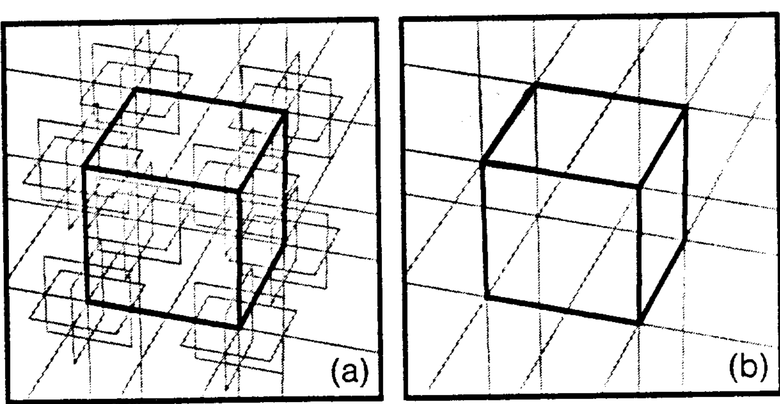


Figure 5-7. Manhattan lines and planes. (a) All eight vertices of a cube trigger (0 0), (0 90), and (90 0) alignment *planes*. (b) All eight vertices trigger (0 0), (0 90), and (90 0) alignment *lines*.

The problem in identifying alignment line directions is illustrated in Figure 5-8. While the alignment lines in this figure look similar to those in Figure 5-7, (0 65) alignment lines are being used instead of (90 0). In other words, all of the lines in Figure 5-8 are in the plane of the square, but the (0 65) lines almost look as though they are going in and out of that plane. One hint that

the scenes are different comes from the perspective projection; the (0 65) lines diverge from bottom to top, while the (0 90) lines in Figure 5-7 converge. I have not found any entirely satisfactory solution to this problem. The user can resolve the ambiguity by glancing at the alignment lines menu to see what alignment line directions are active or by measuring the direction of a given line.

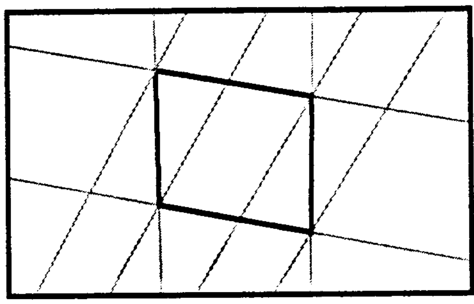


Figure 5-8. Ambiguity in alignment line orientation. The four vertices of a square triggering (0 0) (0 65) and (0 90) alignment lines.

5.2.3 Facilitate Reuse of Work

One of the biggest ways to save time in composing a scene, is to reuse geometry from existing scenes. In Sutherland's 1963 paper on Sketchpad, he remarked:

> Each time a drawing is made, a description of that drawing is stored in the computer in a form that is readily transferred to magnetic tape. A library of drawings will thus develop, parts of which may be used in other drawings at only a fraction of the investment of time that was put into the original drawing.

To reuse a geometric object, the borrower must understand the hidden structure built into the data by the original designer and whether or not that object can be modified in such a way that it can be incorporated in the new design. These factors are discussed in this section.

Reduce archaeology. All 2 1/2-D editors allow some amount of hidden structure in a scene. In particular, because one object may obscure another object, a scene may contain arbitrary amounts of unseen geometry. My colleagues often complain that they have tried to edit a picture made by someone else and the picture turned out to be much more complex than it looked. Three-dimensional scenes have similar problems. For instance, a scene built using Boolean point set operations (union, intersection and difference) on a set of primitives can be built in any number of ways using different sized primitives and/or a different ordering of point set

operations. Neither grid, constraint, nor snap-dragging approaches address this problem. However, the constraint approach compounds the problem by adding a constraint network to the list of kinds of hidden structure in a picture. To reuse constrained geometry, the borrower must learn not only what shapes are present, but must understand what variations on the shapes are allowed or prevented.

Support transformation-invariant editing. Whether the user is borrowing geometry made by someone else, or reusing his own creations, he will get more use out of them if they can be easily translated, rotated, scaled or skewed into new positions and then edited further. Both grids and constraints can hinder this process. Grid systems usually provide operations to transform objects. However, transformations can take the objects off the grid. For instance, if a rectangle is constructed on a grid and rotated 60 degrees, its vertices are probably not on grid points any longer, making further editing of the rectangle difficult.

Oddly enough, some constraint systems exhibit a similar problem. If a rectangle is defined, as shown in Figure 5-9, by constraining all 8 of its degrees of freedom, then reusing the rectangle at a new position or orientation requires changing the constraint network. It is relatively straightforward to update the constraints automatically in response to an interactive translation or rotation, but not all constraints systems have this capability. Van Wyk's IDEAL language [VanWyk82], on the other hand, allows shapes to defined with some degrees of freedom unconstrained. These definitions are "called" by providing enough constraints to constrain the remaining degrees of freedom, making IDEAL shapes easy to use in different contexts.

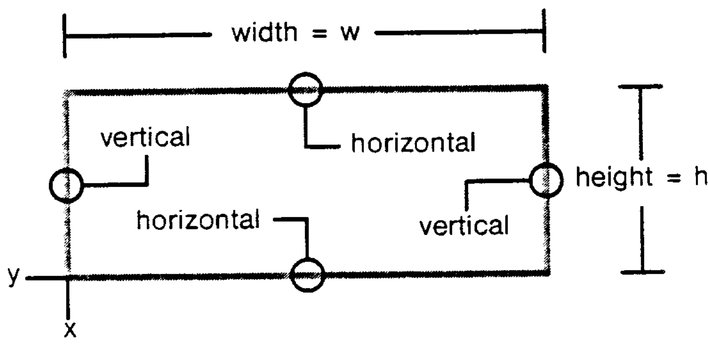


Figure 5-9. Constraining the eight degrees of freedom of a rectangle.

5.2.4 Summary

Figure 5-10 graphically summarizes some of the aspects of comparison for two-dimensional geometric design. The time per operation is bounded below by the speed with which a user can invoke a command. For graphical operations, let's consider the time to move the mouse and click a button to be this minimum time t_1. For creating a scene from scratch, the user must somehow enter an initial position for each of the scene vertices. In this case, at least one operation will be needed to enter each point. Call this number of operations n_1. This puts us at point A in the *n-t* plane as shown. However, with constraint systems, the minimum number of operations n_2 will be larger than this, because constraints must be added to position the points, putting us on a more expensive point B in the *n-t* plane.

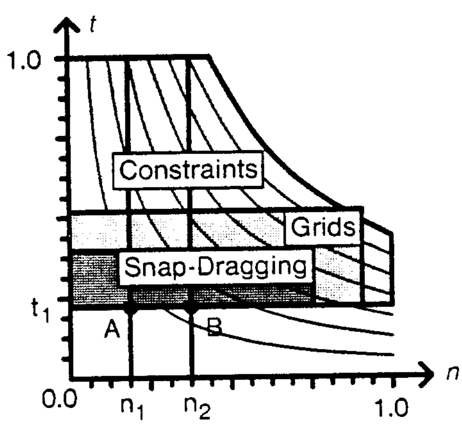


Figure 5-10. Productivity ranges for the techniques, in two dimensions.

We can reach points to the left of n_1 by reusing previous geometry. In the limit, we can compose a desired scene in no operations at all if the scene has already been drawn by someone else. In the case where some changes are needed, constraint-based systems stand to do very well, since it may be possible to reach the desired state just by changing the numerical value associated with one of the constraints. For instance, if the scene is a typeface where the stroke widths of all of the characters are related by simple constraints, it may be possible to create a heavier typeface simply by changing the stroke width of one of the characters.

In three dimensions, grids cease to be practical. Snap-dragging on the other hand gains an extra advantage, because the alignment objects allow the mouse to be used directly to place scene points in three dimensions, almost as rapidly as in two dimensions. A constraint user will have to

fall back on some other approach such as typing coordinates, or building up the scene from predefined volumes and then adding constraints to modify them. As a result, the lower bound on the average operation time may rise to a higher value t_2 for constraints while remaining near t_1 for snap-dragging, at least when scenes are created from scratch. This increases the discrepancy between points A and B as shown in Figure 5-11.

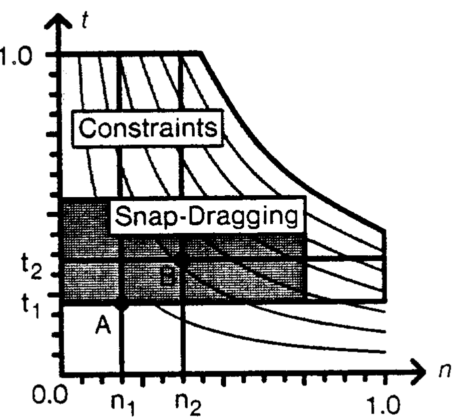


Figure 5-11. Productivity ranges for two techniques, in three dimensions.

In conclusion, for many scenes in both two and three dimensions, snap-dragging offers a way to precisely create those scenes in a shorter time than either the grid or constraint approaches. It is to be expected however, that the constraint-based approach still offers advantages in those cases where objects with moving parts are to be modeled, as for animation, or indeed whenever the designer knows *a priori* which parameters of the scene he is going to wish to experiment with, as is the case with the font designer changing the stroke widths of the characters in a typeface. Likewise, grids will continue to be popular among those users who need a quick way to get a simple job done.

It may be possible to satisfy all of these users with a layered system that provides a grid by default, provides snap-dragging upon request, and allows constraints to be added by sophisticated users. Grids and snap-dragging would be used for the bulk of the scene-composition tasks, with constraints used to model object behavior and to enforce early design decisions. The construction of such a hybrid system is the subject of on-going research.

6. Experiences with Users

When the *Drink* button was pressed it made an instant but highly detailed examination of the subject's taste buds, a spectroscopic analysis of the subject's metabolism and then sent tiny experimental signals down the neural pathways to the taste centers of the subject's brain to see what was likely to go down well. However, no one knew quite why it did this because it invariably delivered a cupful of liquid that was almost, but not quite, entirely unlike tea.

– Douglas Adams
The Hitchhiker's Guide to the Galaxy

Gargoyle was released in July 1986 (20 months ago) to a community of users, mostly in the Computer Science Laboratory (CSL) of the Xerox Palo Research Center. Within several months, Gargoyle was in regular use by 20 to 30 people including research scientists, graduate students, secretaries, and artists. A brief history of the Gargoyle project is provided in section 6.1. Three kinds of data have been collected on the experiences of users with Gargoyle.

1) Questionnaires have been used both to find out how many pictures are being made and to understand what aspects of snap-dragging are being used. These results are given in section 6.2.

2) Gargoyle is capable of recording all user commands in a log. For several months, these logs were collected from a number of users. This data shows, on a keystroke by keystroke basis, how frequently each of the Gargoyle commands is used. These results are summarized in section 6.3.

3) Users were solicited for examples of the pictures they had created with Gargoyle. Some of these pictures are included in section 6.4.

In addition, videotapes are being taken of Gargoyle sessions to get a better understanding of the strengths and weaknesses of the Gargoyle user interface. This ongoing work will be discussed in Chapter 7.

6.1 A Brief History of the Gargoyle Project

Gargoyle began as a summer project in CSL in 1985. With Maureen Stone, I designed a two-dimensional illustration system that would serve as an editor for the graphical objects described in the Interpress page description language [Bhushan86] and that would incorporate a novel

approach to precise editing. During that summer, I built a two-dimensional editor for straight-line objects that demonstrated the ideas of gravity, alignment objects, and interactive transformations. Because the idea looked promising, I continued the project in the Fall, working part-time. Ken Pier joined the project in November of 1985 to help turn the simple prototype into a complete illustrator. A paper on snap-dragging [Bier86] was submitted in January to the SIGGRAPH '86 conference.

Beginning in 1986, I began to work at Xerox PARC full time. With the added help of David Kurlander, a summer student from Columbia University, Gargoyle gained enough functionality to be released to users in CSL in late July of 1986. By that time, Gargoyle could edit text, conics, and several varieties of spline curves, could apply different colors to curves and regions, had a file format, and was fairly reliable.

Within several months, Gargoyle had replaced the grid-based editor, Griffin, as the illustrator of choice in this community. It would be simplistic to attribute the change-over entirely to snap-dragging. Gargoyle also differs from Griffin in its support of color raster images and in the flexibility with which synthetic shapes can be split apart and welded together at arbitrary points. However, users were enthusiastic about the new construction technique.

During late 1986 and early 1987, work proceeded to make Gargoyle more object-oriented (in order to prepare for expansion) and to support a larger fraction of the shapes present in Interpress. The gravity function was reimplemented to compute intersection points on the fly, a change that improved performance and made the code simpler, since it was no longer necessary to keep a set of computed intersections up-to-date.

In April of 1987, I began work on a prototype three-dimensional editor that came to be called Gargoyle3D. I began with the modeling and rendering routines present in Solidviews, a three-dimensional illustrator that I had built as part of my Master's work at MIT. I reworked this code so that input handling and screen refresh worked as in Gargoyle. Beginning with the "block" (constrained rectangular parallelepiped) shape class, I implemented the routines that Gargoyle objects implement, including routines to display the object while some of its parts are being interactively transformed, to draw selection and attractor feedback, to decide which parts should be selected when the skitter is nearby, and to change colors of lines and faces.

