Snap-Dragging in Three Dimensions

Eric A. Bier
Xerox PARC
3333 Coyote Hill Rd.
Palo Alto, CA 94304
bier.parc@xerox.com

Abstract: A large portion of the user interface in interactive solid modeling systems is devoted to the problem of placing and orienting objects in three dimensions. In particular, many operations must be provided for selecting control points, curves and surfaces, and for translating, rotating and scaling scene components into precise relationships with other scene components. By factoring these operations carefully, it is possible to provide the desired functionality so as to reduce both the size of the user interface and the time that it takes to use it. With snap-dragging, the user takes advantage of three main elements that work together: a general-purpose gravity function, alignment objects that can be created many at a time, and smooth-motion affine transformations. Scene composition is achieved in a single perspective view using a mouse and keyboard. With 19 mouse commands, 15 keyboard commands, 5 menus of numbers, and 1 single-level menu of numerical transformations, this user interface has fewer commands and requires fewer keystrokes than the skitters and jacks technique reported earlier.

CR Categories and Subject Descriptors: I.3.6 [**Computer Graphics**]: Methodology and Techniques – Interaction techniques

Additional Keywords: Scene composition, interactive design, geometric construction, constraint systems

1. Introduction

High-powered workstations make it possible to significantly improve the user interface for composing three-dimensional scenes. This paper describes a technique for precisely placing points and objects in a three-dimensional scene. This technique, an extension of snap-dragging to three dimensions, greatly reduces the number and complexity of commands that are needed to position scene objects precisely. It requires no input devices other than a keyboard and mouse and is compatible with two-dimensional snap-dragging, so users can quickly go back and forth between editing two-dimensional shapes, such as curves to be extruded, and true three-dimensional shapes. Snap-dragging can be used to compose and shape polyhedra, quadrics, spline patches, and any other surface type that can be defined in terms of control points.

Snap-dragging is the combination of three interactive techniques that work well together: gravity, alignment objects and interactive transformations [Bier86a, Bier88]. The gravity function enables a three-dimensional cursor, called the *snap-dragging skitter*, to snap to points, curves, and surfaces in the scene. A set of alignment objects (lines, planes, and spheres) can be constructed at object vertices and control points, providing ruler-and-compass style constructions in three dimensions. The skitter can be snapped to these alignment objects and their points and curves of intersection as well as to scene objects. Finally, interactive transformations (translation, rotation, and scaling) track the motion of the skitter, which continues to snap to objects during transformations allowing precise transformations to be applied to scene objects.

All of the components of snap-dragging – gravity, alignment objects, and smooth motion transformations – have appeared in previous systems (see section 2). Snap-dragging improves on these techniques individually and provides a new way to factor the user interface to reduce both interface complexity and the average time required to perform a construction. In particular, snap-dragging makes these improvements:

(1) Other systems have a picking mode that snaps to vertices, another mode that snaps to edges, and a third mode that snaps to faces. This makes it essential to switch modes frequently in the course of normal operation. Snap-dragging also has three

modes, but the functionality is distributed differently so that one mode can be used most of the time, with the other modes being used only when the first mode fails.

(2) A single menu selection can result in the creation of many alignment objects, both immediately and in response to future construction operators. Because several of these objects may be useful for a given construction the user needs less than one mouse click on average per useful alignment object created.

(3) All of the smooth-motion transformations follow the skitter and can be given precise end conditions using gravity.

Snap-dragging achieves its economy of user interface by taking advantage of real-time feedback and by performing some computations, such as computing alignment objects and their intersections, automatically. As a result, the user can often achieve a construction by *choosing* (e.g., pointing to an intersection point) rather than by *describing* (e.g., asking for the intersection of two shapes to be computed).

A prototype three-dimensional snap-dragging system, Gargoyle3D, has been implemented in the Cedar programming language [Swinehart86], on the Dorado personal workstation [Pier83]. On the Dorado, smooth motion can only be achieved with small scenes displayed as wireframes. The author looks forward to the improved performance that will be available from current hardware.

The remainder of this paper describes the background, user interface, and implementation of snap-dragging as follows: Section 2 describes previous work. Sections 3, 4, and 5 describe the snap-dragging gravity functions, alignments objects and interactive transformations, giving examples to show how they are used. Section 6 lists all of the snap-dragging commands and describes those that are not described elsewhere. Section 7 presents some performance measurements. Section 8 presents my conclusions and plans for future work.

2. Previous Work

Many previous systems have addressed the problem of providing a user interface for scene composition that is easy to use. Constraint-based approaches achieve both precise affine transformations and precise point placement by solving simultaneously non-linear equations. Direct manipulation approaches have combined techniques such as gravity, alignment objects, and smooth motion transformations to make local changes to the picture. These direct manipulation systems differ in how this functionality is factored into primitives.

2.1 Constraint-Based Approaches

A wide variety of surface types can be shaped by control points, including polygons, quadric surfaces and spline patches (see Lin's paper on variational geometry [Lin81] for a description of point-parameterized solid models). One intriguing approach to positioning control points is to represent relationships as a set of constraint equations. Early two-dimensional constraint-based systems include Sketchpad and ThingLab [Sutherland63, Borning79].

An on-going project at the Computer Aided Design Laboratory of MIT's Mechanical Engineering Department has used this approach to design families of parameterized mechanical parts whose shape is determined by specified forces, torques and other design criteria [Light82, Serrano84]. Using constraints to position points requires investing time in creating and verifying a constraint network, a process that can be like debugging computer programs. Many constraints are needed; even a simple cube has 24 degrees of freedom. This investment will pay off for applications, such as mechanical engineering or animation, where a debugged constraint network will be used many times before it needs to be modified. However, to quickly produce a single solid model, this technique is too time-consuming.

Rossignac, in his constraint approach from constructive solid geometry gives a complete ordering to the constraints [Rossignac86]. This makes them easier to understand and debug. Other work has focused on reducing the number of constraints that must be entered by hand [Congdon82, Lee83, Chyz85]. However, the user must still understand and debug the constraint network that results (see [Bier88] for a more detailed analysis of constraint approaches). In addition, before constraints can be added, the unconstrained geometry must be described. A quick sketching technique, such as snap-dragging, could be useful for this first step.

