Designing Solid Objects Using Interactive Sketch Interpretation

David Pugh
School of Computer Science
Carnegie Mellon University

ABSTRACT

Before the introduction of Computer Aided Design and solid modeling systems, designers had developed a set of techniques for designing solid objects by sketching their ideas on pencil and paper and refining them into workable designs. Unfortunately, these techniques are different from those for designing objects using a solid modeler. Not only does this waste a vast reserve of talent and experience (people typically start drawing from the moment they can hold a crayon), but it also has a more fundamental problem: designers can use their intuition more effectively when sketching than they can when using a solid modeler.

Viking is a solid modeling system whose user-interface is based on interactive sketch interpretation. Interactive sketch interpretation lets the designer create a line-drawing of a desired object while *Viking* generates a three-dimensional object description. This description is consistent with both the designer's line-drawing, and a set of geometric constraints either derived from the line-drawing or placed by the designer. *Viking*'s object descriptions are fully compatible with the object descriptions used by traditional solid modelers. As a result, interactive sketch interpretation can be used with traditional solid modeling techniques, combining the advantages of both sketching and solid modeling.

David Pugh
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
dep@cs.cmu.edu

1 INTRODUCTION

Sketching has long been an important element of the design process. For hundreds of years, people have designed by making quick, abstract drawings or "sketches." Sketching was used both to specify embryonic concepts and to refine these concepts into workable designs. Thirty or so years ago, the advent of Computer Aided Design (CAD) and solid modeling systems began to revolutionize some aspects of the design process. These programs let designers create a model of a three-dimensional object on the computer. This model can then be analyzed in ways that would be difficult or impossible without the computer. For example, CAD systems and associated programs can display realistic images, do stress analyses, and generate milling machine programs from the computer's model of the object.

Unfortunately, the CAD revolution did not extend to at least two critical aspects of the design process: exploring new ideas and refining these ideas into workable designs. With current CAD systems, the model typically changes in large, discontinuous steps. The designer is often forced to fully specify a change before he or she has a chance to see how it interacts with the rest of the model. This makes "feedback driven" design, in which the designer uses feedback from one change to guide the next change, difficult on a solid modeler: the magnitude of each change is too large to let the designer use his or her intuition effectively. As a result, designers will often use pencil and paper to "work out" a change before making the change on the computer.

The techniques used to design objects on pencil and paper are different from those used to design objects on a solid modeler [13]. Sketching, in this context, is a visual and intuitive process in which a drawing is refined over time by making small, incremental changes. At each point in the process, the designer uses feedback from one change – the appearance of the modified sketch – to guide the next change. The continual feedback lets the designer use his or her intuition effectively.

This paper presents a solid modeling system, *Viking*, that lets the user design three-dimensional objects using techniques normally used to create and refine two-dimensional