By mid-year, I had implemented the gravity function and interactive transformations

described in this dissertation. In September and October, my snap-dragging prototype came together. Alignment lines, planes, and spheres were added, along with code for computing their intersections. In addition, a line segment class was implemented as an alternative to "blocks." I began work on this dissertation and on a SIGGRAPH paper, "Snap-Dragging in Three-Dimensions," which was submitted in January of 1988.

As of this writing, Gargoyle3D is still far from being a complete illustrator. It lacks a number of features that would be essential to its acceptance by serious users: it is not integrated with a reliable shading renderer and its only modeling primitives are blocks and straight line segments. It has only been used by a few alpha-testers. However, better rendering is becoming available and the object-oriented structure of Gargoyle3D will facilitate the inclusion of new classes of modeling primitives. In the years to come, we will see to what extent skills transfer between the two editors and how frequently designers make use of three-dimensional snap-dragging to produce solid models and illustrations.

6.2 Feedback from Users

For the most part, each researcher in CSL has a Dorado workstation on his/her desk. The Gargoyle software is included as part of the standard software that is periodically installed on most Dorados. Thus, CSL researchers can use Gargoyle without special arrangements and can interleave graphical editing with text editing, reading mail, writing code, and so on. In fact, Gargoyle illustrations have been used as comments in Cedar program modules. To gain an understanding of how Gargoyle is being used, I sent two questionnaires to users, one to find out how many illustrations were being made, and a second to discover how users felt about snap-dragging. The results are reported in this section.

By August of 1987, Gargoyle had been in use for just over a year. Users were asked how many Gargoyle pictures were stored on their file server accounts, and how many of these they had made themselves. Twenty-six individuals responded to the survey. They reported having 2831 files on their accounts, of which more than 1500 were original versions. These Gargoyle users divided neatly into three groups: one heavy user (our staff artist) with over 934 original pictures on file servers, three frequent users (2 Gargoyle implementors and a manager) with over 356 original pictures between them, and twenty-two occasional users (researchers and summer

students) with over 211 original pictures between them. These totals and the corresponding average values are summarized in Table 6-1. These figures are a conservative lower bound on the number of pictures that were produced. Where users said that "most" of the pictures on their account were made by them, I have used 50% of the number reported. Also, pictures stored on local disks and pictures included in text documents rather than stored as separate picture files are not counted.

1 Heavy User	934 pictures	934 pictures average
3 Frequent Users	356 pictures	119 pictures average
22 Occasional Users	211 pictures	10 pictures average
26 Users	1501 pictures	58 pictures average

Table 6-1. The number of original Gargoyle pictures on file servers after 13 months.

Our staff artist produced a particularly large number of illustrations because he was using Gargoyle to make key-frames for a two-dimensional animation sequence showing some of the steps in the fabrication of integrated circuit chips.

In March of 1988, Gargoyle had been in use for 20 months. A second questionnaire was distributed to discover how people felt about Gargoyle and snap-dragging after using it for an extended period. Nineteen users responded. Not all users answered all questions. The questions and answers are summarized below.

1) What sorts of pictures have you made with Gargoyle? (18 users responded. Next to each response in Table 6-2 is the number of people that named each type. A given user could name more than one type. I have sorted the responses into four categories.)

Technical Drawings		Business Drawings	
box-and-pointer:	6	business graphics:	3
graphs:	2	business forms/forms:	2
networks:	4	stationery:	1
technical line drawings:	5	logos:	1
technical illustrations:	2	badges:	1
annotating circuit schematics:	1		
perspective drawings:	1	**Miscellaneous**	
isometric drawings:	1		
		page layouts/text assemblages:	7
		title pages:	1
Art		floor plans/blueprints:	4
cartoons:	2	house electrical plans:	1
simple art/crude art:	4	font design:	2
illustration art:	3	signs:	2
book dust jacket:	1	greeting cards/party invitations:	3
representational line art:	1	maps:	2
		tools (color palettes, rulers, crop marks, brackets):	1

Table 6-2. Picture types that have been made with snap-dragging.

2) To what uses have you put these pictures? (18 users responded. The responses, in Table 6-3, are weighted by the number of people that named each use. A given respondent could name more than one use. I have sorted the responses into four categories.)

Presenting Technical Ideas		Business	
illustrations for technical papers:	9	stationery:	1
illustrations for talks:	10	business cards:	1
lecture notes:	1	flyers:	1
design documents:	3	trade show booth art:	1
papers:	1		
documenting hardware & software:	6	**Miscellaneous**	
		book layout:	1
Advertisements		dust jacket for a published book:	1
T-shirt designs:	7	personal use:	2
advertisements/posters:	3	as a texture map for 3D graphics	1
greeting cards/party invitations:	2	animation keyframes	1
banners:	1	no use:	1

Table 6-3. Applications for which snap-dragging has been used.

3) For your applications, how useful is it to be able to snap the cursor to the following? (The responses of 18 users are shown in Table 6-4. Our staff artist emphasized that snapping to intersection points was "very very" important for his applications.)

	Very	Moderately	Slightly	Not at all
Vertices and control points?	16	1	1	0
Lines and curves?	13	4	1	0
Intersection points?	16	0	2	0

Table 6-4. The importance of snapping to various geometric features.

4) How frequently do you change the strength (extent) of gravity? (The responses of 17 users are shown in Table 6-5. One of the "Frequently" responses was from our staff artist.)

Frequently	Occasionally	Rarely	Never
6	8	1	2

Table 6-5. The frequency of gravity strength changes.

5) Gargoyle has two gravity types, points-preferred and lines-preferred. Points-preferred is the default. When gravity is turned on, how often do you use points-

preferred? (The responses of 16 users are shown in Table 6-6.)

Always	Usually	Half-The-Time	Rarely	Never
10	4	0	2	0

Table 6-6. Use of points-preferred gravity.

6) How much do you use these alignment types? (The responses of 17 users are shown in Table 6-7.)

	Frequently	Occasionally	Rarely	Never
Slope lines?	13	3	1	0
Circles?	5	7	4	1
Distance lines?	3	7	6	1
Angle lines?	2	2	9	4
Midpoints mode?	6	7	2	2

Table 6-7. Use of alignment types.

7) How frequently do you add values to the alignment menus? (The responses of 18 users are shown in Table 6-8.)

Frequently	Occasionally	Rarely	Never
10	3	3	2

Table 6-8. Adding values to alignment menus.

8) Do you usually leave the "automatic rule" turned on? (The responses of 17 users are shown in Table 6-9.)

Always on	Usually on	Occasionally on	Usually off	Always off
5	7	3	2	0

Table 6-9. Leaving the automatic rule on.

9) How often do you use interactive rotation, scaling, or skewing? (The responses of 16 users are shown in Table 6-10.)

Frequently	Occasionally	Rarely	Never
8	3	2	3

Table 6-10. Use of interactive rotation, scaling, and skewing.

10) How did you learn to use Gargoyle? (The responses of 18 users are shown in Table 6-11. The responses are weighted by the number of people that named each source of information. A given user could name more than one source.)

From the tutorial document:	13
From the reference manual:	2
By word of mouth/demo:	8
Trial-and-error:	15
By writing the code:	1

Table 6-11. Learning snap-dragging.

11) What techniques, other than snap-dragging, have you used for specifying precise point positions and transformations? What do you like/dislike about snap-dragging relative to these techniques? (These responses are summarized below.)

12) Other comments? (These responses are summarized below.)

While space prohibits reproducing all of the responses to questions 11 and 12 here, I will summarize them and include representative quotations.

How snap-dragging compares. Of the 10 users who responded to question 11, most had used some other illustrator before, usually one where grids were used as the main source of precision. In the five questionnaires that actually discussed the comparison, snap-dragging was preferred to grids. One respondent wrote:

> I have used grids before and they were OK for box-and-arrow diagrams for a while, but usually had problems when I went to make fine-tuning changes to the picture.

Another respondent made the comparison to grids this way:

> Snap-dragging and alignment lines are much easier to use and allow more precise placement.

Another respondent who had used grids and simple gravity in other systems wrote "snap-dragging is much more general."

Informal impressions have been similar. One Gargoyle user remarked in a conversation that he was pleased that he did not have to "play games on a lattice" (i.e., figure out how to get his design to work out evenly on a regular grid). Our staff artist remarked that he missed grids for a while but came to enjoy designing his own custom grids made of alignment lines. A third user remarked that, after experiencing alignment objects, he would not want to be without them.

What users disliked. One user, who reported in question 10 that he had learned snap-dragging exclusively by trial-and-error, also commented that he had trouble using snap-dragging:

> I quite often find myself struggling to get snap-dragging to do what I want. I'm a trial-and-error learner, and there seem to be things about snap-dragging that I have been unable to deduce either from reading the manual or playing around. There are times that I just cannot get the cursor to snap to, say, the midpoint of a line, no matter how hard I try and whatever modes or gravity I use. There are other times I cannot get it to snap to a line and avoid the points. Yet! At other times it does the above things perfectly for me.

Another user remarked that the screen frequently became cluttered with feedback.

> [The] trouble with snap-dragging is that the selection feedback/anchor/hot feedback frequently make it difficult to see what you're doing.

Another noted that it was sometimes necessary to measure objects, perform calculations on the resulting values, and then add the results to the alignment menus to create a shape when no obvious construction came to mind. One respondent remarked that, when compared to editing pictures using computer programs, interactive techniques have more trouble selecting desired points:

> [Dislike:] The awkwardness of snapping to the "right" point in a mess of points. With programs, I can just name the point.

Other users had a set of features that they would like to see added.

Overall impressions. Most respondents who commented on snap-dragging were pleased with it. One respondent reported getting a lot of use out of gravity:

> I just feel secure that I can scale my diagrams without fear of discovering a place that doesn't join -- and it doesn't cost me anything to be a perfectionist!

Some respondents included suggestions for extensions to Gargoyle. For instance, one user, who was sold on snap-dragging, also wanted a grid and a ruler to show approximately how large shapes are without an explicit operation to measure them (a suggestion that is in tune with the principles described in chapter 5):

> Adding specific numbers to the alignment menu, using grids and rulers, then combining with snap-dragging . . . what else is there? I often want more automatic grids, rulers, and other

objects that are not really objects to augment the free-hand aspects of snap-dragging.

One mostly-positive respondent noted that constructions did not always come out as expected, and that a simpler user interface might be appropriate for everyday tasks:

> Gargoyle is the best illustration system I've used, and the snap-dragging paradigm makes it much easier to make precise drawings than in grid-based systems, even those with gravity for snapping to endpoints of objects. I do occasionally find that I have missed making an alignment I intended, but have not yet determined the major cause of this (carelessness?). I think my use of snap-dragging has less of the "recreational geometry" feel of many of the Gargoyle demos, which might mean that less functionality would be adequate. However, I believe I have used each and every one of the alignment features at least once, and they were critical at that point. Overall, I'm very happy with the system.

Some respondents had no reservations. One wrote "I think snap-dragging has sufficed for everything."

These results and those of sections 6.3 and 6.4 will be discussed in section 6.5 below.

6.3 Some Operation Count Data

During the last few months of 1987, 17 Gargoyle users participated in a user study. Each Gargoyle operation performed by these users was automatically recorded in a session log. Each user ran a program that totals up the number of times each operation is invoked during a session and sent me the results by electronic mail. I ran a second program that computes keystroke totals for each user, sorted by type of operation. 225,064 keystrokes were collected in a four month period. This data can be used to discover what operations are actually used and with what frequency. Here is a list of the types of commands in each category:

At the time of the study, Gargoyle had approximately 340 distinct commands not including services provided by other tools (e.g., colors can be mixed in a separate color mixing tool). In order to interpret the results, I have divided these 340 commands into 14 categories. I will describe the types of commands that each category contains.

1) *Selecting and Searching.* Includes commands for selecting and deselecting vertices, edges, paths and regions, and clusters of objects when the user points to them. Also includes commands for selecting classes of objects, such as "all objects with blue borders."

2) *Adding and Deleting Geometry.* Includes commands for adding predefined shapes

such as circles and squares, for adding text by typing, and for deleting selected object parts.

3) *Interactive Adding Geometry.* Includes the "Caret Placement" command, the "Add Line Segment" command, and a command for sketching in rectangles.

4) *Interactive Transformations.* Includes translation, rotation, scaling, and skewing.

5) *Menu Transformations.* Includes translation, rotation, scaling, and skewing by numerical values, which the the user can select from any text document.

6) *Viewing.* Includes scrolling, zooming, and centering.

7) *Style.* Includes commands for changing area color, line color, line width, dash pattern, and text font.

8) *Set State.* Includes commands for setting the default area color, default line color, default line width, default dash pattern, and default text font.

9) *Set Snap-Dragging State.* Includes commands for dropping the anchor, making objects hot or cold, activating alignment types, modifying the alignment menus, and changing the gravity type.

10) *File Operations.* Includes commands for saving and retrieving illustrations to and from disk storage, for moving illustrations to and from illustrated documents, and for printing.

11) *Overlap Order.* Includes commands for changing the order in which objects are drawn.

12) *Groups.* Includes commands that give a name to an arbitrary collection of scene parts. These collections can then be selected by name.

13) *State Browsing.* Includes queries about the current state of the scene and requests for on-line documentation. The system will respond with textual descriptions of colors, text fonts, dash patterns, and so on.

14) *Special Actions.* Includes operations to undelete the most recently deleted objects, redo the last command on a new selection, or abort the operation in progress.

The results are summarized in Table 6-12. This data represents 225,064 commands from

3058 session logs collected from 17 users over the period 9/21/87 to 1/27/88. I have underlined the fractions for those categories that accounted for more than 5% of all commands. To the right of the fraction for each category, I list the most frequently used commands from that category including all commands that individually averaged more than 5% of all commands and a few of the most frequently used commands that averaged less than this.