2.2 Direct Manipulation Approaches

Another way to place points precisely is to extend drafting tools, such as ruler, compass, protractor, and T-square into three dimensions. This is done in the Jessie editor at UC Berkeley where commands are provided, for instance, to construct an alignment line given two selected points. Two-dimensional systems based on drafting techniques include Ellis's layout editor [Ellis83] and CIMLINC's CIMCAD drafting system. The problem with this approach is that it is tedious. It can require several keystrokes and pointing actions to create each alignment object.

Many geometric design systems use a *gravity function* to help the user place object points on vertices, edges, surfaces and their intersections. GRIN [Fitzgerald81], GMSolid [Boyse82], Jessie [Siegel86], and Sketchpad III [Johnson63] implement gravity functions that snap the cursor to lines and curves. Solidviews [Bier83, Bier86b] allows the cursor to be placed on object surfaces. Most existing gravity functions only snap the cursor to one type of scene object at a time (e.g., vertices or edges but not both), requiring the user to invoke mode-switching commands frequently.

Many interactive solid modeling systems provide affine transformations that vary smoothly, often in response to dials. These systems include Jessie [Siegel86], GRAMPS

[O'Donnell81], and Parent's sculpting technique [Parent77]. Robertson, Card and Mackinlay point out that when objects move smoothly, it requires less cognitive effort on the part of the user to determine which objects have moved and how [Robertson89]. Dials allow objects to be moved until they "look right" or until a desired object is visible but are not generally useful for precise motions.

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, Bier86b], GRIN [Fitzgerald81], and is described in Nielson's article on manipulation techniques [Nielson86].

If the points that are used to parameterize the transformations are just different positions of a cursor, then smooth dragging, cursor positioning, and transformation operations are unified. This technique is used in a number of modern two-dimensional illustrators including Adobe Illustrator [Adobe87] and Xerox Pro Illustrator [Xerox88]. Snap-dragging extends this idea to three dimensions.

One way to factor the direct manipulation operations is to provide a dialog for each operation. The user selects an operation (e.g., rotate). The system prompts the user for an operation sub-type (e.g., rotate about a coordinate axis, rotate about a line determined by two points, etc.) and then prompts the user, in turn, for each of the arguments. This approach was taken for instance in SCOT [Upstill85]. Another approach to factoring is skitters and jacks [Bier86b]. With this approach, all operations involve selecting the objects to be transformed, selecting a coordinate system (called a jack) relative which to translate, rotate, or scale, specifying the amount by which to transform by filling in a form, and clicking a menu button or using a mouse motion to perform the transformation. Jacks can be placed based at the skitter position. While skitters and jacks is a relatively compact interface for many common transformation operations, it has no uniform way to construct new points in free space, has an impoverished gravity function, does not allow individual control points to be moved, and, for most transformations, requires moving attention away from the scene area to use the menus.

3. Gravity

The snap-dragging gravity function must be computed several times per second so that the skitter moves smoothly as the mouse moves. As a result, the gravity function cannot consult the user to disambiguate its operation in cluttered scene regions; it must choose one point at which to place the skitter. Fortunately, if the gravity function chooses a point other than the desired one, the user can move the cursor until the correct skitter position is achieved. In cases where the desired point is obscured or in a hopelessly cluttered region, the user can cycle through the objects near the cursor line. Thus snap-dragging gravity computes not only a best point but also an ordered list of close points.

From the user's standpoint, the gravity algorithm compares distances on the screen rather than in 3-space. The algorithm determines which object point projects nearest to the mouse cursor, finds the corresponding point in three dimensions, and places the skitter at that point. It also determines the orientation of the skitter. The skitter's z axis is chosen to be perpendicular to the edge or face (if any) that the skitter snaps to, and the x axis is chosen to be tangent to the edge (if any) that the skitter snaps to. If the skitter snaps to a vertex, its axes are determined by the faces and edges that terminate at that vertex. The skitter's orientation becomes relevant when it is used to place the anchor (see Figure 9), which is in turn used as an axis of rotation.

The main gravity mode is *points-preferred*. In this mode, the skitter snaps to a vertex or point of intersection if one projects close to the mouse cursor. If not, the skitter will snap to a line or edge if one is close by. If not, the skitter will snap to a face if one is under the cursor. If not, the skitter will snap to a default plane. The other modes, *lines-preferred* and *faces-preferred*, are used when the user needs to point to an edge near, but not on, a vertex, or to a face near, but not on, an edge, respectively. With points-preferred and lines-preferred gravity functions, a vertex or edge that projects near the cursor is considered by gravity even if it is obscured by a face. This allows easy access to hidden components. Figure 1 shows points-preferred gravity being used to place the skitter on a vertex or a face (Figure 1(a)), and a rear-facing edge (Figure 1(b)). The dark circle is the mouse cursor. The shape with three mutually perpendicular axes is the skitter. Its z axis is given a triangle shape to make it distinctive.

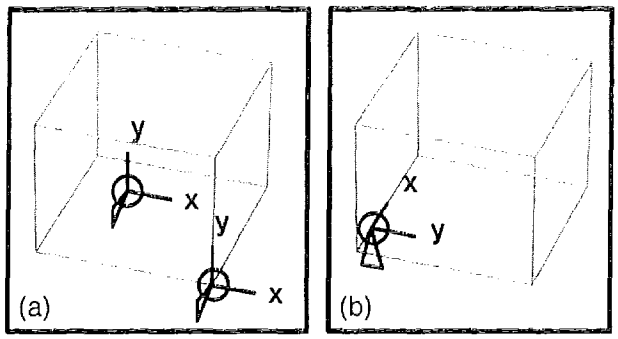

Figure 1. Points-preferred gravity. Snapping the skitter (a) to a vertex or to a face, (b) to a (back-facing) edge.