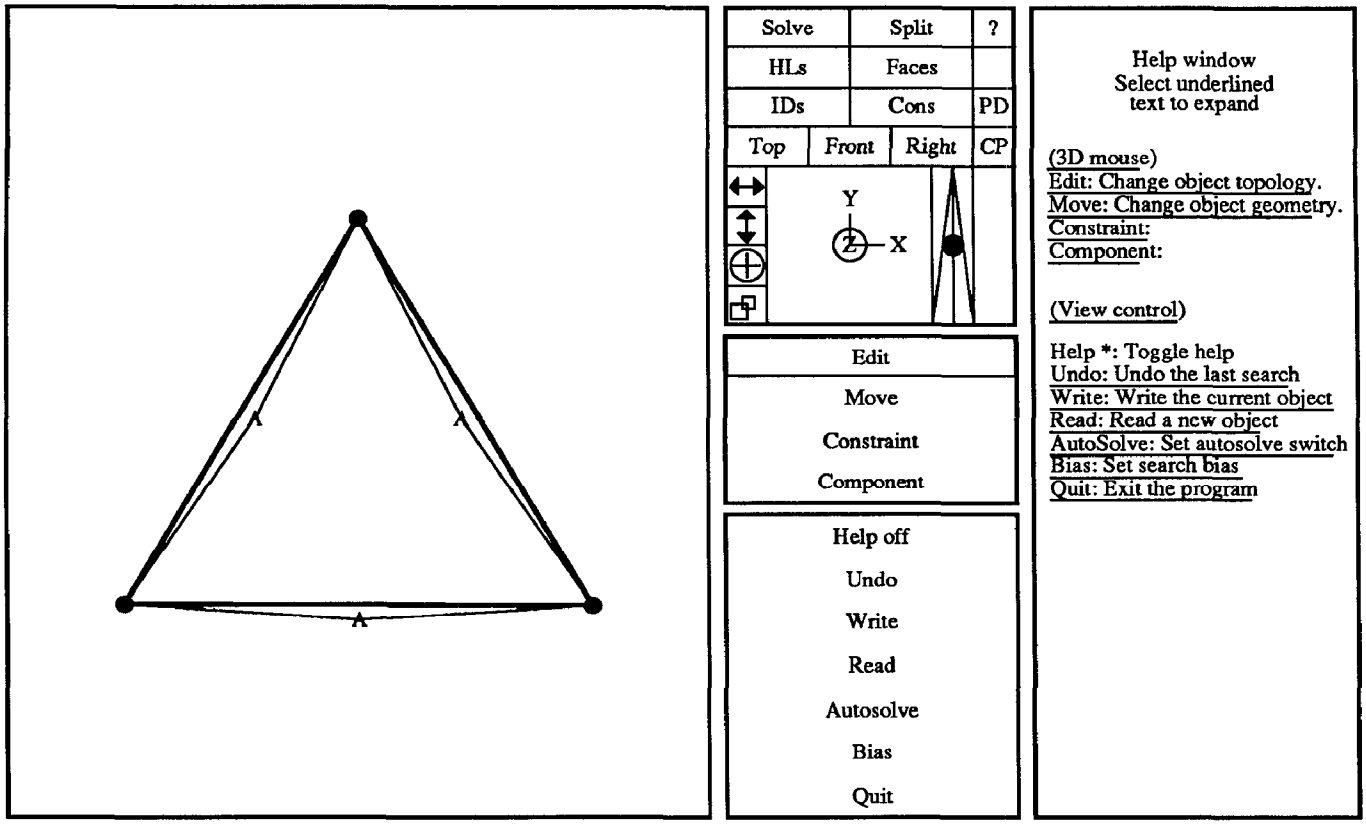

Figure 1: *Viking*'s display.

sketches. *Viking* uses interactive sketch interpretation to create a "what you draw is what you get" user-interface. Users can create a line-drawing of a desired object and use sketch interpretation to generate a three-dimensional object that is consistent with the line-drawing. Users can also place geometric constraints on the object. These constraints, together with a set of constraints derived from the line-drawing, are used to define a vertex geometry in subsequent interpretations. Geometric constraints let the user create precisely dimensioned objects. The resulting user-interface combines the power of traditional solid modeling systems with the continuous feedback of sketching.

2 THE *VIKING* SOLID MODELER

Viking extends the direct manipulation metaphor to three-dimensional object design by letting the user modify an object by changing its line-drawing. For most changes, deducing an appropriate change in the object description is trivial. For example, if the user erases a line, delete the corresponding edge. With other changes, such as making a line-segment visible, there is no obvious corresponding change in the object description. Sketch interpretation is used in these cases to generate a new object description that is consistent with the modified line-drawing.

Sketch interpretation divides the task of interpreting a line-drawing into two parts: finding a surface-topology and solving for a vertex geometry. The first part is done by generating surface-topologies that are consistent with the line-drawing until one that is acceptable to the user is found. The second is done by using a geometric constraint solver to find a vertex geometry that satisfies a system of constraints either derived from the line-drawing and the proposed surface-topology, or placed by the user. The surface-topology and vertex geometry combine to form a three-dimensional object description that is consistent with both the line-drawing and the constraints.