Category	Fraction	Commands
(1) Select And Search:	<u>0.385</u>	(Select Joint 0.173, Select Top Level 0.085, Extend Selection 0.055, Select Segment 0.031)
(2) Add and Delete:	<u>0.176</u>	(Add Character 0.126, Delete 0.034, Close 0.004)
(3) Interactive Add:	<u>0.132</u>	(Caret Placement 0.083, Add Line Segment 0.029, Copy And Translate 0.017)
(4) Interactive Transformations:	<u>0.090</u>	(Translate 0.080, Scale 0.005, Rotate 0.003)
(5) Menu Transformations:	0.008	(Menu Scale 0.004, Menu Rotate 0.002, Menu ScaleX 0.001)
(6) Viewing:	0.036	(Scroll 0.014, Zoom 0.007, Reset 0.002)
(7) Style:	0.028	(AreaColor 0.010, LineWidth 0.008)
(8) Set State:	0.005	(SetScaleUnit 0.001)
(9) Set Snap-Drag State:	<u>0.090</u>	(DropAnchor 0.024, MakeHot 0.013, ToggleGravity 0.012, LiftAnchor 0.007, ToggleSlope 0.008, ToggleRadius 0.005, MakeAllCold 0.005, ToggleDistance 0.003)
(10) File Operations:	0.030	(Get 0.006, Save 0.006, Print 0.005)
(11) Overlap Order:	0.004	(MoveToTop 0.002, MoveToBottom 0.002)
(12) Groups:	0.002	(SelectGroup 0.001, AddToGroup 0.001)
(13) Browse State:	0.005	(ShowFontValues 0.002, ShowStrokeValues 0.001)
(14) Special Actions:	0.007	(Abort 0.005, Again 0.001, Undelete 0.001)

Table 6-12. The fraction of keystrokes devoted to each category of command.

While keystroke count data does not directly tell us how much time the user spends on each of the sub-tasks, it is easy to collect and allows us to make rough comparisons between dissimilar operations and more convincing comparisons between similar operations. For instance, an "Add Character" command is counted whenever the user adds text to an illustration by typing at the keyboard. We expect the time per keystroke to be relatively small for typing, so little time is actually spent adding characters even though an average of 126 operations per 1000 are devoted to this task. On the other hand, we see that interactive translation is performed 0.080/0.005 = 16 times more often than interactive scaling. Scaling is more complicated since it requires placing the anchor, which in turn requires placing the caret. However, it seems fair to conclude that more time is spent translating objects than scaling them.

The operation counts suggest that Gargoyle users spend a significant amount of time performing scene composition tasks. The operations "Add Line Segment", "Copy And Translate", "Translate", "Scale", "Rotate", "Skew", all of the menu transformations, and all of the operations that set snap-dragging state are certainly scene composition operations. These account for 23.4% of all operations. The "Select Joint" operation is often used to make the Gargoyle viewer listen to subsequent keystrokes after the user has been using other tools, making the count for "Select Joint" hard to interpret. However, it is clear that all of the other selection operations support either scene composition operations or style operations. Since only 2.8% of all operations change style properties, vs. at least 23.4% for scene composition, it seems fair to count a proportionate number of selection operations towards scene composition. ("Select and Search" - "Select Joint") * 0.234/(0.234+0.028) $\approx$ 19.0% additional operations for scene composition, bringing the total to 42.4%. The "Caret Placement" operation is also an important scene composition operation, but like Select Joint, it is often used to make the Gargoyle viewer listen to subsequent keystrokes. Together, "Select Joint" and "Caret Placement" account for about 25.5% of all commands. Finally, the viewing, overlap order, grouping, and abort commands may be considered part of scene composition, representing another 4.7%. Hence, anywhere from 42.4% to 72.6% (42.4% + 25.5% + 4.7%) of all commands are devoted to scene composition.

On the other hand, relatively few commands are spent on other tasks. Setting style parameters, setting default style parameters, looking at textual descriptions of style parameters, and selecting the objects whose properties are to change accounts for about 6.1%. Filing and printing operations account for another 3%. Finally, 17.6% of all operations are devoted to adding characters, adding pre-defined shapes from a menu, and deleting shapes.

These data suggest that, in an illustration system, scene composition accounts for a significant amount of the total time spent. In particular, more time is spent describing geometrical aspects of the illustration than other aspects such as font, color, or dash pattern. This emphasis on geometry can be accounted for by noting that technical illustrations are often black-on-white line drawings. Faster precise scene composition, then, will almost certainly improve the productivity of designers making technical illustrations.

6.4 A Picture Gallery

All of the 84 figures and 17 tables seen so far in this dissertation were made with Gargoyle and Gargoyle3D. However, these pictures are almost exclusively technical line drawings, used to illustrate aspects of the implementation and use of snap-dragging and other interactive geometric design techniques. In this section, I show some of the other uses of snap-dragging, including fine art, T-shirt designs, poster art, floor plans, and whimsical illustrations.

Figure 6-1 was one of the earliest fine art drawings done with Gargoyle. Andrew Glassner studies Celtic knotworks as a hobby and has drawn many by hand. He noted that Gargoyle was the first illustrator he had used that actually helped him make Celtic knotworks. This picture was used as the cover for *Computer Graphics* (Vol. 21, No. 2, April 1987).

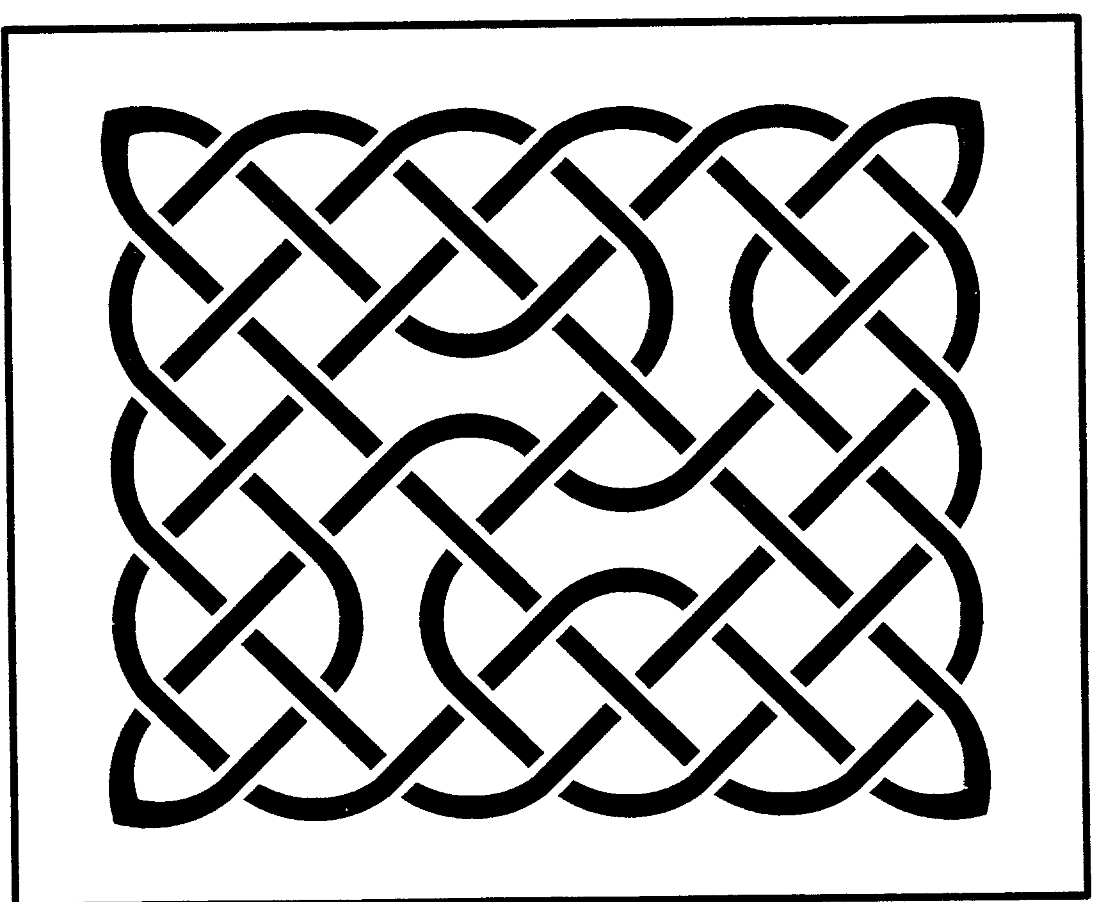


Figure 6-1. "Celtic Study 1," Andrew Glassner, 1986. Re-colored by the author.

Figure 6-2 was used as a T-shirt design for a softball team. Snap-dragging was used to make the motion lines parallel, to offset the motion lines from the softballs, and to arrange the letters in "Brownian Motion" to form an arc. This picture is made of 630 Gargoyle objects, where each object is either a black line, a filled region with a black border, or a text character. The black symbols around the border of the illustration were included to help the silk-screeners align the four color separations used to print the T-shirts.

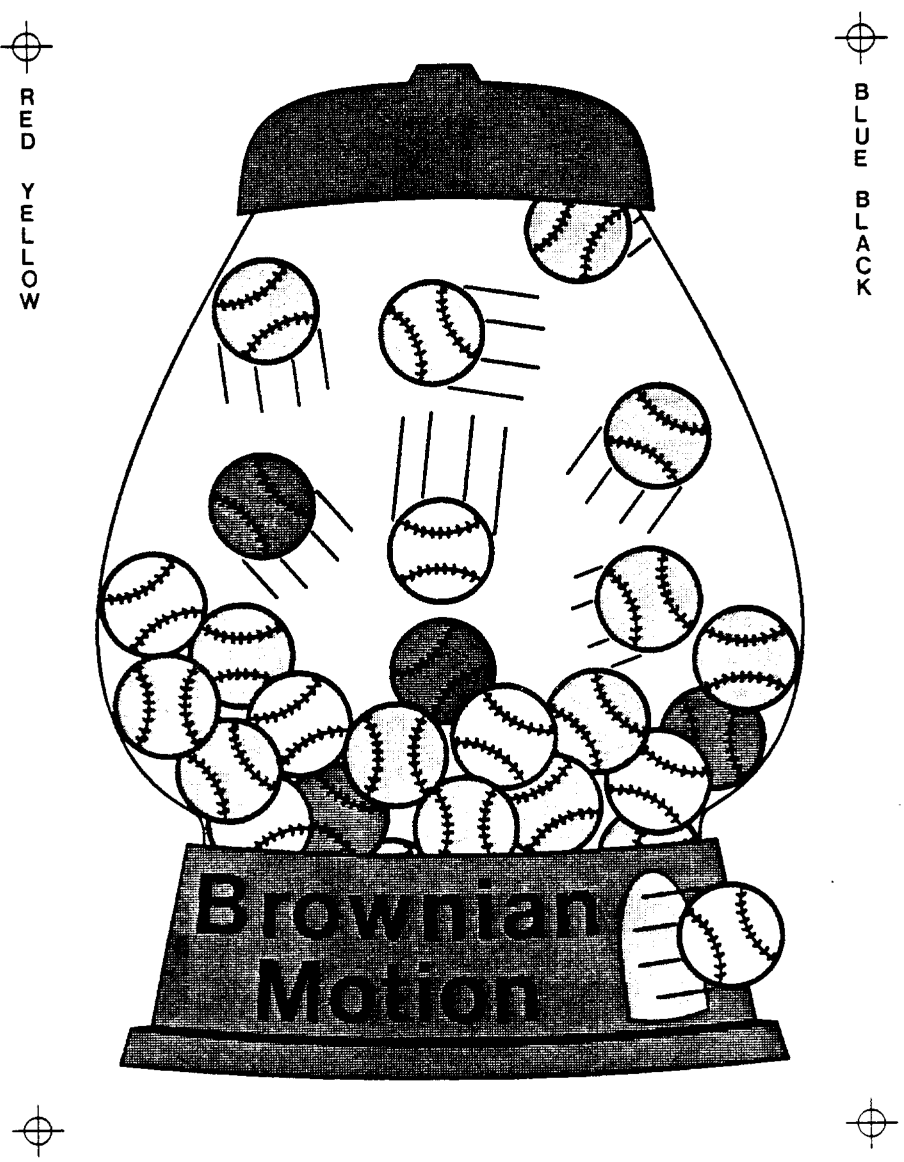


Figure 6-2. "Brownian Motion T-Shirt," Jock Mackinlay and Polle Zellweger, 1987.

Figure 6-3 is a T-shirt design for a group of square-dancers. Alignment lines and gravity were used to achieve symmetry, particularly in the switchblade. The loop of the noose was constructed, at first, of two paths of straight line segments placed at a constant offset from each other using distance alignment lines, to get the rope to have a constant width. An interpolating spline was fit to the resulting vertices, and the result was hand-edited to remove artifacts that resulted from the initial linear approximation.