With more effort, the user can also point to obscured *faces*. The user points at the desired face point and clicks the mouse to initially place the skitter on the nearest face to the eye point. A keyboard command is then used to move the skitter back through the obscured faces until the desired face is reached.

Gravity operates during a variety of operations including operations to place the skitter, add new line segments, or apply smooth-motion affine transformations. In Figure 2, points-preferred gravity is used to place the new endpoints of a line segment with two mouse button clicks.

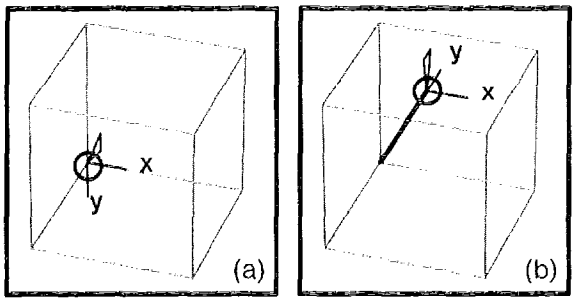

Figure 2. Adding a line. (a) Placing the first end on a vertex. (b) Rubber-banding the line segment to a face point.

The gravity function must compute the distance of the mouse cursor from the projections of faces, edges, and vertices. In the current implementation, all of these computations are actually performed in three dimensions to avoid projecting scene objects onto the screen. First, rays are cast from the eye point through the cursor to find all intersected faces. Next, for edges and alignment curves, the closest point, p, to the cursor ray and the distance of p to the cursor ray are computed. This distance is divided by the depth (z coordinate) of p relative to the eye point to approximate the distance of the curve's projection from the cursor. If this division is not performed, the user must place the cursor very close to the projection of distant objects in order to snap to them. Finally, the distance of the cursor ray to vertices and points of intersection is computed and divided by their depth. The gravity function combines the results of these computations and chooses one best face, edge or vertex to snap the skitter to. This choice depends on the current gravity mode.

Remarkably, the computation of intersection points that are near the cursor can be done on the fly at almost no cost and with a very simple routine; there is no need to precompute and store the pairwise intersections of curves with curves and curves with faces. Instead, these points are computed as needed in the process of computing the gravity mapping for a particular mouse cursor position. The algorithm relies on two observations:

(1) The intersection point of two curves will only occur close to the cursor ray if both curves pass near the cursor ray.

(2) Once an intersection point has been found that projects onto the screen within a distance t of the cursor, better intersection points can only come from two curves that project within distance t of the cursor.

Thus the gravity algorithm first finds the edges that project within a tolerance distance of the cursor. These edges are sorted by increasing distance from the cursor. Intersection computations are performed on the nearest pairs of edges first. As soon as an intersection point is found, all edges farther from the cursor than the intersection point are removed from further consideration.

The same trick can be used to reduce the number of vertices that are considered by the gravity algorithm. Instead of asking each object that is near the cursor to compute the distance from each of its vertices to the cursor, we ask only the edges that have been found to be close and perform this computation for their endpoints.

In general, snap-dragging spends its time not on vertices and intersection points, but on edges and faces. Ray-tracing and hit detection for edges can be computed quickly enough to allow several cursor updates per second on my Dorado workstation on scenes with a few hundred edges. Some performance figures appear in section 7.

4. Alignment Objects

The current implementation of snap-dragging includes three different types of alignment objects: lines, planes, and spheres. The user activates a set of alignment objects by choosing from an extensible menu of alignment values. Three factors make the alignment objects in snap-dragging particularly powerful: (1) each alignment value can trigger the construction of many (identically-shaped) alignment objects at the same time, (2) any combination of alignment values can be activated at once, and (3) intersection curves and intersection points are computed automatically and are gravity-active. This section shows how alignment objects are used and describes some of the details of their implementation.

The alignment menus are shown in Figure 3. A desired direction of alignment lines is activated by selecting an (azimuth, slope) pair from the row of items labelled "Line". Azimuth and slope are both angles in degrees. Likewise, a desired orientation of alignment planes is activated from

the "Plane" row, and a desired radius of alignment sphere from the "Radius" row. The "Line" and "Plane" menus initially contain values corresponding to the x, y and negative z axes of WORLD. The "Radius" menu initially contains a set of fractions and small integers, 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. The alignment menus in Gargoyle3D.

The user can extend the "Line" and "Plane" menus using two other menus: "Azimuth" and "Slope". 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 on the "Line" row. Pairs may be added to the "Plane" menu in a similar fashion. Figure 3 shows the alignment menus after lines of azimuth 10 and slope 0 have been added.

All of the alignment menus can also be extended by typing new values and by measuring values from the scene. After each mouse operation, Gargoyle3D measures the displacement vector through which the skitter has moved and reports the length and direction angles of this vector. Any of these values may be added to the alignment menus.

Adding alignment objects to the scene is accomplished in two steps. First, the user selects those scene objects that are to trigger alignment objects and issues a keyboard command to turn all of the selected vertices into alignment triggers, called "hot points." Hot points are drawn as large white squares, as shown in Figure 4. Next, the user activates a value from the alignment menus. At each hot point, the system constructs alignment objects of all currently active values. In Figure 4(a), the user has made all three vertices of an equilateral triangle hot and activated alignment spheres. The system constructs three spheres and their circles of intersection. By snapping to vertices and points of intersection, the user constructs the tetrahedron of Figure 4(b).

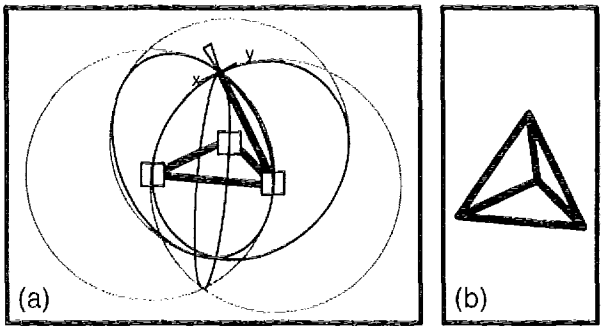

Figure 4. Tetrahedron construction. (a) A new line segment is stretched to a point where all three spheres meet. (b) The last two segments are snapped to existing vertices.