2.1 *VIKING*'S USER-INTERFACE

Figure 1 shows *Viking*'s display after creating an equilateral triangle. The left window shows a line-drawing of underlying object description and the upper center window shows the view transform used to generate the line-drawing. Both windows let the user directly modify their contents. The user can, for example, move a vertex by dragging it to a new location with the mouse. The user can also dynamically change the view transform by dragging the mouse across the orientation triad, rotating the view about an axis perpendicular to the mouse's motion [9].

The line-drawing displays more than just an object's shape. Thick, thin and double lines respectively correspond to edges adjacent to zero, one and two faces in the object description. Circles correspond to vertices that can be moved by the constraint solver when solving for a vertex geometry. Triangles correspond to vertices whose positions are considered fixed constants by the constraint solver. Constraints are drawn in a variety of ways. Distance constraints, for example, are shown by thin, bent lines. In Figure 1, the "A" symbol at the bend indicates that all three sides of the triangle

have the same length.

2.1.1 VIKING'S COMMAND MODES

The four items shown in the center window of Figure 1 (*Edit*, *Move*, *Constraint* and *Component*) correspond to the four most commonly used modes in *Viking*. These modes determine how mouse actions in the line-drawing's window are interpreted. If the user enters either *Constraint* or *Component* modes, the center window is overwritten with a specialized menu.

Edit mode is used for changing the appearance of the line-drawing displayed in the image window. While in it, the user can draw new edges, erase old ones and change the visibility of line-segments. For the first two actions, both the line-drawing and the underlying object description change. For the last action, only the line-drawing changes: the underlying object description is not always modified: the Autosolve switch, located on in the bottom center window, determines whether *Viking* will automatically generate a new interpretation after the user changes the visibility of a line-segment, or wait until the user explicitly requests a new interpretation.

Move mode is used for placing tacks, and moving vertices and edges. Tacks are simple constraints that either lock a vertex into a fixed position or force an edge to pass through a point in space. If the Autosolve switch is on, *Viking* will use the constraint solver to maintain the constraints as the user drags a vertex or edge around with the mouse. Otherwise, the vertex or edge will follow the mouse without maintaining the constraints.

Constraint mode is used for placing or editing geometric constraints on the object. The constraint menu lets the user select a constraint template and then define constraints by picking vertices or edges to "fill in" the blanks. The user can also modify or delete previously defined constraints. Whenever the user adds a constraint, *Viking* will attempt to find a solution to the new system if the Autosolve switch is turned on.

Component mode is used for manipulating groups of vertices, edges and faces. Every component has a coordinate transform that defines the effective position of its vertices. The coordinate transform is generated from eleven variables that control a component's size (using both an axis-independent variable and three axis-dependent variables), position, and orientation (using quaternions [12]). The user can lock or free these variables independently and the constraint solver can manipulate the free variables when solving for a vertex geometry.

2.1.2 SKETCHING IN THREE-DIMENSIONS

Sketching is traditionally done in only two dimensions. With *Viking*, however, sketches are three-dimensional entities. This both aids and hinders the user. A three-dimensional "sketch" can help the user visualize the object it represents. But it also means the user must specify the location of each vertex in three-dimensions.

A simple mechanism for specifying a vertices' approximate location is needed. If the user can place every vertex near its correct position, then the user can rotate the object and the line-drawing will behave intuitively. This lets the user continue the design process until he or she knows enough to start using constraints to specify the vertices' position precisely. Also, since the vertices start close to a geometry that satisfies the constraints, the constraint solver will need less time to find a solution.