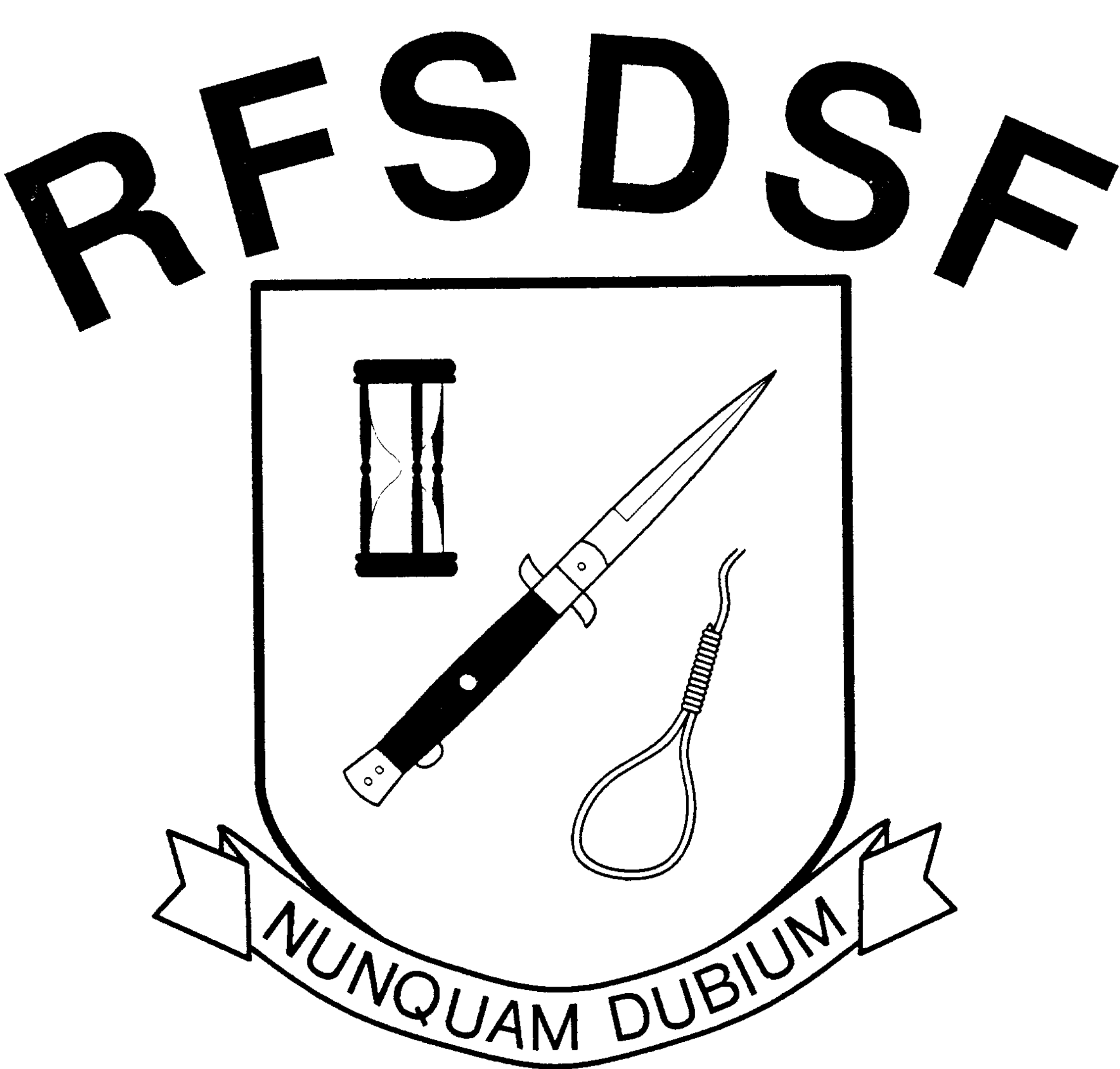


Figure 6-3. "Ray Fuller School of Dance and Street-Fighting," Pavel Curtis, 1987.

Figure 6-4 is a third T-shirt design. Its parts show several capabilities of Gargoyle and snap-dragging. The S of "Snap-Dragging" was constructed by David Kurlander, using a construction from a book of constructed Roman letters by David Lance Goines [Goines82]. The words "Snap-Dragging" and their surrounding rectangles were interactively skewed until the lower right corner snapped onto a control point in the gargoyle's hand. The gargoyle was constructed by Maureen Stone, using gravity to get the lines that make up the gargoyle to end precisely on other lines (e.g., the collar ends on the body) and alignment lines to make the geometrically regular parts of the picture, such as the gargoyle's pedestal and the stripes on the sleeves.

Figure 6-4. "Snap-Dragging T-Shirt," Maureen Stone and David Kurlander, 1986.

Our staff artist, Steve Wallgren, drew Figure 6-5 as an advertisement for a Christmas Party. Gravity and alignment lines were used to place the keys next to each other and to align the four groups of keys with each other. The keys of the workstation were originally constructed with horizontal and vertical lines, and then interactively skewed into their final shape. The Christmas tree scene displayed on the screen is a sampled image that was skewed into place.

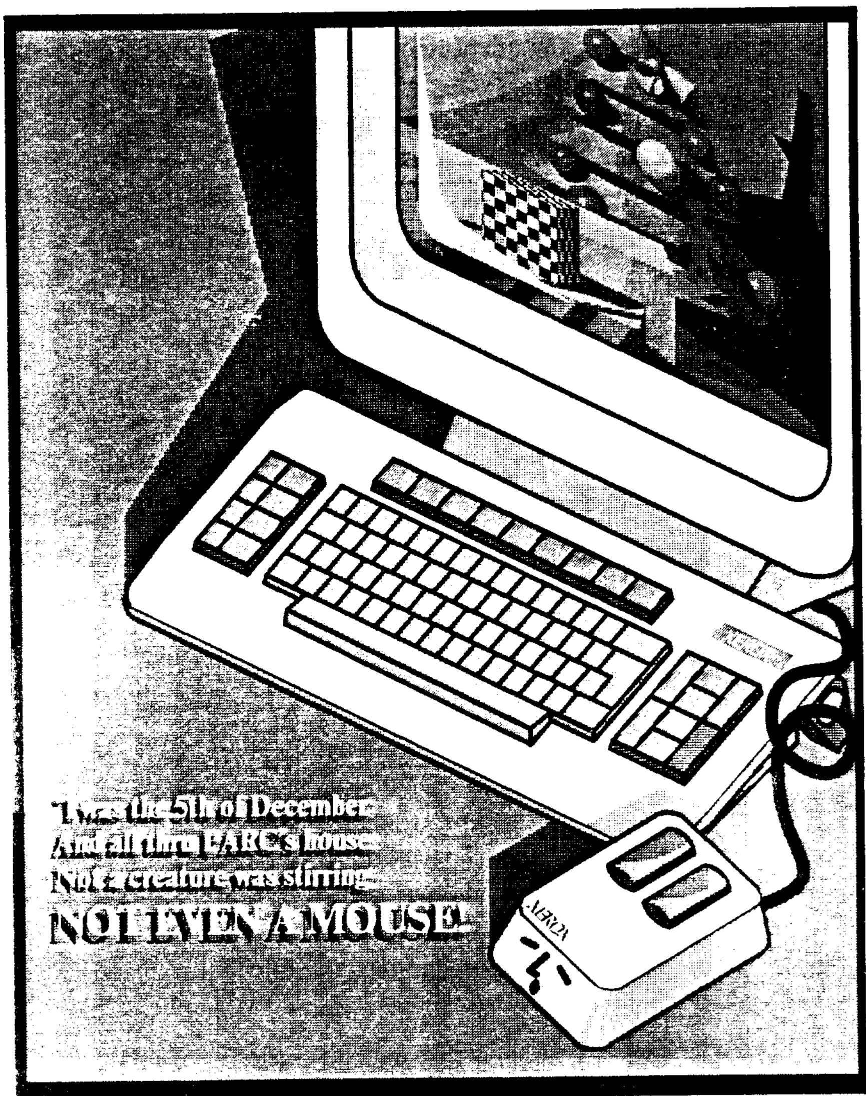


Figure 6-5. "Christmas 1986," Steve Wallgren, 1986.

Doug Wyatt used Gargoyle to construct a floor plan of his new house (Figure 6-6). Enlargements of this picture were used to plan remodeling and furniture placement. Doug used slope lines and distance lines heavily in this construction. Many measured values were added to the distance menus to allow the spacings between walls to be drawn to scale. This picture contains 235 Gargoyle objects.

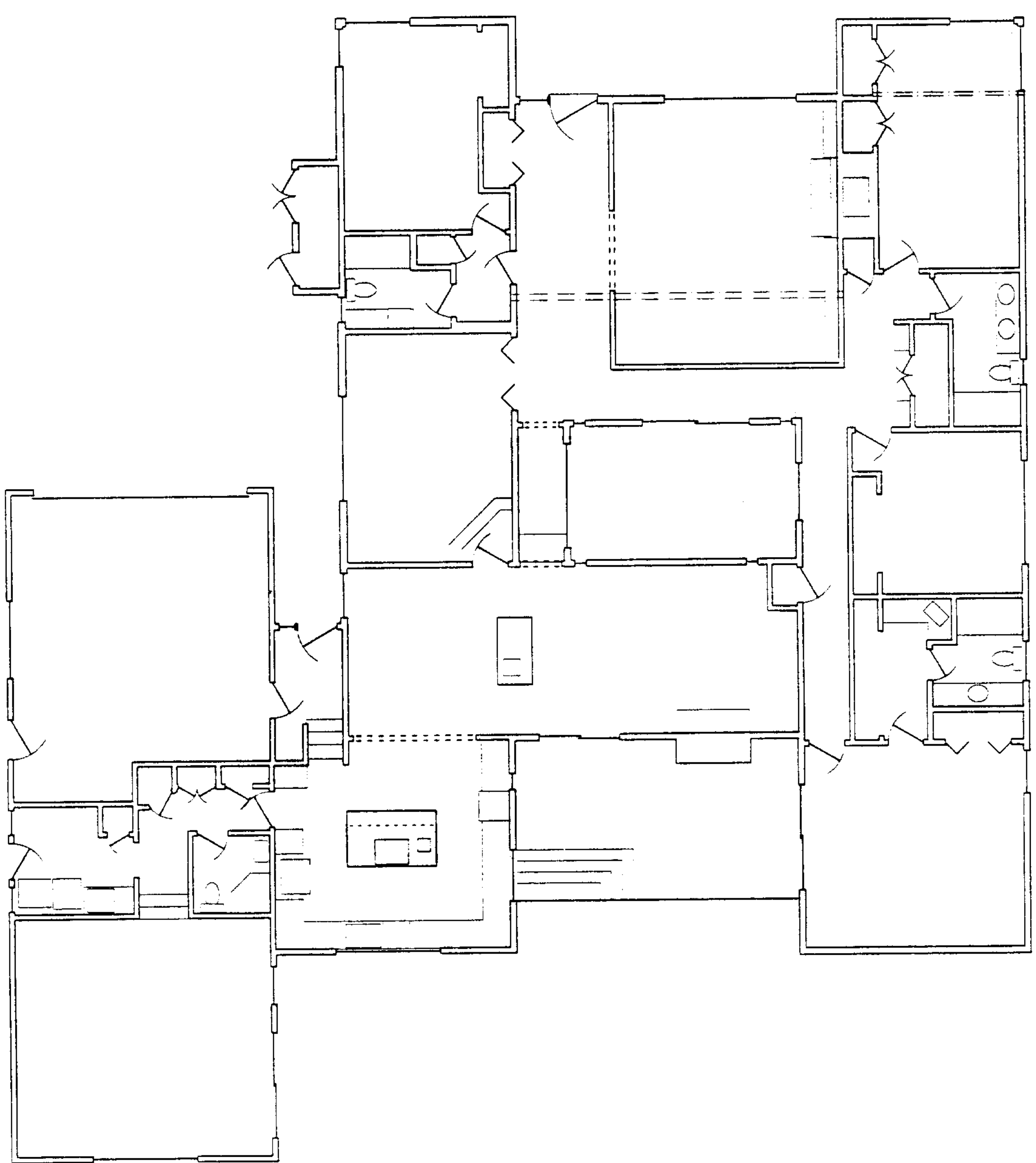


Figure 6-6. "House Plan," Doug Wyatt, 1987.

Figure 6-7 was produced using both Gargoyle and Gargoyle3D. It was made when blocks were the only primitives available in Gargoyle3D and the only alignment objects available were slope lines. Gargoyle3D was used to construct a model of the desk, Rubik's cube, envelope box, trays, shelf, books, and to place the poster on the wall. A projection of the 3-D scene was post-processed in Gargoyle where text and a scanned-in photograph were skewed into place.