To avoid screen clutter, each of the alignment objects is drawn as a single thin gray curve, and intersection curves are displayed as thin black curves. Each sphere is shown by its silhouette circle, as shown in Figure 4. Alignment planes are shown as small gray squares, as shown in Figure 5. The black object that resembles a nautical anchor in this figure is the *anchor*, a special object whose center point is always hot. Alignment lines are drawn as thin gray lines (see Figure 6). When the user snaps the skitter to an alignment object, it becomes thick and black.

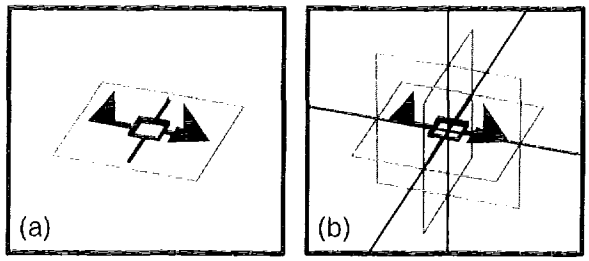

Figure 5. Alignment planes triggered by the anchor. (a) A horizontal alignment plane. (b) Three perpendicular alignment planes and their lines of intersection.

When the skitter snaps to an intersection point, Gargoyle3D highlights all of the alignment objects and/or the hot points that contributed to that intersection point. In Figure 6, the skitter is snapped to the intersection point of two alignment lines, each of which is triggered by two hot points. An asterisk is placed at all four hot points. In order to provide this feedback, Gargoyle3D must notice when two or more hot points generate the same alignment object. Such an alignment object is called a *duplicate*.

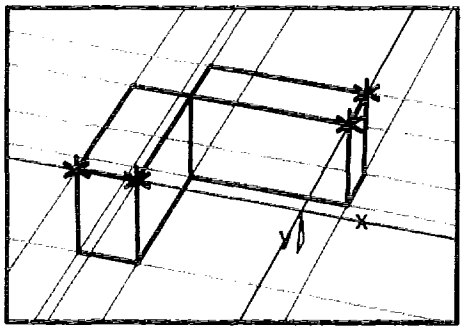

Figure 6. Snapping to an intersection point. (The white squares that indicate hotness are not displayed during interactive operations.)

Hidden line elimination is not performed on alignment objects, even when it is performed on scene objects, as shown in Figure 6. So, it is easy to snap to alignment curves, even when they are obscured. This also allows Gargoyle3D to draw new alignment lines without performing hidden line computations when the user activates new alignment values.

Several types of alignment lines can be used simultaneously. In Figure 7, the user has selected spheres of radius 4 inches and three orthogonal alignment planes.

A sphere and three planes are constructed, centered on the anchor. The system automatically computes the circles of intersection where the planes meet the sphere. Using points-preferred gravity, the user can sketch in a precise wireframe octahedron using one or two mouse clicks per line segment (most of the segments can be entered consecutively so the second vertex of one segment becomes the first vertex of the next).

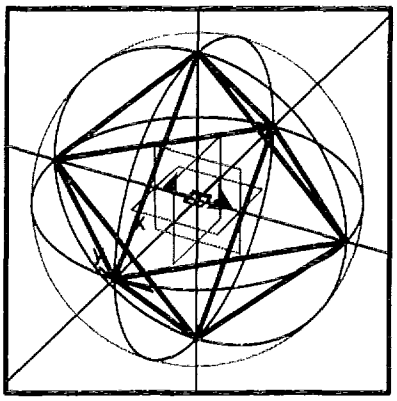

Figure 7. Constructing an octahedron using three alignment objects and their intersection curves.

By activating a mode, called the *automatic rule*, the user can request that the stationary control points of an object, some of whose points are moving, become hot automatically. This makes it easy to align object points with other points of that same object.

While the system is willing to create many alignment objects at once, the user must be careful not to request too many. Any combination of alignment values and hot points that produces more than, say, 100 alignment lines will create enough screen clutter to be more distracting than helpful. Also, alignment lines should be used in preference to alignment planes when possible because the intersection of alignment planes with other planes and alignment spheres quickly creates too many curves.

The designer of a snap-dragging system must carefully consider which intersection curves and points to compute. This decision affects both the implementation and the user. Gargoyle3D currently computes the intersections of alignment objects in all pairs, the intersections of straight edges with all alignment objects and other straight edges, and the intersections of alignment lines with all object surfaces. It would be possible to compute other intersections, including the intersections of all object surfaces with other object surfaces or of object surfaces with alignment planes (and some of these combinations are contemplated for future work); however, additional intersections may contribute to screen clutter, make skitter motion under gravity more jerky, and reduce performance. More powerful workstations and better rendering will affect this trade-off in the future.

The intersection curves of two alignment spheres or of alignment spheres with alignment planes are circles. The gravity algorithm must compute the distance between the each circle and the cursor ray. This is equivalent to finding what radius torus with its tube centered around the circle would be tangent to the cursor line. Rather than solve a fourth-degree equation for each circle, Gargoyle3D approximates the distance as follows: The intersection of the cursor ray with the plane of circle is computed and then the distance of the resulting intersection point to the circle is found. This approximation becomes exact when the cursor ray hits the circle, so the user can compensate for the approximation by pointing more carefully. This approximation is worst when the circle is viewed edge-on. The error can be reduced when the circle becomes nearly edge-on by replacing, in the first step of the algorithm, the plane of the circle by the cylinder that passes through the circle perpendicular to its plane.

Each time the current set of alignment objects changes, it may be necessary to calculate new intersection curves, draw new alignment objects on the screen, and keep track of duplicates. Getting good performance requires making these changes incrementally. The following algorithm has been fully implemented only for two-dimensional snap-dragging; I am confident it will work in three dimensions as well. Call the current set of alignment objects the *align bag*. To keep the align bag up to date, one must handle six cases:

(1) More objects are hot or new hot objects are added to the scene; the new alignment objects generated by these hot points and their intersection points are added to the align bag.