Geometric constraints are not, by themselves, a good mechanism for specifying approximate vertex positions. In part, this is because the constraint solver works best when all vertices are near a solution. Relying on the constraint solver to move a vertex a significant distance is, at best, time consuming and often results in unexpected and unwanted solutions (assuming any solution is found). A more fundamental problem with using constraints for rough positioning, however, is their precision. Often, users do not know the precise location of a vertex until late in the design process. Using constraints to position a vertex before the user knows its precise location is time consuming since the constraints will have to be changed later, when the precise dimensions are known. It can also be intimidating: people do not like answering questions until after they know the answers.

The user can position a vertex in three-dimensions by showing where it "should be" in two different views. Unfortunately, this technique forces the user to work in two different views, which is difficult. For example, it is not always obvious which vertex in one view corresponds to which vertex in the other.

When no other information is available, *Viking* uses a simple rule when drawing edges: both end-points have the same z-coordinate in the display's coordinate space. For many cases, such as drawing a short edge from an existing vertex, this is sufficient. In other cases, neither this method nor the alternatives given above suffice. Because of this, *Viking* provides two additional mechanisms to let the user easily specify the location of a vertex in three-dimensions: preferred directions and cutting planes.

Preferred directions are three-dimensional vectors. When the user draws an edge, *Viking* draws short lines parallel to each preferred direction at the new edge's origin. As the user moves the mouse, the edge's endpoint is projected onto the closest preferred direction.

Preferred directions can be defined in two ways. First, the user can define vectors in object space, such as the x, y and z axes, for preferred directions. Any new edge, no matter where it is drawn, will be able to use these preferred directions. Second, the user can put preferred directions on automatic. In this case, *Viking* automatically defines preferred directions depending on the context in which the user started to draw the new edge. If the user is drawing an edge from an existing vertex, then the preferred directions are defined to be parallel to each of the edges radiating from the vertex. If the user is drawing an edge from an existing edge,

then one preferred direction is defined to be parallel to the edge and, for each adjacent face, a preferred direction is defined to lie in that face's plane and be perpendicular to the edge. If these rules generate one preferred direction, then two preferred directions are added that are perpendicular to the original preferred direction and each other. If two preferred directions were generated, then a third preferred direction perpendicular to the first two is added.

A cutting plane is a plane defined in object space. Cutting planes are a tool for both positioning a vertex in three-dimensions and helping the user visualize the object's three-dimensional structure. The user can position a vertex in three-dimensions by moving it parallel to the cutting plane or parallel to the cutting plane's normal.

The user can manipulate the cutting plane by moving it parallel to its normal, changing the orientation of its normal, and controlling the way in which it is displayed. The user can, among other things, make the cutting plane opaque or translucent, highlight the intersection of the cutting plane with the object, show the orthogonal projection of the object onto the cutting plane, and draw height poles between each vertex and the cutting plane.

3 IMPLEMENTATION

Viking's implementation of interactive sketch interpretation uses two distinct data-structures: one holds the current object description and the other holds the line-drawing displayed to the user. The user can modify the line-drawing and most changes automatically propagate to the current object description, maintaining consistency between the two data-structures. The user can also change the viewpoint, in which case the line-drawing is recreated from the new view transform and the current object description.

Sketch interpretation generates a new object description when the user makes a change that can not be propagated to the object description automatically. *Viking*'s sketch interpretation algorithm splits the task of generating a new object description into two parts: finding a surface-topology that is consistent with the line-drawing and solving for a vertex geometry that satisfies the object's implicit and explicit constraints. Together, the surface-topology and the vertex geometry completely describe a three-dimensional object. The new object description is consistent with both the line-drawing created by the user and any geometric constraints he or she may have specified.

Viking uses arc-labeling [10], an extension of Huffman-Clowes line-labeling [3, 8] to non-trihedral vertices, to generate a surface-topology from a line-drawing and an old object description. The surface-topology defines a set of faces that are consistent with the line-drawing. Since line-drawings can have many different interpretations, *Viking* uses heuristics to seek out the more desirable interpretations first. *Viking* generates surface-topologies in order of increasing cost, where the cost is based on several heuristics, including:

- how similar the surface-topology is to the current object's surface-topology and
- if the user has given a preferred object type, how close the surface-topology is to the user's preferred type.