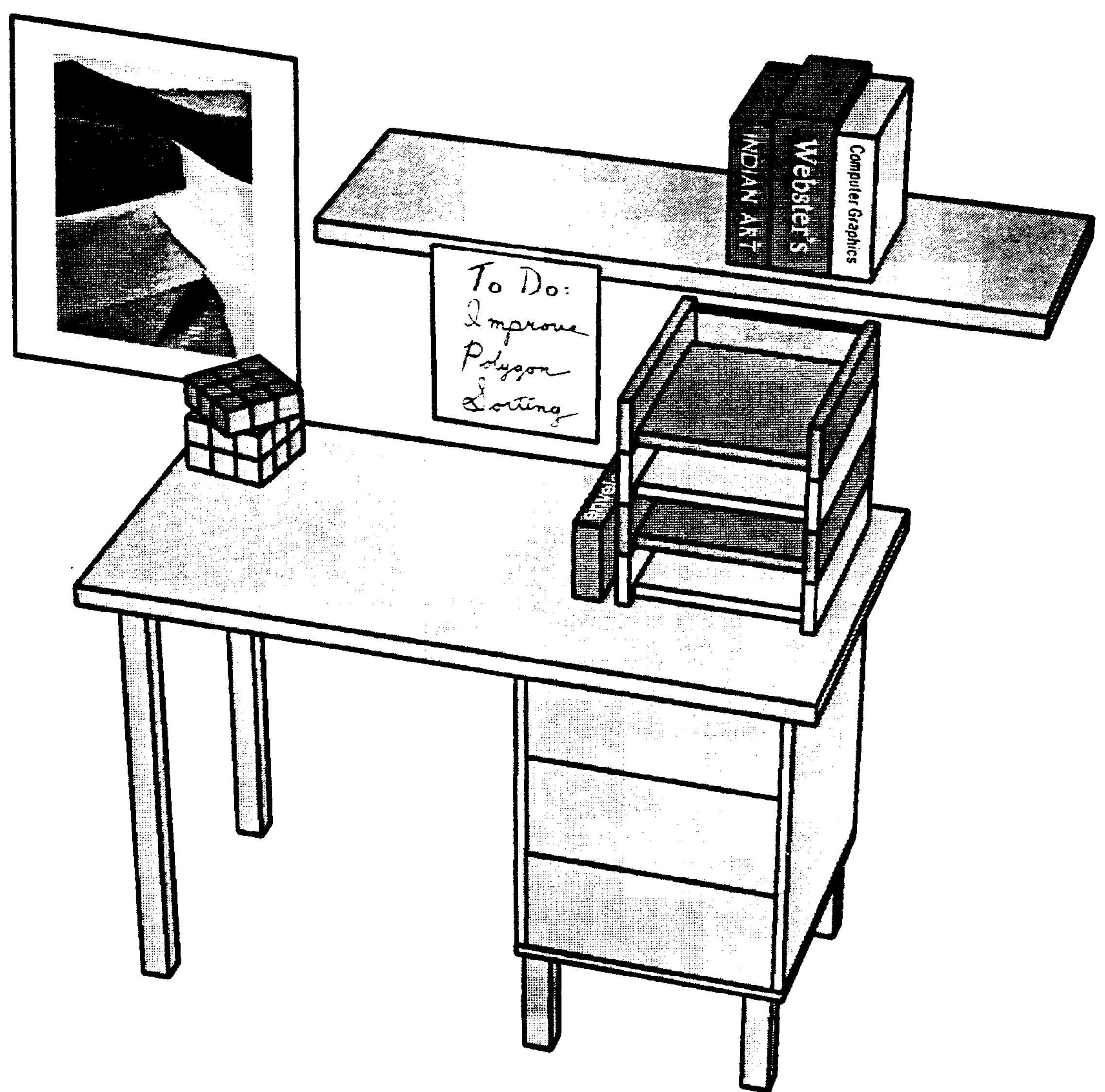


Figure 6-7. "Desk," Eric Bier, August 1987. The poster in the background is from "Sand Dunes, Sunrise" a photograph by Ansel Adams.

6.5 Discussion

The user studies described above suggest a number of conclusions about snap-dragging and its three sub-techniques: gravity, alignment objects, and interactive transformations. In particular, we can draw these conclusions about gravity:

1) A gravity function that snaps to vertices, control points, lines, curves, and intersection points is useful for the tasks performed at PARC. 16 of 18 respondents felt that it was very important, for their applications, to be able to snap the cursor to vertices, control points, and intersection points of curves. 17 of 18 felt that it was either very important or moderately important to be able to snap the cursor to lines and curves.

2) Points-preferred gravity does serve as a general-purpose mapping function. In fact, some users, including our staff artist, have never tried lines-preferred gravity. 10 of 16 respondents reported that they *always* use points-preferred gravity. One of the respondents who reported *rarely* using points-preferred gravity also reported that snapping to vertices and intersection points was only slightly important to his application while snapping to lines and curves was very important.

3) Changing the strength of gravity is used more often than changing from points-preferred to lines-preferred gravity types, as a way to tailor the gravity mapping. 14 of 17 users reported changing the strength of gravity either frequently or occasionally. Furthermore, 2.3 out of 1000 keystrokes, averaged over all users, go to changing the gravity extent, while only 1.3 out of a 1000 go to changing gravity type. Since most users report that snapping to lines and curves is important, and most users always employ points-preferred gravity, it seems fair to conclude that these users reduce the strength of gravity enough that there are no points within the points-preferred tolerance while they are snapping the caret to lines and curves.

We can draw these conclusions about alignment objects:

1) Slope lines, circles, distance lines, and midpoints mode are used either occasionally or frequently by a majority of users. In fact, of 17 respondents, 16, 12, 10, and 13 respectively reported frequent or occasional use of slope lines, circles, distance lines, and midpoints. Angle lines, on the other hand, are used either rarely or never by a

majority of users. Only 4 respondents reported using them frequently or occasionally. This data is corroborated by the keystroke statistics for the operations that activate and de-activate these values in the alignment menus. ToggleSlope, ToggleRadius, and ToggleDistance account for 8, 6, and 3 keystrokes, respectively, in every 1,000 keystrokes, averaged over all users. ToggleAngle, on the other hand, accounts for only 3 keystrokes in 10,000.

2) Most users add values to the alignment menus. 13 of 18 users reported adding values either frequently or occasionally. Keystroke statistics show that radii and slopes are added most often (2.1 and 1.2 keystrokes per 1,000, respectively), while distances and angles are added less often (0.7 and 0.1 keystrokes per 1,000, respectively). However, for special applications, such as the floor plan in Figure 6-6, and the placement of figures in this dissertation, values are added to the distance menu more often than to the other menus.

3) The "automatic rule" is either always on or usually on for a majority of users (12 of 17 respondents). Furthermore, the automatic rule accounts for only 8.9 keystrokes in 10,000. Together, these statistics suggest that little effort is spent turning this rule on and off, and that the rule is usually active when it is needed.

We can draw this conclusion about interactive transformations:

1) Most users do take advantage of interactive transformations other than translation. 11 of 16 respondents reported using interactive rotation, scaling, or skewing either frequently or occasionally. Keystroke statistics show that interactive rotation, scaling, and skewing are performed with frequencies of 3.4, 4.8, and 1.8 keystrokes per 1,000 respectively. For comparison, interactive translation is performed with a frequency of 80.2 keystrokes per 1,000. We can also conclude that performance optimizations to the implementation of translation are especially worthwhile.

We can draw these conclusions about scene composition and snap-dragging:

1) Scene composition is a major part of the process of completing an illustration. Keystroke counts suggest that from 42.4 to 72.6% of all keystrokes in Gargoyle go towards scene composition operations.

2) Snap-dragging supports many drawing applications. In responses to questionnaires, users recalled 27 types of illustrations and 19 different applications for the pictures they had made with Gargoyle. While these categories certainly overlap, it is clear that snap-dragging is capable of supporting many different tasks.

3) Tutorials and trial-and-error are important ways of learning to use snap-dragging. 13 users out of 18 cited the tutorial and 15 users out of 18 cited trial-and-error as important ways to learn snap-dragging.

4) Users who had used both grids and snap-dragging now prefer snap-dragging because it is "much more general" and allows "more precise placement". This conclusion is drawn from questionnaire comments. As noted in chapter 5, snap-dragging provides the functionality of grids, plus more besides, so it is not surprising that users find it more general.

5) There is more work to be done. As suggested by the questionnaire comments, graphical feedback (for selections, hotness, the attractor, and alignment objects) can be overwhelming, and some users would like a simpler interface for those times when only a simple task must be done.

From talking to users, I have discovered that complaints about feedback have two aspects. When there is a lot of feedback (e.g., when many objects are selected), the feedback takes a noticeable amount of time to display. Furthermore, many small shapes are drawn, which can clutter up the screen. Projects are underway to use simpler feedback when many shapes are selected and to speed up the drawing of feedback. Also, once clustering (hierarchical structuring) is implemented in Gargoyle, it will be possible use a single feedback token to show that all of the parts of a cluster are selected or hot. In the current implementation, a token is needed for each primitive object.

There are many ways in which Gargoyle could be simpler. As we saw in section 3.1.5, there are 19 different keyboard and mouse button combinations used for selecting, deselecting, adding line segments, adding splines, adding boxes, interactive transformations, caret placement, and copying. For occasional users, this is a lot to learn. In the future, Gargoyle may offer users the choice of activating these functions from menus, so occasional users have less to learn. Furthermore, in the future, advanced and infrequently used operations, be they snap-dragging

operations, operations for setting style parameters or operations printing results, may be separated out into an "advanced" control panel, reducing the number of buttons that users must sift through to find common operations.

More plans for the future are discussed in the next chapter.

7. Plans for Future Work

Provided we do not kill ourselves off, and provided we can connect ourselves by the affection and respect for which I believe our genes are also coded, there is no end to what we might do on or off this planet.

– Lewis Thomas
Late Night Thoughts on Listening to Mahler's Ninth Symphony

The snap-dragging techniques described in this dissertation were not originally intended as a complete scene composition approach. In my original plans, they were intended to serve as a front end for a larger system that was to include a constraint-solver and a set of routines for recognizing the symmetry of objects and using this symmetry as the basis for further editing operations.

To my surprise, the basic snap-dragging techniques of hot points, alignment lines, gravity, and transformations kept suggesting ways to perform geometric constructions without adding much complexity to the system. My previous experiences with user interfaces have convinced me that simplicity is indeed a virtue. As a result, I spent my time developing snap-dragging, and I continue to discover new techniques that would extend the utility of snap-dragging without changing its basic paradigm.

In this chapter, I discuss the original geometric duality that suggested snap-dragging and a similar duality that may make snap-dragging better at constructions involving rotations. Next, I discuss ways that snap-dragging could be extended to more rapidly edit symmetrical objects. Then, I discuss a snap-dragging technique for positioning the viewing pyramid (camera) precisely relative to a three-dimensional scene. Finally, I discuss some on-going work aimed at improving the user interface to snap-dragging in Gargoyle and Gargoyle3D.

7.1 A Duality: Moving Lines to Points vs. Moving Points to Lines

In my original vision of an interactive scene composition technique, I imagined a system where the user could select an object, and drag it around the scene. When an edge or symmetry axis of the moving object nearly lined up with a point in the scene, the moving object would make a little jump so that the edge or axis exactly lined up with the scene point. In this vision, alignment lines would follow the moving object around, snapping to things. Figure 7-1(a) shows a

hexagon being translated in this manner with three of its six symmetry axes visible. The user wishes to align one of these symmetry axes with the highlighted corner of the grey square.

There was a serious user interface problem with this approach. If the scene is filled with many objects, the alignment lines of the hexagon can be attracted to widely separated objects at nearly the same time; in order to understand the motion of the moving objects, the user would have to constantly scan the entire illustration. Otherwise, the motion would appear to be random. On the other hand, if we take the alignment lines off of the moving object and copy them onto all of the scene points that are of interest, then we have snap-dragging, as shown in Figure 7-1(b). In essence, we have taken the problem of translating a line to touch a point and turned it into the problem of translating a point to touch a line. Fortunately, with this second problem, the objects that cause the moving objects to snap are all near the cursor, so the motion of the moving objects is easy to account for. Other applications of the same duality suggested alignment circles, angle lines and distance lines.

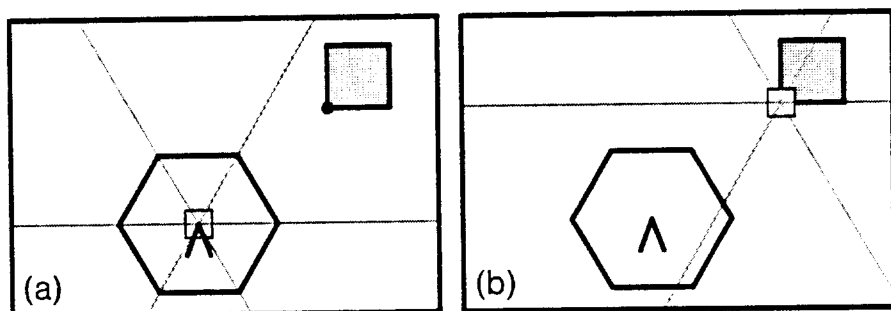


Figure 7-1. The duality of alignments. The user drags a hexagon by its center. (a) Alignment lines that move with the hexagon. (b) Alignment lines triggered by stationary objects.

This same duality continues to suggest ways of solving problems that come up. For instance, say we wish to rotate a white square around its center until one edge is collinear with the lower left point of gray square, as suggested in Figure 7-2 where the alignment lines rotate with the white square.

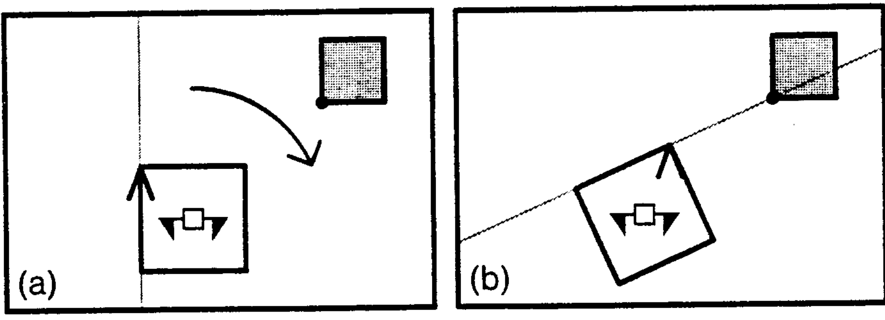


Figure 7-2. Alignment line that rotates with the object.