(2) More objects are cold or hot objects are deleted from the scene. Step 1 is inverted.

(3) An alignment value is activated. The Cartesian product of this value with all hot points is computed. The resulting alignment objects and their intersections are added to the align bag.

(4) An alignment value is deactivated. Step 3 is inverted.

(5) An interactive transformation is beginning. Alignment objects triggered by moving hot points are temporarily removed from the trigger bag. If the automatic rule is on, stationary points of partially-moving objects are made temporarily hot, and step 1 is invoked.

(6) An interactive transformation ends. Step 5 is inverted.

Gargoyle3D currently uses an $O(n^2)$ algorithm for detecting duplicate alignment objects. However, it is possible to use hashing to improve this algorithm to nearly $O(n)$. For each alignment line or plane, compute its distance, d, from the origin. For each alignment sphere, compute the distance, d, from its center to the origin of WORLD coordinates. Use one hash table for each alignment value. Use $\lfloor d / \varepsilon \rfloor$ as the hash table key, k, where ε is the farthest apart two alignment objects can be and still be considered duplicates. Each bucket of the hash table is a linked list of alignment objects. Any duplicates of a given alignment object will be

in the buckets whose keys are key-1, key, or key+1. Check all three buckets before adding the object.

5. Transformations

All of the snap-dragging transformations smoothly follow the skitter. Because the skitter continues to snap to scene objects and alignment objects during transformations, objects can be placed precisely. This section describes how translation, scaling, rotation about a point, and rotation about an axis depend on the skitter position and gives an example of each transformation in use.

During a translation operation, the selected objects are translated by the vector from the initial skitter position to the final skitter position. In Figure 8, a selected tetrahedron is translated until its corner snaps onto the corner of an octahedron. The skitter was placed on the corner of the tetrahedron with the "Place Skitter" command. When this command is completed, the cursor moves independently of the skitter, allowing the user to use the mouse for other applications, or invoke menu functions, without disturbing the skitter position. During the "Translate" operation, the skitter tracks the cursor, bringing any selected objects with it (Figure 8(a)). Since the skitter obeys the gravity function, the skitter can be snapped onto scene objects, achieving a precise translation (Figure 8(b)).

As mentioned in section 3, when there are no objects near enough to the cursor ray to snap the skitter to, the skitter moves on a plane, the *default plane*, that is parallel to the screen. The default plane assures that objects move smoothly in the two directions parallel to the screen with a discontinuity in depth whenever the skitter jumps from an object to the plane or from the plane to an object. These small jumps are a standard part of snap-dragging and help the user tell when gravity has drawn the skitter to a new object.

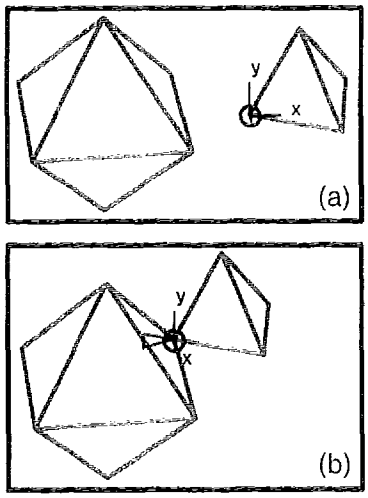

Figure 8. Translation. (a) The "Translate" command begins. (d) Snapping the polyhedra together.

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

The anchor, which was introduced in the last section, is used variously 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 9). The square in the middle of the anchor faces in the z direction, and the barbed line segments lie along the x direction (with an extra arrowhead showing the positive x direction). The anchor is similar in function to the jacks of the skitters and jacks technique, except that there is only one anchor per scene.

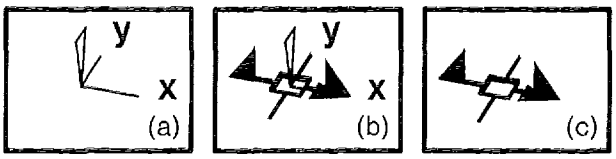

Figure 9. Placing the anchor. (a) The skitter is placed. (b) The anchor is created. (c) The skitter is removed.

During rotation about a point, the selected objects rotate about the anchor point. This rotation occurs through the angle between the line determined by the anchor and the original skitter position, and the line determined by the anchor and the final skitter position. 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 final skitter position determines the axis of rotation. This rotation operation is ideal for rotating two edges to be coincident, as shown in Figure 10.

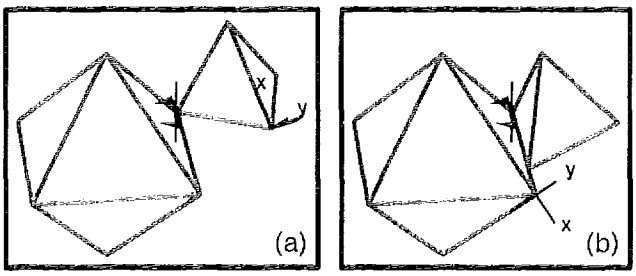

Figure 10. Rotation about a point. (a) The anchor is placed at the shared vertex of the two polyhedra. The skitter is placed on the tetrahedron edge. (b) The skitter snaps to the octahedron.

The scaling operation scales the selected objects by the ratio of the current skitter-to-anchor distance to the original skitter-to-anchor distance. In Figure 11, the tetrahedron is scaled until its edges are congruent to those of the octahedron.

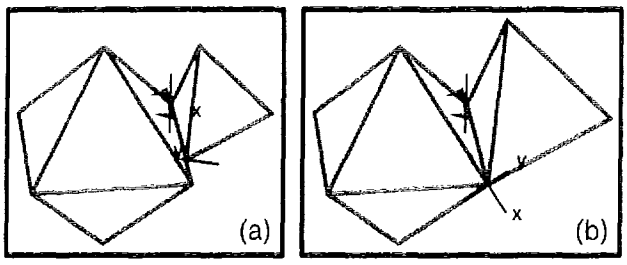

Figure 11. Scaling. (a) The skitter is placed on the nearest vertex of a selected tetrahedron. (b) The tetrahedron is scaled until the skitter snaps to a vertex of the octahedron.