Surface-topologies are generated until the user either accepts one or aborts the search. In my experience, the desired surface-topology is normally the first surface-topology found.

Once an acceptable surface-topology has been found, a non-linear constraint solver finds a vertex geometry that satisfies a system of geometric constraints. These constraints fall into three categories:

- world: every face is a planar polygon.
- image: visible lines are in front of obscuring faces.
- explicit: constraints explicitly defined by the user.

The first two types of constraints are implicit constraints since they are automatically generated by *Viking*. World and explicit constraints are always part of the system of equations used by the constraint solver. Image constraints are only used when finding a vertex geometry after generating a new surface-topology for the object.

The constraint solver uses an algorithm developed by Bullard and Biegler [2]. This algorithm repeatedly solves a system of linear equations derived from the non-linear equations and their first derivatives until the global error is reduced below a threshold. The vertex positions from the current object are used as the initial solution for the new system of constraints. The solver tends to move the vertices only in small, well controlled steps and, as a result, solutions tend not to differ unnecessarily from the vertex geometry in the current object.

Once an acceptable surface-topology and vertex geometry have been found, *Viking* replaces the current object description with the new interpretation. A new line-drawing is then generated from the new current object description and the current view transform. The user can manipulate the new line-drawing just like the old one, letting the user continue the cycle of modification and interpretation.

4 EXAMPLES

4.1 CREATING A CHAIR

This section describes a session using *Viking* to create an "easy chair." This example is somewhat contrived (for example, chairs are not normally made from homogeneous blocks) but it does convey the flavor of *Viking*'s user-interface. It also demonstrates how modifying the line-drawing can be used as a substitute for constructive solid geometry. It took me less than two minutes to transform the cube in Figure 2a into the chair in Figure 2i.

Preferred directions (see Section 2.1.2) were on automatic throughout this example. As a result, whenever the user started to draw an edge, *Viking* defined a set of context

dependent vectors that could be used to position the edge's endpoint in three-dimensions. For example, preferred directions made it possible to draw the new edge in Figure 2b so that it was parallel to the edge between the upper and lower vertices at the right and back of the cube.

Figure 2a shows the initial object, a cube loaded from a library of standard objects. The first step in turning this cube into a chair is to add a raised back. Figure 2b shows the user drawing a new edge up from the upper-right corner of the cube. The user has finished drawing the edges for the chair's back in Figure 2c and is in the process of hiding the line-segments that would be obscured if the chair's back was solid and opaque.

In Figure 2d, the user deleted one unwanted vertex and is in the process of deleting the other (the user must pick a vertex twice to delete it: the first pick highlights the selected vertex, the second deletes is). These vertices are unwanted because deleting them and redrawing the missing edges ensures that the chair's back is a single, planar surface. If these vertices had not been deleted, *Viking* would have found an interpretation in which the chair's back and sides were each formed by two faces.

Deleting a vertex also deletes its adjacent edges and faces, although *Viking* preserves the hidden status of line-segments whose obscuring face is deleted. For example, in Figure 2d, the line at the bottom-back of the cube is drawn with a single, thin line (indicating that it is adjacent to only one face) since the top, back and right faces of the cube were deleted when the first vertex was deleted. Also, the entire line remains hidden, even though the face obscuring its right segment has been deleted.

Figure 2e shows the user redrawing some of the edges that were deleted when the user deleted the unwanted vertices, in preparation for using sketch interpretation to generate a new object description. Figure 2f shows, from a different viewpoint, the user starting to draw a lowered seat on the first interpretation found for Figure 2e. Since the user had set the search bias to prefer solid objects, *Viking* sought out an interpretation corresponding to a solid object. As a result, the interpretation contains faces that were not needed to generate an object description consistent with Figure 2e since they would have been hidden by the rest of the chair.