In the dual view, each scene point of interest would trigger an alignment object that we could snap the square to in order to complete the construction. One alignment scheme that would do the job would place at each hot point the two lines that are of a known distance from the anchor. If we set this distance to be half the length of the side of the white square, as shown in Figure 7-3(a), then we can accomplish the desired construction by snapping the caret to the intersection point of such a line with an alignment circle as shown in Figure 7-3(b).

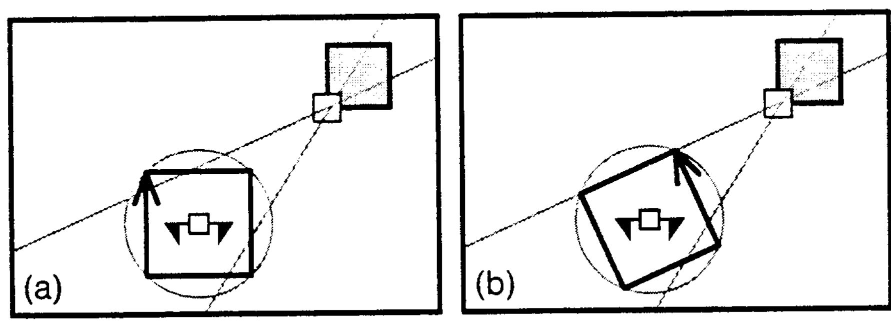


Figure 7-3. Rotating the square using alignment lines of known distance from the anchor.

It would not be worth adding these new alignment lines, call them anchor-distance lines, to the user interface just to facilitate the construction above. However, further reflection reveals other uses. For instance, by setting the distance to be equal to the radius of a circle, placing the anchor at the center of the circle, and making a point hot, one constructs the two lines that are tangent to the circle and pass through the hot point, as shown in Figure 7-4.

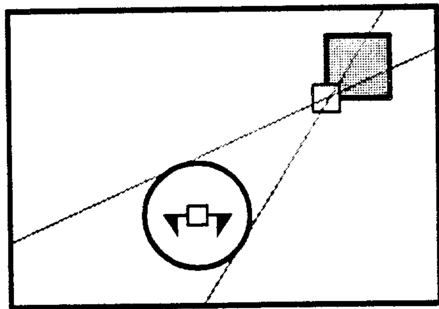


Figure 7-4. Anchor-distance alignment lines help construct the tangent from a point to a circle.

7.2 Symmetry Operations

Symmetry is an important element of geometry both in two and three dimensions. In two dimensions, for instance, it is often important to align the centroid and symmetry axes of regular polygons with other scene points. Centroids and symmetry axes are important in three dimensions as well. For mechanical design, symmetry is often a design principle; more symmetrical shapes can be assembled with repetitive methods and tend to be easier to manipulate with mechanical equipment. For instance, a robot hand will find a cylinder easier to grab than a block because it can be grabbed from many more angles.

Making symmetrical objects with snap-dragging requires a certain amount of sophistication. To construct a pentagon, for example, one would have to employ slope lines that differ by 72 degrees or angle lines of 72 degrees and sketch in each of the five edges sequentially. Likewise, to construct the midpoint of a line segment, one would have to measure the segment, manually divide by 2, and create alignment circles of the resulting radius. In Gargoyle and Gargoyle3D, I have compensated for these deficiencies by adding special-purpose features: Regular polygons are available from a menu, and line segment midpoints become gravity-active at the press of a button. However, there are no general purpose tools that take advantage of symmetry to improve scene composition productivity.

I have begun to design some symmetry tools that would work well with snap-dragging. One very simple technique is to use the anchor position as a center of symmetry, and to have the user specify a symmetry group. Subsequent operations will be done at symmetrical positions around the anchor. All of the symmetry axes (and symmetry planes in three dimensions) would be

represented by alignment lines (and alignment planes). For instance, in Figure 7-5(a), the user is adding a line segment from the lower right corner of the frame to the caret, which is slightly left of center. The user has selected dihedral symmetry of order 3 (see Weyl's book for a very readable description of common symmetries in two and three dimensions [Weyl52]). Hence, the user is adding six line segments at the same time. By snapping to an alignment line, the user can get the new segments to touch in pairs even though the moving objects themselves are not gravity-active, as shown in Figure 7-5(b).

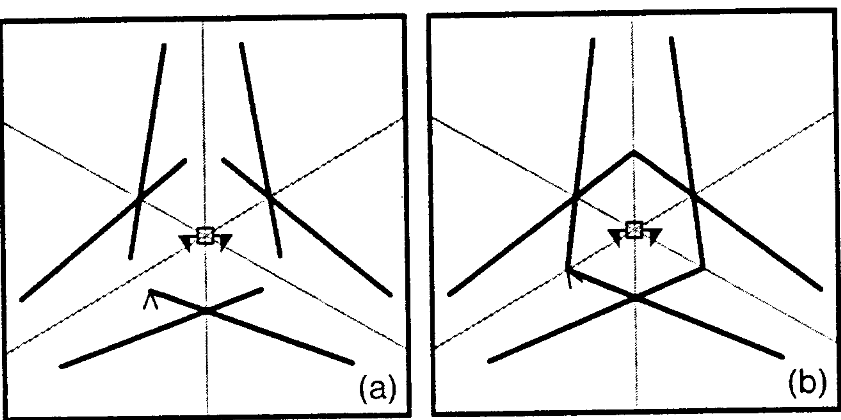


Figure 7-5. Adding a line segment while dihedral symmetry of order 3 is maintained.

A more elaborate technique would involve extending gravity so that snapping to moving objects is possible. This would facilitate constructions like the one shown in Figure 7-6.

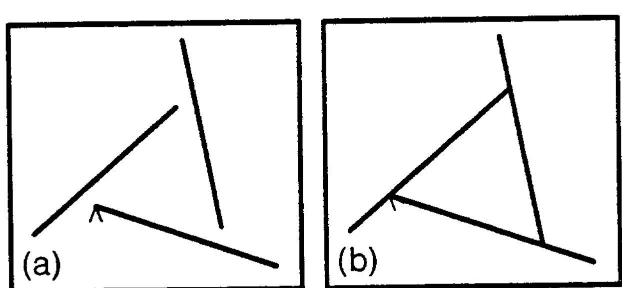


Figure 7-6. Adding a line segment while cyclic symmetry of order 3 is maintained and the segments created by symmetry are gravity-active.

One last idea is to automatically compute the center of symmetry, symmetry axes and symmetry planes of objects (when requested) and make them gravity-active. Often, these are the features of an object to which a designer most wishes to align other objects. The center of symmetry can be computed on the fly, like intersection points and midpoints in the Gargoyle

editors. Symmetry axes and symmetry planes would need to be computed as the scene changes, so that they can be displayed.

7.3 Positioning the Camera

In three-dimensions, the position of the camera (the eyepoint and the viewing direction) is important both during the modelling process and as part of producing illustrations from the model. While it is often more important to move the eyepoint rapidly than precisely, there are times when precise placement is desirable. For instance, it is useful to be able to center a point on the screen, or to move 90 degrees from a top view to a front view. Snap-dragging can be extended to place the eyepoint precisely in a manner similar to that used to place scene objects precisely. Some prototype techniques are being tested.

For example, the Gargoyle3D user can activate a mode, "Move Camera." In this mode, interactive translation and rotation work as they usually do during the set-up and interaction phases. However, during the completion phase, instead of moving the selected objects to their new positions, Gargoyle3D moves the camera so that the selected objects will appear as they did at the end of the interaction phase. In other words, if you want an object to appear at a certain place on the screen, you drag it until it appears there; the camera is then moved so that the object appears at the desired screen position. The other objects, which were stationary during the interactive transformation, also appear to move since the camera has moved. As usual, gravity is active during these transformations, so the camera can be moved by precise distances and angles.

7.4 Improving the User Interface

Input from users, in various forms, is being used to motivate a new design for the user interface. A small group of researchers interested in improving Gargoyle have begun to use videotape to perform some simple human factors experiments. Experienced Gargoyle users are asked to perform a set of construction tasks. The camera records the screen, the keyboard, the mouse, and the face of the subject. All user actions are also recorded on Gargoyle session logs at the same time. Videotaping appears to be a rapid way to discover flaws in the user interface.

For instance, during our first videotaping session, we discovered that most subjects would accidentally invoke the "Add Constrained Rectangle" operation when they had intended to

invoke the "Select Objects In Box" operation. Curious, we examined the keystrokes that are involved. Both operations require placing the caret to indicate the first corner of a rectangle. The "Caret Placement" command is invoked by clicking the leftmost mouse button with the SHIFT key held down. Both commands require clicking the rightmost mouse button twice. "Add Constrained Rectangle" is invoked if the SHIFT key is down. "Select Objects In Box" is invoked if the SHIFT key is up. The sequence of steps – releasing the SHIFT key, placing a finger on the right mouse button and clicking it twice – is apparently too complicated. This and other aspects of the user interface are undergoing redesign.

Also, the videotapes supplement the operation counts mentioned in section 6.3 to provide information about where Gargoyle users actually spend time. Many user actions that do not show up in operation counts, such as time for thinking, correcting a mistake, or experimenting with different ways to achieve a construction, do show up in videotapes.

8. Summary

"Tut, tut, child!" said the Duchess. "Everything's got a moral, if only you can find it."

– Lewis Carroll
Alice in Wonderland

To build a precise geometric scene with an interactive computer program, a designer must communicate to the program how vertices, control points, and entire shapes are to be placed relative to each other. Productive scene composition is achieved when this communication is rapid and effective. A new interactive technique, snap-dragging, is presented that facilitates this communication by taking advantage of raster graphics, a two-dimensional pointing device, and the computational power of a high-performance workstation. Snap-dragging allows objects to be edited in both two and three dimensions. In three dimensions, it works in a single perspective view.

Previous illustration programs have incorporated several methods for facilitating communication between man and machine. Grid-based systems reduce precise point specification to a matter of using a pointing device to choose one point from a finite set of points. Constraint systems allow geometric relationships to be described in familiar terms such as "parallel" or "sloping at 30 degrees" and deduce for the user the geometry that the constraints determine. Drafting systems construct shapes in a step-by-step fashion, where lines, circles, and their intersections act as intermediate steps. Dial systems generate many frames per second of smooth motion, taking advantage of the user's familiarity with the continuous motions of real-world objects.

Snap-dragging is related in part to each of the previous techniques. It allows point positions to be specified precisely with a pointing device, allows relationships to be expressed in familiar terms, constructs shapes in a step-by-step ruler and compass fashion, and displays motions smoothly when objects are translated, rotated, scaled, or skewed.

Snap-dragging is the combination of three sub-techniques – gravity, interactive transformations, and alignment objects. The gravity function, *MultiMap*, arbitrates between points, curves, surfaces, and their intersections. The interactive transformations – translation, rotation, scaling, and skewing – are coupled to the motion of the software cursor. Because this cursor can be

precisely placed by gravity, the transformations can, in turn, be performed precisely. *Alignment objects*, consisting of lines, circles, planes, and spheres, provide a customized "grid", on which the software cursor can be placed. To create alignment objects, the user specifies a set of vertices and edges, called *hot* parts, and a set of slopes, angles, and distances, called *alignment values*. The system constructs alignment objects of the selected values at each of the hot parts. It can also generate hot vertices and edges automatically through the action of an *automatic hotness rule*. During all of the snap-dragging operations, graphical feedback is provided to highlight the object that the cursor is snapping to, the selected scene shapes, and the positions of the active alignment objects.

The software cursor position is the chief source of cooperation between snap-dragging operations. Such operations as creating a new line segment, placing a center of rotation, and transforming shapes are all coupled to the cursor position. Furthermore, the main function of gravity and alignment objects is to place the cursor. As a result, an amplification occurs; all of the operations that use the position of the software cursor take advantage of all of the operations that precisely set that position. Furthermore, because snap-dragging places *points* precisely, it can be used to edit the large variety of shapes that can be defined in terms of control points, including spline curves, spline surfaces, conics, and quadrics.

The three snap-dragging sub-techniques can be implemented efficiently. The gravity routine, MultiMap, quickly rejects objects that are far from the cursor and computes intersection points on the fly to achieve simple code and good performance. The current sets of triggers, gravity-active scene objects, and alignment objects are represented in three data structures that are updated incrementally. Finally, incremental screen refresh is used during interactive transformations to provide rapid smooth motion.

Productive scene composition is achieved by three strategies: reducing the number of operations required to build a shape, reducing the average time per operation, and encouraging re-use and sharing of previously designed shapes. To reduce operation count, snap-dragging allows the user to place shapes at precise positions as they are created. In this way, sketching and constraining can occur simultaneously. Furthermore, snap-dragging computes multiple alignment objects in response to a single command. To reduce the time needed for a user to invoke a command, it provides simple effective feedback, and computes intersection points without being

commanded to do so. Finally, snap-dragging encourages re-use of geometry. It relies on little state other than the shapes of the objects under construction, so a user can easily understand a shape created by someone else, and it allows shapes to be edited precisely even after they have been translated, rotated, scaled or skewed.

Snap-dragging and constraint-based systems differ in the set of scene composition applications that they support productively. Both techniques can describe a rich set of precise shapes. However, in constraint-based systems, the user invests time in building up a constraint network that determines both the final shape of the constrained objects, and how they will behave if one or more constraints are changed. This investment pays off in applications that call for changing the resulting shapes in a constrained fashion. In snap-dragging, the user describes shapes without any additional effort to specify their future behavior. This approach pays off when the application calls for producing a single illustration quickly.