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 12, we rotate the tetrahedron until it shares a face with the octahedron. In Figure 12(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 having the anchor take its orientation from the skitter. Throughout the operation, the left face of the tetrahedron remains coplanar with the skitter, so when the skitter is snapped onto the top vertex of the octahedron the two faces become coplanar (Figure 12(b)).

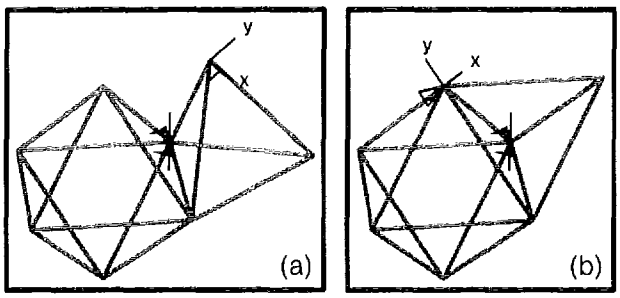

Figure 12. Rotating about an axis. The tetrahedron rotates about the x axis of the anchor (and hence around the edge shared with the octahedron).

6. The Snap-Dragging Commands

Snap-dragging in three dimensions currently employs 19 mouse-based commands, 15 keyboard commands, and a pop-up menu with 10 commands, for a total of 44 commands. This section lists all of the commands and briefly describes those that were not covered above.

6.1 Mouse Commands

Snap-dragging is implemented on a workstation with a keyboard and a three-button mouse. Using the Control and Shift keys on the keyboard in concert with both single and double clicks on the mouse buttons, 24 operations can be invoked. With this arrangement, all of these 19 mouse-oriented commands can be accommodated:

Selecting components (6 commands): Select Vertex, Select Edge, Select Cluster, Select Polygon, Select Within Rectangle, Extend Selection

Deselecting components (5 commands): Deselect Vertex, Deselect Edge, Deselect Cluster, Deselect Polygon, Deselect Within Rectangle

Placing the skitter (1 command): Place Skitter

Adding geometry to the scene (2 commands): Add Edge, Add Block

Transformations (4 commands): Translate, Rotate About Point, Rotate About Axis, Scale

Copying (1 command): Copy & Translate

The selection and deselection commands are not specific to snap-dragging. They allow the user to select vertices, edges, polygons, and complete assemblies (called clusters) in any combination. Once selected, these objects can be transformed, made hot or cold, given new colors and so on.

So that the user can reliably select scene components by pointing, Gargoyle3D provides feedback to disambiguate selections. In Figure 13, a black square highlights a vertex that has been selected. By itself, this feedback would be ambiguous, because it is impossible to tell which of the three blocks the vertex belongs to. To solve this problem, Gargoyle3D highlights, with white squares, the non-selected vertices of the cube whose vertex is selected. By pointing near the vertex but at different angles around it, the user can select the coincident vertices belonging to the other two cubes. The implementation of these selection operations uses the snap-dragging gravity routines with parameters to tune them for selection tasks.

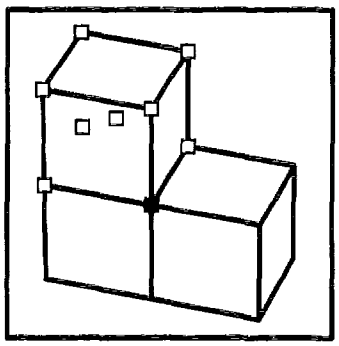

Figure 13. Selection feedback

The Place Skitter command uses gravity to place the skitter within the scene. The scene is not changed. This command is used before a transformation, before an object is added to the scene, and before the anchor is dropped, since all of these operations depend on an initial skitter position.

Because edges and blocks are important shapes, the Add Edge and Add Block operations are provided to add them to the scene with a single button click. Add Edge adds a new line segment from where the skitter was before this operation began to wherever the skitter is placed by the end of the operation. Likewise, Add Block adds a block, aligned with the WORLD coordinate axes, whose two

diagonally opposite corners are these two skitter positions.

The transformations were described in detail in section 5. Copy & Translate works just like Translate, except that it copies the objects before translating them.

6.2 Keyboard Commands

By holding down the Control key and an alphabetic keyboard key, with or without the Shift key, the user can invoke up to 52 commands. Snap-dragging uses 15 of these keyboard combinations for these common operations:

Anchor placement (2 commands): Lift Anchor, Drop Anchor

Selecting components (3 commands): Select All, Cycle Selection Forward (Backward)

Changing the gravity function (3 commands): Cycle Forward (Backward) through the Three Gravity Functions, Toggle Gravity On and Off

Changing hotness (5 commands): Make Hot, Make Cold, Make All Hot, Make All Cold, Toggle the Automatic Rule On and Off

Placing the skitter (2 commands): Cycle Skitter Forward (Backward)

Lifting and dropping the anchor was described in section 5. Select All selects all of the objects in the scene. The Cycle Selection commands select, in turn, all of the objects that were under the cursor during the most recent selection operation. This allows obscured objects to be selected. Cycling though the gravity functions allows the user to choose from points-preferred, lines-preferred, and faces-preferred gravity functions. Gravity can be turned on and off. With gravity off, the skitter always moves on the default plane. Make Hot and Make Cold make the selected objects hot or cold. Make All Hot and Make All Cold make all scene objects hot or cold. The Cycle Skitter commands allow the skitter to be placed on obscured faces by placing it, in turn, on all surfaces that were behind the cursor the last time the Place Skitter operation was performed.

6.3 Menu Commands

To specify a transformation numerically the user can invoke one of these 10 operations, provided in a pop-up menu:

RotateX, RotateY, RotateZ, Scale, Translate and their five inverses.