The user has finished drawing a lowered seat for the chair in Figure 2g and is in the process of removing some unwanted and unnecessary edges. In Figure 2h, the user is exposing the line-segments that would be visible if the chair's seat was lower than its arm rests. Figure 2i shows, from a different viewpoint, the first interpretation found for Figure 2h.

Even though the chair looks correct in Figure 2i, the geometry is not correct. For example, some edges that should be parallel to each other are skewed about 10°. These problems can be fixed in a minute or two by using geometric constraints. But, since the next example demonstrates the constraint solver, that part of the design process is skipped.

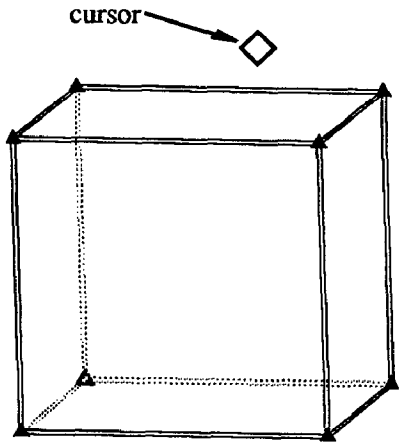

Figure 2a: Initial object: a unit cube.

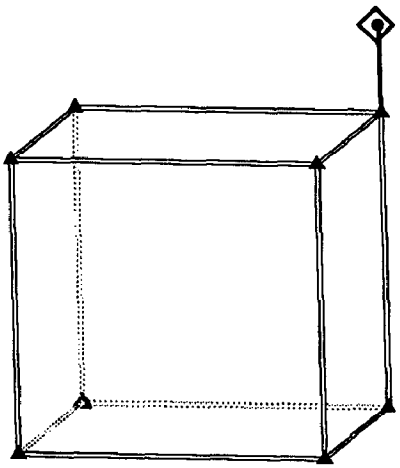

Figure 2b: Drawing the chair's back.

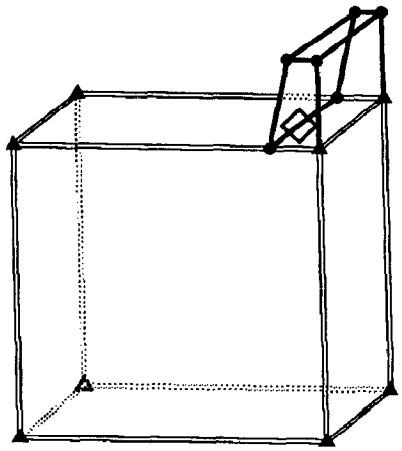

Figure 2c: Hiding obscured line-segments.

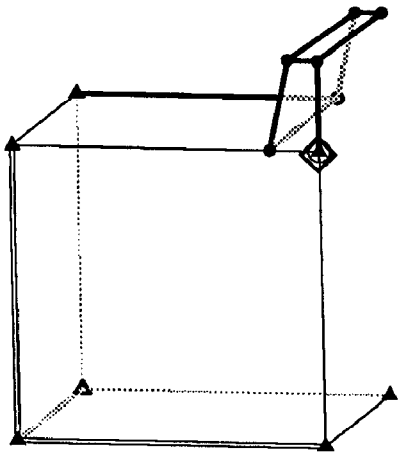

Figure 2d: Remove unwanted vertices and edges.

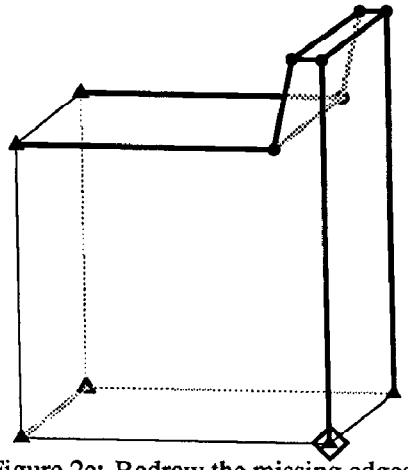

Figure 2e: Redraw the missing edges.