Experiences with users confirm that snap-dragging can be productively used in real applications. Gargoyle's users have made frequent use of gravity, alignment objects, and interactive transformations while producing over a thousand illustrations for technical writing, business, fine art, and other applications. Most users were able to learn the technique by reading a tutorial or by trial and error. Some users have had difficulty learning to use the current implementation of snap-dragging or have had difficulty getting predictable results. Studies involving videotaping subjects are in progress to determine the causes of these difficulties and to suggest improvements.

Currently, snap-dragging is being extended to more effectively support the production of illustrations. Extensions include new types of alignment objects, operations for editing symmetrical objects, and operations that modify the three-dimensional viewing position.

In the future, snap-dragging may be applied in domains other than illustration. For example, because it works in both two and three dimensions, snap-dragging could be used to enter initial shape descriptions in both the drafting and solid-modeling phases of computer-aided design and manufacturing. The resulting shapes could be modified using constraints or another approach tailored to mechanical design. Snap-dragging could also be used as an educational tool. Constructions such as finding the largest square that fits in a hexagon (Figure 3-52) can be discovered in an intuitive manner by using smooth motion transformations and alignment objects

to try out configurations of shapes and to discover geometric theorems.

Interactive scene composition is a difficult problem. The user must translate a geometric concept into a sequence of commands and convey these commands to the computer using three limited devices: a screen of limited size, a pointing device of limited resolution, and a keyboard that is limited in speed by the user's fingers. As the session progresses, the user must keep track of all of the information that constitutes the state of the man-machine dialog. Snap-dragging facilitates interactive scene composition in several ways. It uses the familiar notions of slope, angle, and distance as the basic concepts with which the user can describe precise relationships. It graphically shows the chief elements of the session state including the constructed shapes, the selected shape parts, and the alignment objects. It uses a gravity function to map coarse mouse motions onto precise scene positions. Finally, it uses the full power of a modern workstation to compute alignment objects and intersection points so the user can specify a desired scene position with a single pointing action. By combining these techniques, snap-dragging makes it possible to interactively compose many precise geometric scenes more easily and more rapidly than was possible with previous techniques.

References

[Adobe87] Adobe Systems Inc. *Adobe Illustrator^{TM} User's Manual.* Adobe Systems Inc., 1870 Embarcadero Rd., Palo Alto, CA 94303, 1987.

[Atkinson88] Russell R. Atkinson, personal communication, 1988.

[Baudelaire79] Patrick C. Baudelaire. Draw. In the *Alto User's Handbook*, Xerox Palo Alto Research Center, 3333 Coyote Hill Rd., Palo Alto, CA 94304, 1979.

[Baudelaire80] Patrick Baudelaire and Maureen Stone. Techniques for interactive raster graphics. SIGGRAPH'80 proceedings, *Computer Graphics*, Vol. 14, No. 3, 1980, pp. 314-320.

[Bhushan86] Abhay Bhushan and Michael Plass. The Interpress page and document description language. *IEEE Computer*, Vol. 19, No. 6, June 1986, pp. 72-77.

[Bier83] Eric A. Bier. Solidviews: an interactive three-dimensional illustrator. Master's Thesis. MIT EECS, May 1983.

[Bier86] Eric A. Bier and Maureen C. Stone. Snap-dragging. SIGGRAPH'86 proceedings, *Computer Graphics*, Vol. 20, No. 4, 1986, pp. 233-240.

[Bier87] Eric A. Bier. Skitters and jacks: interactive 3d positioning tools. In *Proceedings of the 1986 Workshop on Interactive 3D Graphics* (Chapel Hill, NC, October 23-24, 1986), ACM, New York, 1987, pp. 183-196.

[Borning79] Alan Borning. *A Constraint-Oriented Simulation Laboratory.* Xerox PARC, Palo Alto, CA 94304, July 1979. Xerox Report SSL-79-3, Stanford CS Dept. Report STAN-CS-79-746.

[Boyse82] John W. Boyse and Jack E. Gilchrist. GMSolid: Interactive modeling for design and analysis of solids. *IEEE Computer Graphics and Applications*, Vol. 2, No. 2, March 1982, pp. 27-41.

[Chen88] Michael Chen, S. Joy Mountford, and Abigail Sellen. A study in interactive 3-D rotation using 2D control devices. To appear in SIGGRAPH'88 proceedings, *Computer Graphics*, 1988.

[Chyz85] George Walter Chyz. Constraint management for constructive geometry. Master's thesis. MIT Mechanical Engineering, 1985.

[Congdon82] Robert M. Congdon. Graphic input of solid models. Master's thesis, MIT Mechanical Engineering, February 1982.

[Elliot78] W. S. Elliot. Interactive graphical CAD in mechanical engineering design. *Computer-Aided Design*, Vol. 10, No. 2, March 1978, pp. 91-100.

[Ellis83] Andrew E. Ellis. An advanced user interface for the layout phase of design. Master's thesis. MIT Mechanical Engineering, November 1983.

[Faux79] I. D. Faux and M. J. Pratt. *Computational Geometry for Design and Manufacture*, chapter 5. Ellis Horwood Limited, 1979. Distributed by John Wiley and Sons.

[Fitzgerald81] William Fitzgerald, Franklin Gracer, and Robert Wolfe. GRIN: Interactive graphics for modeling solids. *IBM Journal of Research and Development*, Vol. 25, No. 4, July 1981, pp. 281-294.

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

[Goldberg83] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Hillyard78] R. C. Hillyard and I. C. Braid. Analysis of dimensions and tolerances in computer-aided mechanical design. *Computer-Aided Design*, Vol. 10, June 1978.

[Jensen74] Kathleen Jensen and Niklaus Wirth. *Pascal: User Manual and Report.* Springer-Verlag, second edition 1974.

[Johnson63a] Timothy E. Johnson. Sketchpad III, a computer program for drawing in three dimensions. In *Tutorial and Selected Readings in Interactive Computer Graphics*, ed. Herbert Freeman, IEEE Computer Society, Silver Spring, MD, 1984, pp. 20-26. Reprinted from AFIPS 1963.

[Johnson63b] Timothy E. Johnson. Sketchpad III, three-dimensional graphical communication with a digital computer. Master's thesis. MIT Mechanical Engineering, 1963.

[Kasik82] David J. Kasik. A user interface management system. SIGGRAPH'82 proceedings, *Computer Graphics*, Vol. 16, No. 3, July 1982, pp. 99-106.

[Lampson81] Butler W. Lampson and Kenneth A. Pier. A processor for a high-performance personal computer. *Proceedings of the 7th Symposium on Computer Architecture*, SigArch/ IEEE, May 1980, pp. 146-160.

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

[Light80] R. A. Light. Symbolic dimensioning in computer-aided design. Master's thesis. MIT Mechanical Engineering, February 1980.

[Light82] Robert Light and David Gossard. Modifications of geometric models through variational geometry. *Computer-Aided Design*, Vol. 14, No. 4, July 1982, pp. 209-214.

[Lin81a] Vincent C. Lin. Variational geometry in computer-aided design. Master's thesis. MIT Mechanical Engineering, May 1981.

[Lin81b] V. C. Lin, D. C. Gossard, and R. A. Light. Variational geometry in computer-aided design. SIGGRAPH'81 Proceedings, *Computer Graphics*, Vol. 15, No. 3, August 1981, pp. 171-177.

[Lubow87] Martha Lubow. *Working out with AutoCAD*$^{(R)}$. New Riders Publishing, P.O. Box 4846, Thousand Oaks, CA 91360, 1987.

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

[Mitchell79] James G. Mitchell, William Maybury, and Richard Sweet. *Mesa Language Manual.* CSL-79-3, Xerox Palo Alto Research Center, 3333 Coyote Hill Rd., Palo Alto, CA 94304.

[Nelson85] Greg Nelson. Juno, a constraint-based graphics system. SIGGRAPH'85 Proceedings, *Computer Graphics*, Vol. 19, No. 3, 1985, pp. 235-243.

[Newell85] From a private communication with Martin Newell, at CIMLINC in Menlo Park, CA, 1985.

[Newman79] William M. Newman and Robert F. Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill, New York, 1979.

[Nielson87] Gregory M. Nielson and Dan R. Olsen Jr. Direct manipulation techniques for 3D objects using 2D locator devices. In *Proceedings of the 1986 Workshop on Interactive 3D Graphics* (Chapel Hill, NC, October 23-24, 1986), ACM, New York, 1987, pp. 175-182.

[Nievergelt84] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: an adaptable, symmetric multi-key file structure. *ACM Transactions on Database Systems*, Vol. 9, No. 1, March 1984, pp. 38-71.

[O'Donnell81] T. J. O'Donnell and Arthur J. Olson. GRAMPS -- A graphics language interpreter for real-time interactive three-dimensional picture editing and animation. SIGGRAPH'81 proceedings, *Computer Graphics*, Vol. 15, No. 3, August 1981, pp. 133-142.

[Opperman84] Mark Opperman. A Gremlin tutorial for the SUN workstation. Internal document. EECS Department, UC Berkeley, Berkeley CA 94720, 1984.

[Parent77] Richard E. Parent. A system for sculpting 3-D data. SIGGRAPH'77 proceedings, *Computer Graphics*, Vol. 11, No. 2, 1977, pp. 138-147.

[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, June 1983, pp. 252-269.

[Pier88] Kenneth A. Pier, Eric A. Bier, and Maureen C. Stone. An introduction to Gargoyle: An interactive illustration tool. To appear in *Proceedings of EP88, International Conference on Electronic Publishing, Document Manipulation, and Typography*, April 1988.

[Richardson87] Rick Richardson. Dhrystone 1.1 benchmark summary (and program text) informal distribution via "Usenet." Based on the version of September 21, 1987.

[Roth82] Scott D. Roth. Ray casting for modeling solids. *Computer Graphics and Image Processing*, Vol. 18, 1982, pp. 109-144.

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

[Siegel86] H. B. Siegel. Jessie: An interactive editor for unigrafix. U.C. Berkeley Electrical Engineering and Computer Science Department, Computer Science Division Report No. UCB/CSD 86/279, 1986.

[Sutherland63a] Ivan Sutherland. *Sketchpad, A Man-Machine Graphical Communication System.* Ph.D. dissertation, MIT. MIT Lincoln Laboratory Technical Report No. 296, Lexington MA.

[Sutherland63b] Ivan E. Sutherland. Sketchpad, a man-machine graphical communication system. In *Tutorial and Selected Readings in Interactive Computer Graphics*, ed. Herbert Freeman, IEEE Computer Society, Silver Spring, MD, 1984, pp. 2-19. Reprinted from AFIPS 1963.

[Swinehart86] D. Swinehart, P. Zellweger, R. Beach, and R. Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 4, 1986, pp. 419-490.

[Tamminen81] Markku Tamminen. The EXCELL method for efficient geometric access to data. Acta Polytechnica Scandinavica, Mathematics and Computer Science Series No. 34, Helsinki, 1981.

[Upstill85] Steve Upstill, Tony DeRose, and John Gross. SCOT: Scene Composition Tool, CS-Technical Report, U.C. Berkeley Computer Science Division, December 1985.

[VanWyk82] Christopher J. Van Wyk, "A High-Level Language for Specifying Pictures," ACM Transactions on Graphics, Vol. 1, No. 2, April 1982, pp. 163-182.

[Weicker84] Reinhold P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, Vol. 27, No. 10, Association for Computing Machinery, 1984, pp. 1013-1030.

[Weyl52] Hermann Weyl. *Symmetry*. Princeton University Press, Princeton, New Jersey, 1952.

[Wolfe81] R. Wolfe, W. Fitzgerald, and F. Gracer. Interactive graphics for volume modeling. *IEEE 18th Design Automation Conference*, 1981, pp. 463-470.

[Xerox88] Xerox Corp. *Xerox Pro Illustrator Reference Manual.* Xerox Document Systems Business Unit, 475 Oakmead Parkway, Sunnyvale, CA 94086, 1988. (In preparation)

Appendix A. Gravity Algorithm Details

As described in Chapter 4, MultiMap is built on top of three routines, FacesBehind, PointsInNeighborhood, and CurvesInNeighborhood. FacesBehind uses ray casting to compute an ordered list of the surfaces that are directly behind the cursor. For a description of ray casting see Roth's helpful paper [Roth82]. PointsInNeighborhood and CurvesInNeighborhood are described below.

PointsInNeighborhood finds some of the closest vertices, control points, and intersection points that are within tolerance distance t of the cursor **q**, and returns them in sorted order from nearest to farthest. In particular, it finds the nearest such point o_{min} at some distance d_{min} from **q** and all points that are no more than $d_{min}+s$ from **q**, where s is on the order of floating point round-off, up to a total of m points. For Gargoyle, m is 20. If there are more than m points all nearly closest to **q**, only the nearest 20 are returned. It is possible to implement PointsInNeighborhood with variable length arrays or lists, so that m is effectively infinite. However, finite arrays simplify efficient implementation with little cost to the user.

PointsInNeighborhood examines all of the points in the scene in the order in which they appear in the data structures. Since this ordering has nothing to do with their distance from **q**, this order is effectively a random order for our purposes. All points that are farther from **q** than t are quickly ruled out. The rest of PointsInNeighborhood fills an array h of length m with up to m of the closest points to **q** that are within distance $d_{min}+s$ of **q**.