RotateX(Y,Z) rotates the selected objects around the x(y,z) axis of the anchor by the specified number of degrees. Scale scales them by the specified factor. Translate translates them by a vector, described as three decimal numbers; this vector is interpreted relative to the axes of the anchor. The user types or selects the numerical arguments to these commands.

7. Performance

Two features of snap-dragging require significant computation: computing the gravity function multiple times per second, and computing the intersection curves of alignment objects and scene objects when alignment values are activated. This section describes some performance tests that were performed on a Dorado 16-bit workstation running the Cedar programming environment. This configuration is roughly comparable in computational speed to a SUN 3/160, using Dhrystone as a benchmark.

To test the performance of gravity in Gargoyle3D, I ran three tests. The first test performs points-preferred gravity on a scene consisting of an array of cubes. As most systems have some capability for pointing to vertices, edges, and polygonal faces, this should be a good figure for comparison. I tried different sized arrays of cubes. For each scene, I moved the cursor so that it remained in front of the cubes and took the average over about 20 cursor positions. Here were the results for five different array sizes (all times are in milliseconds):

2x2 (48 edges, 32 vertices):	91 ms
2x4 (96 edges, 64 vertices):	159 ms
4x4 (192 edges, 128 vertices):	199 ms
4x8 (384 edges, 256 vertices):	211 ms
8x8 (768 edges, 512 vertices):	259 ms

The slow growth in time for the larger arrays reflects the fact that most of the cubes are culled by quick rejection tests. Profiling reveals that about 10% of this time is ray-tracing, 48% is finding the distance to edges and computing an orientation for the skitter in case a given edge is chosen, and 41% is for finding the nearest vertices and computing an orientation for the skitter in case a given vertex is chosen. Better factoring of this code would probably improve these results by at least a factor of 2.

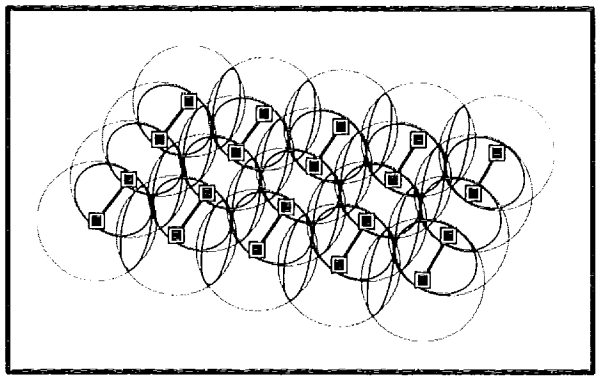

Figure 14. A gravity test case with 10 edges, 20 spheres, and 31 intersection circles.

The second test case, shown in Figure 14, is a scene with 20 hot vertices, 20 spheres, and 31 intersection circles. It is rare that a user would ask for this many alignment spheres at the same time. Even so, gravity can be computed in 45 milliseconds on the average for this scene. When the

spheres are made smaller so that they are precisely tangent, this time drops to 42 milliseconds. If the spheres are made smaller still so that the do not touch, the time drops to 28 milliseconds.

A final case, shown in Figure 15, tests gravity performance in the presence of many alignment lines. The three axes of a coordinate system were drawn with 10 hot vertices evenly spaced along each axis. Three types of alignment lines were activated in the x, y, and z directions. This creates 90 total lines (or 57 lines once duplicates are removed). For this scene, the gravity computation takes 43 milliseconds on average.

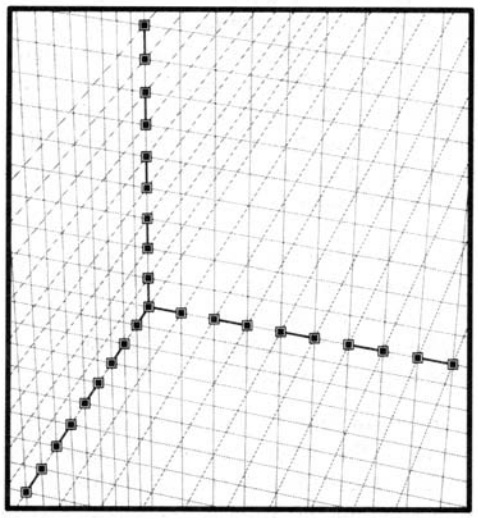

Figure 15. A scene with 30 hot vertices and three active alignment values.

None of these gravity times is prohibitive, and times can be expected to improve with further tuning of the code and faster hardware. Furthermore, the gravity algorithms are linear in the number of scene curves. Rendering algorithms are at best linear in the number of visible curves. Thus, we can expect gravity times to grow no faster than the complexity of scenes that we are able to render. Keep in mind that these times only includes the gravity computation. The total time for an operation would also include the time to update the screen; that time, however, is much more dependent on the available rendering hardware and software.

Three tests were also run of the time taken to compute new alignment objects and their intersections when the user activates a new alignment value. The first test reuses the scene of Figure 14. When the user activates the sphere radius value, the system computes the 20 spheres and 31 intersection circles and adds them to the align bag. This takes 80 milliseconds. This time is reduced to 62 milliseconds if the spheres are tangent, and 50 milliseconds if the spheres do not touch. Profiling reveals that, in the 50 millisecond case, 50% of the time is spent walking the scene data structure enumerating hot points. 24% of the time is spent computing intersections and 18% is spent checking for duplicate spheres.

The second test reuses the example of Figure 15. The three slope line values were turned on one at a time. Each value causes 30 alignment lines to be triggered, 11 of which are duplicates. Adding the first set took 55 milliseconds, the second set took 63 milliseconds, and the third set took 64 milliseconds. Profiling reveals that, for the first set, 46% of the time is spent removing duplicates, 27% is spent walking data structures to enumerate hot points, and 18% is spent allocating storage for the resulting alignment lines. The hashing algorithm described in section 4 should reduce the time for removing duplicates to almost zero.