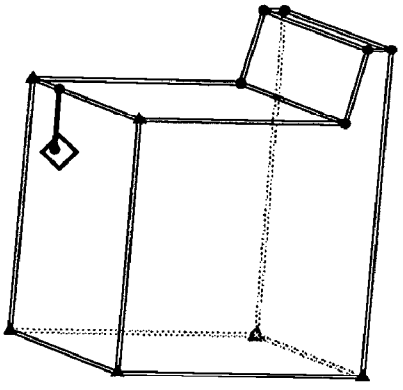

Figure 2f: Drawing the chair's seat.

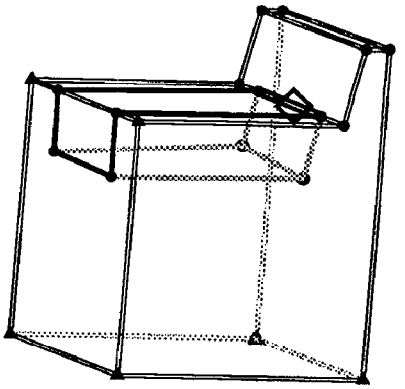

Figure 2g: Remove unwanted edges.

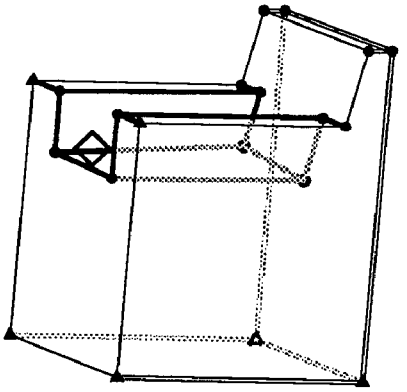

Figure 2h: Exposing visible line-segments.

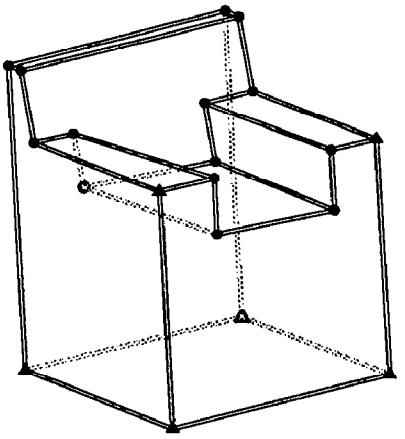

Figure 2i: The "completed" chair.

4.2 AN EXERCISE IN GEOMETRY

Suppose you have the following problem: if you place a solid equilateral tetrahedron face to face with a solid equilateral octahedron, how many faces does the resulting polyhedron have? The polyhedra are positioned and sized so that three of the tetrahedron's vertices coincide with three of the octahedron's vertices. Answering this question, by using *Viking* to create the object shown in Figure 31, takes me less than three minutes.

Figure 3a shows the user starting to draw the two polyhedra. In Figure 3b, the user has changed the view transform by rotating it about the horizontal axis and is in the process of completing the octahedron's wire-frame. Figure 3c shows the user hiding the line-segments at the "back" of the polyhedra. Figure 3d shows the first interpretation found after hiding the rest of the line-segments that should be obscured.

The edges in Figure 3d were drawn without using either preferred directions or a cutting plane to position the vertices in three-dimensions. The user made no attempt to draw the edges so that they all had exactly the same length. Instead, geometric constraints will be used to turn these "rough sketches" into equilateral polyhedra.

Figure 3e shows the effect of adding and solving for equal length constraints on the tetrahedron's edges. Figure 3f shows the effect of placing a similar set of constraints on the octahedron. The bent lines and "A" symbols indicate that all of the tetrahedron's edges have the same length. The bent lines and "B" symbols do the same for the octahedron's edges. In both Figures 3e and 3f, the vertices have moved to accommodate the constraints. Figure 3g, in which display of the constraints has been turned off, shows the two polyhedra from a different direction.