Ignoring intersection points for the moment, PointsInNeighborhood begins as depicted in Figure A-1(a). Let d_{tol} be the maximum distance that a point may be from **q** and still be worth considering. Initially, $d_{tol} = t$. Let d_{min} be the distance of the nearest point found so far. Initially, $d_{min} = \infty$. Throughout the algorithm, $d_{tol} = \text{MIN}[t, d_{min}+s]$, where MIN is a function that returns the smaller of its two arguments. As each new point is considered, one of three actions is performed: (1) If the new point is closer than any point considered so far, d_{min} becomes the distance of that point from **q**, and d_{tol} is recomputed as just described. The new point is added to h, throwing out the worst point in h, if necessary. Figure A-1(b) shows the results after point A is encountered. (2) If the new point is not the closest but is within d_{tol} of **q**, it is added to h, if h is not full. If h is full, it is added to h if it is better than the worst point in h, by throwing out the

worst point. (3) If the new point is not within d_{tol} of **q**, it is ignored.

Once all of the points have been considered, d_{min} will be the distance from **q** of the nearest point, and h will contain up to m of the nearest points to **q** that are within d_{tol}. Array h may also contain some points farther than d_{tol} from **q**—points that were added to h before the minimum value of d_{min} was discovered. Figure A-1(c) shows the final circles of radius d_{min} and d_{tol}. If the points are considered in alphabetical order, A, B, D, and E will be in h at this stage. A, D, and E were added to h by case (1), B was added by case (2), and C was rejected by case (3).

Once h has been computed, the values in h are sorted. During sorting, all values greater than d_{tol} are discarded. For the example of Figure A-1, the list (E, D) would be returned.

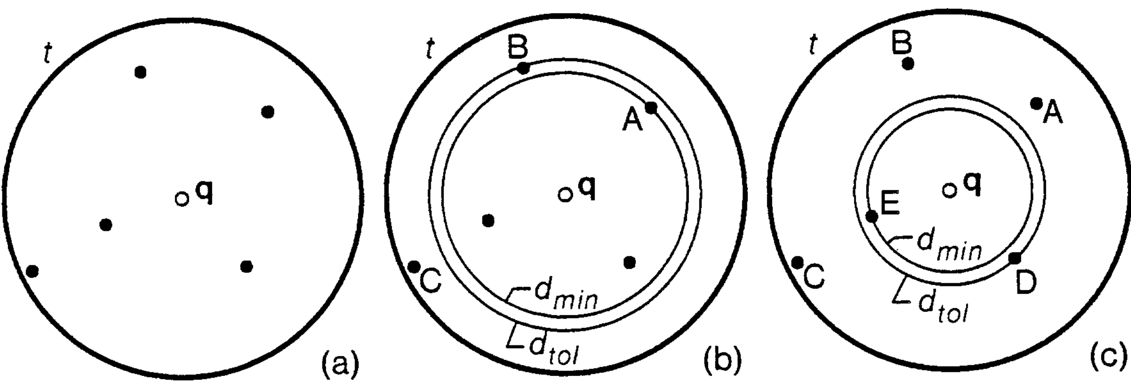


Figure A-1. PointsInNeighborhood. (a) Consider all points within distance t of q. (b) Once A is found, C is no longer of interest, but B is. (c) At the end, the nearest point, E, has been found, and D and E are returned.

For small values of t, PointsInNeighborhood is more complicated than necessary. It would suffice, for instance, to compute PointsInNeighborhood in two passes. The first pass would compute d_{min} and the second pass would extract all points that were within d_{tol} of **q**. However, for large values of t (strong gravity functions) or cluttered scene regions, keeping track of d_{tol} as it shrinks allows large numbers of points to be rejected cheaply. For instance, if the bounding box of a shape does not intersect with the d_{tol} circle, all of the vertices and control points of that shape can be rejected without further consideration.

PointsInNeighborhood computes intersection points and adds them to h using the same criteria used for vertices and control points. For intersection points, keeping track of d_{tol} can result in dramatic performance improvements. In fact, for the grid example of section 4.1.6 less

than two intersection points are calculated on average for each call to MultiMap.

When intersection points are to be computed, PointsInNeighborhood is passed, as an argument, a description of all of the curves that are within distance r of q, where r is the points-preferred distance described in section 4.1, $r \leq t$. Furthermore, these curves are sorted from nearest to farthest. Basically, we iterate through all pairs of curves, considering near curves first, computing intersection points. However, if a curve is farther than d_{tol} from **q**, then its intersection point(s) with any other curve must also be farther from **q** than d_{tol}. Thus, a curve farther than d_{tol} need not be considered. Often, the intersection point of the two nearest curves is the nearest intersection point. Thus d_{tol} often reaches its smallest value (ignoring vertices and control points for the moment) after the first intersection computation is performed. This does not necessarily rule out other intersection tests, as shown in Figure A-2. However, it reduces the number of intersections dramatically.

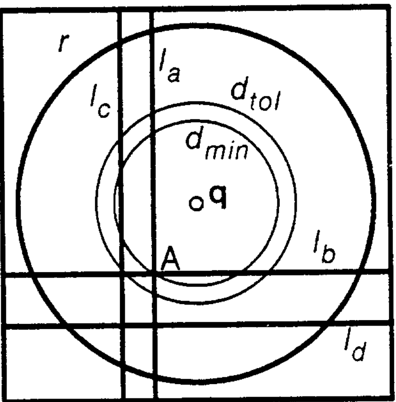


Figure A-2. Although the intersection point, A, of line l_a and line l_b is the closest intersection point to q, line l_c cannot be quickly rejected, as it is still within d_{tol} of q. l_d, however, can be rejected.

In effect, once we have the ability to compute the nearest curves to **q**, intersection points are free.

In pseudo-code, the complete algorithm for PointsInNeighborhood is described below (Algorithm A-1). In addition to returning h as described above, PointsInNeighborhood returns a Boolean value, overflow, which is TRUE if and only if h contains m values that are within d_{tol} of **q** and there are one or more values, within d_{tol} of **q** that are not included in h. The parameter nearCurves is a set of curves, previously computed by CurvesInNeighborhood, all of which are within r of **q**. N is the number of curves in nearCurves. The Boolean parameter, intersections, is

TRUE if PointsInNeighborhood should compute intersections.

The real-valued variables $d_{\min}$ and d_{max}, describe the distances of the closest and farthest objects currently stored in h. Integer variable maxIndex is the index in h of the farthest object. If h is not full, size is the integer index of the next empty cell of h. Real-valued variable *bestTossed* describes the distance from **q** of the nearest object that was seen by AddNeighbor but not included in h, where AddNeighbor is described below.

Note: "←" is the assignment operator.

```
Algorithm: PointsInNeighborhood

    input: nearCurves, alignBag, sceneBag, q, t, intersections;
    output: h, overflow;

    size ← 0; d_max ← 0; maxIndex ← -1; d_min ← ∞; d_tol ← t; bestTossed ← ∞;
    for all entries i of h do h(i).dist ← ∞; h(i).object ← Ø; end loop on entries of h;
    overflow ← FALSE;

    {Compute the intersection points.}
    if intersections=TRUE then begin
        for each curve i in nearCurves (i goes from 1 to N) do
            if curve i is farther than d_tol from q
                then exit this loop; {All higher-numbered curves will be even farther from q}
            for each curve j in nearCurves (j goes from 1 to i) do
                if curve j is farther than d_tol from q then exit this inner loop;
                compute the intersection points of curve i with curve j;
                for each intersection point, p, of curve i with curve j do
                    create an object, o, to represent the intersection at p;
                    if p is within d_tol of q, call AddNeighbor to (try to) add o to h;
                    {AddNeighbor will update d_tol, overflow, and other variables.}
                    end loop on intersection points;
                end loop on curves j;
            end loop on curves i;
        end;

    {Process the vertices and control points.}
    for each object, o, in the scene bag do
        if the bounding box of the object, offset by d_tol, does not contain q then ignore o;
        otherwise, find the vertex or control point, p, of o that is closest to q;
        if p is within d_tol of q, call AddNeighbor to (try to) add o to h;
        {AddNeighbor will update d_tol, overflow, and other variables.}
        end loop on objects;

    sort the elements of h, removing entries that are farther than d_tol from q;
    return the contents of h and the final value of overflow;
```

Algorithm A-1. PointsInNeighborhood.

The PointsInNeighborhood and CurvesInNeighborhood algorithms both use a routine AddNeighbor in their inner loops to place points in h. AddNeighbor works a little differently when used from PointsInNeighborhood than when used from CurvesInNeighborhood. The parameter, isCurve, is TRUE if AddNeighbor is being called from CurvesInNeighborhood and FALSE otherwise. AddNeighbor is described in Algorithm A-2.

```
Algorithm: AddNeighbor

    input: o, h, isCurve;

    Let d be the distance of o from q;
    If d > d_tol, goto Toss;
    If h is not yet full, goto Add;
    If h is full, goto ReplaceOrOverflow;

    Toss:   bestTossed ← MIN[bestTossed, d];
            if bestTossed ≤ d_tol then overflow ← TRUE else overflow ← FALSE;
            return from AddNeighbor;
    Add:    h(size) ← o;
            If d > d_max THEN {d_max ← d; maxIndex ← size};
            d_min ← MIN[d_min, d];
            size ← size + 1;
            if isCurve then d_tol ← MAX[d_min + s, r] else d_tol ← d_min + s;
            return from AddNeighbor;
    ReplaceOrOverflow:
            if d ≥ d_max then goto Toss;
            h(maxIndex) ← o;
            bestTossed ← MIN[bestTossed, d_max];
            d_min ← MIN[d_min, d];
            if isCurve then d_tol ← MAX[d_min + s, r] else d_tol ← d_min + s;
            if bestTossed ≤ d_tol then overflow ← TRUE else overflow ← FALSE;
            scan through h to find the new farthest object from q, o_max;
            update d_max to be the distance of o_max from q;
            update maxIndex to be the index of o_max in h;
            return from AddNeighbor;
```

Algorithm A-2. AddNeighbor.

CurvesInNeighborhood finds some of the closest curve segments that are within distance t of the cursor point $\mathbf{q}$ in sorted order from nearest to farthest. It is very similar to PointsInNeighborhood in control structure. However, there are two major differences. First, CurvesInNeighborhood looks at curves instead of points. Second, no matter how low d_{min} becomes, d_{tol} is never allowed to go below r. Thus, CurvesInNeighborhood is guaranteed to find all curves within r of $\mathbf{q}$ (up to m), even if some of these curves are not within $d_{min}+s$ of $\mathbf{q}$. This way, CurvesInNeighborhood returns all of the curves needed to compute intersections in PointsInNeighborhood. CurvesInNeighborhood is described in Algorithm A-3.

```
Algorithm: CurvesInNeighborhood

    input: alignBag, sceneBag, q, t, r;
    output: h, overflow;

    size ← 0; d_max ← 0; maxIndex ← -1; d_min ← ∞; d_tol ← t; bestTossed ← ∞;
    for all entries i of h do h(i).dist ← ∞; h(i).object ← ∅; end loop on entries of h;
    overflow ← FALSE;

    for each object, o, in the alignBag or sceneBag do
        if the bounding box of o, offset by d_tol, does not contain q then ignore o;
        otherwise, find the point, p, on a segment of o, that is closest to q;
        if p is within d_tol of q, call AddNeighbor to (try to) add o to h;
        {AddNeighbor will update d_tol, overflow, and other variables.}
        end loop on objects.

    sort the elements of h, removing entries that are farther than d_tol from q;
    return the contents of h and the final value of overflow;
```

Algorithm A-3. CurvesInNeighborhood.

Using finite arrays h of length m has several positive effects on performance. The sorting done in PointsInNeighborhood and CurvesInNeighborhood is never done on more than m elements. Likewise, the intersection computations in PointsInNeighborhood need never be done on more than $m(m-1)/2$ pairs of curves. Finally, it is not necessary to dynamically allocate storage for h. However, using a finite array does have several costs.

First, in very cluttered picture regions, only the nearest m objects are returned. If the user tries to use cycling selection in such a region and the desired object is not one of the nearest m objects, he will not be able to select it. This is not a particularly disturbing shortfall, however, because with a reasonably large m, the user probably won't want to cycle through more than m objects anyway. Zooming in on the picture to point more precisely at the object is more practical.

Second, also in cluttered picture regions, MultiMap may fail to compute the nearest intersection point to the cursor. For instance, in Figure A-3, there are many parallel lines near **q**. If there are more than m of these lines, CurvesInNeighborhood will not return the two lines that intersect, so the intersection will not be computed. However, if the user moves the cursor closer to the intersection point, the intersecting lines will be returned by CurvesInNeighborhood and all will be

well. Since users are accustomed to moving the cursor closer to an object until the caret snaps to it, this failing also is easy to ignore.

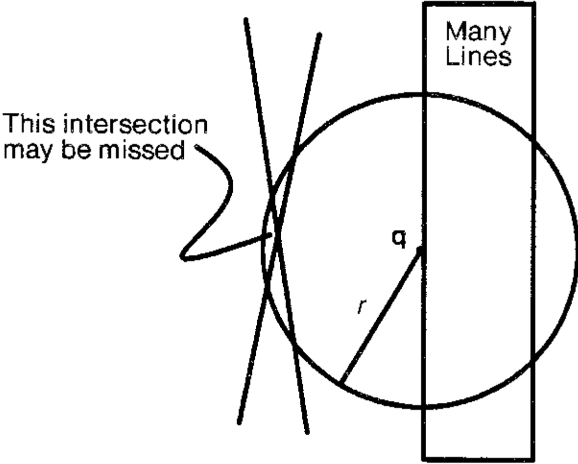


Figure A-3. If many lines are near the cursor, but don't intersect near the cursor, the true nearest intersection may be missed.