The final test uses the grid of Figure 15, but activates three perpendicular sets of alignment planes instead of alignment lines. The resulting scene has 30 distinct planes and 300 intersection lines. As this configuration nearly paints the entire screen black, most users would not create this many intersecting planes. Nonetheless, adding the first set of planes takes only 31 milliseconds, adding the second set takes 149 milliseconds, and adding the third set takes 245 milliseconds. Profiling reveals that about 78% of the 245 millisecond time is spent computing intersections, and about 9% is spent removing duplicate planes.

All of the times for updating the align bag are low. Fast updates are not particularly important for the times when the user activates a new alignment value because such activations are relatively rare. However, the user will frequently move or delete hot objects. At these times, updating the align bag is only a sub-part of the computation and good response depends on rapid updating of the align bag.

8. Conclusions and Future Work

Snap-dragging in three dimensions combines a multi-purpose gravity function, alignment objects that are generated from scene hot points by a Cartesian product rule, and smooth-motion affine transformations that are controlled by a 3-space cursor. Snap-dragging combines these elements to produce a new factoring of the user interface for precise interactive scene composition. It can be controlled by a mouse and keyboard. It works with a single projection and produces distinguishable feedback on a bi-level display.

The current implementation of snap-dragging employs 19 mouse commands, 15 keyboard commands, five extensible menus (one each for the sphere, line, and plane alignment objects plus the slope and azimuth menus), and a single pop-up menu for numerically-specified transformations. Together this user interface provides capabilities for selecting and de-selecting objects, placing a cursor and an anchor, translating, rotating, and scaling objects, copying objects, creating and deleting alignment objects, and changing a gravity function.

Gravity, alignment objects, and interactive transformations work well together. The general-purpose gravity function

allows the user to indicate by pointing whether operations begin and end on vertices, control points, alignment objects, edges, or surfaces. The alignment objects allow points and objects to be placed at precise directions and distances from each other. Because multiple alignment objects are computed at once, many points and objects can be placed precisely with a single setting of the alignment object menus. The interactive transformations work together with gravity and alignment objects to produce precision and smooth motion at the same time.

Snap-dragging is a useful alternative to both constraint-based systems and to other direct manipulation system. In cases where the desired result is a single static scene, snap-dragging provides a fast method of building precise scenes that does not require creating a constraint network. Snap-dragging is an improvement over skitters and jacks in both simplicity and power. Because snap-dragging requires only a single anchor instead of many jacks, there is no need for commands to create, move, select, or delete jacks and the number of steps needed to transform scene objects is reduced from four to three. Furthermore, alignment objects and their intersections provide a significantly richer set of end conditions for transformations.

Snap-dragging is computationally intensive. It requires real-time screen updates, gravity computations, and intersection computations. However, as a result of these computations, the user can perform precise interactive scene composition using a small number of commands and a small number of keystrokes per session.

Future work plans include completing the implementation described above, integrating the result into a real design tool, and adding symmetry operations. To improve performance, I am working on a complete implementation of the algorithms to compute the gravity function and update the align bag. Gargoyle3D will also compute more pairwise intersections in the future than it does currently.

Gargoyle3D is currently just a prototype. To make it interesting for users, it will be necessary to implement a complete set of surface primitives, to provide a better renderer both for quick refresh operations and for producing final artwork, and to provide a convenient way to move the viewpoint around the scene.

Many objects that are designed in practice have a great deal of symmetry. I believe that a snap-dragging system that automatically computes symmetry planes, points, and axes and makes them gravity-active will be a particularly powerful construction tool for many applications.

Acknowledgments

I would like to thank a number of people and organizations who have supported and continue to support this work. Thanks to Andrew Glassner and Polle Zellweger for excellent comments on an early draft of this paper. Thanks to all of my reviewers for their detailed and very helpful suggestions. Thanks to Carlo Séquin, my thesis advisor at UC Berkeley, for providing ample resources, encouragement and good advice during my thesis work on snap-dragging. Thanks to Larry Rowe and Alice Agogino, my other thesis committee members, for listening to these ideas when they were just beginning to gel. Special thanks to Maureen Stone for adopting snap-dragging early on, for co-authoring my first SIGGRAPH paper, and for lots of encouragement. Thanks to Michael Plass for thinking of the name "snap-dragging" one day over lunch.

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 throughout.

References

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

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

[Bier86a] Eric A. Bier and Maureen C. Stone. Snap-dragging. SIGGRAPH'86 proceedings, *Computer Graphics*, Vol. 20, No. 4, 1986, pp. 233-240.

[Bier86b] 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.

[Bier88] Eric A. Bier. *Snap-Dragging: Interactive Geometric Design in Two and Three Dimensions.* Report No. UCB/CSD 88/416, April 28, 1988, Computer Science Division, Department of EECS, UC Berkeley, CA 94720. Also available as Xerox PARC report EDL-89-2.

[Borning79] Alan Borning. *Thinglab -- A Constraint-Oriented Simulation Laboratory.* Report SSL-79-3, Xerox PARC, Palo Alto, CA 94304, July 1979. 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*, pp. 27-41, March 1982.

[Chyz85] George W. 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, 1982.

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

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

[Johnson63] 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.

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

[Light82] R. A. Light. Symbolic dimensioning in computer-aided design. Master's thesis, MIT Mechanical Engineering, February 1980.

[Lin81] 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.

[Nielson86] 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.

[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, vol. 15, no. 3, pp. 133-142, August 1981.

[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.

[Robertson89] George G. Robertson, Stuart K. Card, Jock D. Mackinlay. The cognitive coprocessor architecture for interactive user interfaces. Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, November 1989.

[Rossignac86] Jaroslaw R. Rossignac. Constraints in constructive solid geometry. In *Proceedings of the 1986 Workshop on Interactive 3D Graphics* (Chapel Hill, NC, October 23-24, 1986), ACM, New York, 1987, pp. 93-110.

[Serrano84] 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.

[Sutherland63] Ivan E. Sutherland, "Sketchpad, A Man-Machine Graphical Communication System," in Tutorial and selected readings in Interactive Computer Graphics, ed. Herbert Freeman, pp. 2-19, IEEE Computer Society, Silver Spring, MD, 1984, 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.

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

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