In Figure 3h, the user has added, but not yet solved for, constraints forcing three of the tetrahedron's vertices to be coincident with three of the octahedron's vertices. The bent line and "0" symbol indicates that the distance between the vertices should be zero. Figure 3i shows the solution found by the constraint solver to the system described in Figure 3h. Figures 3h and 3i have, despite appearances, identical surface-topologies: the constraint solver moved the vertices without changing the underlying structure.

In Figure 3j, the view transform has been changed to give a view "straight-down" one of the edges where the tetrahedron and octahedron are in contact. This view suggests that the vertices to either side of this edge are co-planar, forming a single four-sided face. In Figure 3k, the user has merged the six coincident vertices into three vertices, deleted the unwanted edges, and generated a new, seven-sided, interpretation. Figure 31 shows Figure 3k with all constraints hidden. Since all faces must be planar, *Viking* would not be able to find a vertex geometry for Figure 3k unless the quadrilateral faces were planar polygons. The answer, therefore, to the question posed at the beginning of this section is that a tetrahedron and octahedron form a seven-sided polyhedra.

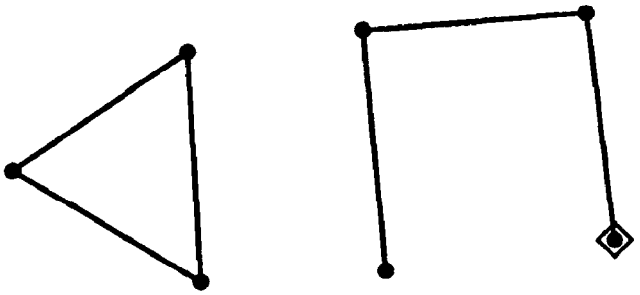

Figure 3a: Drawing the polyhedra.

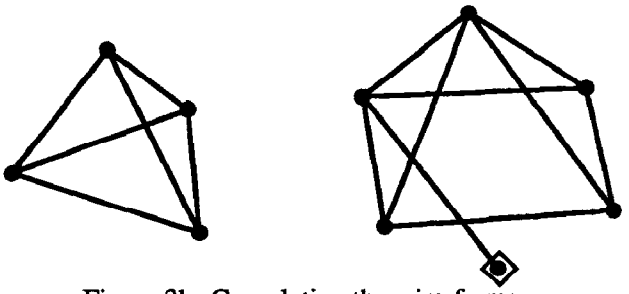

Figure 3b: Completing the wire-frames.

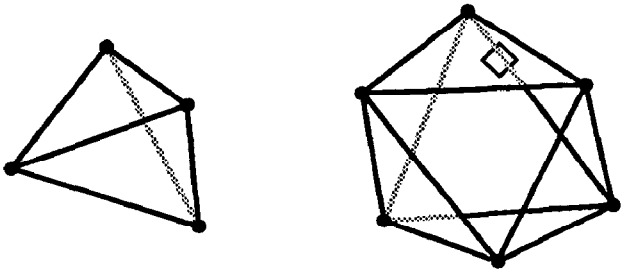

Figure 3c: Hiding obscured line-segments.

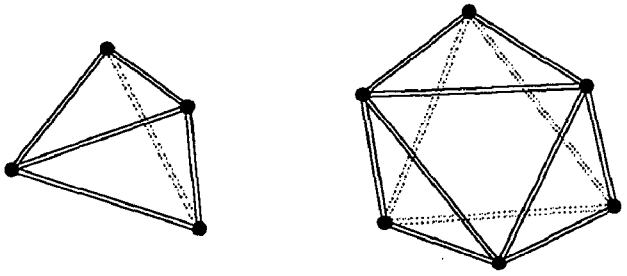

Figure 3d: Generating an interpretation.

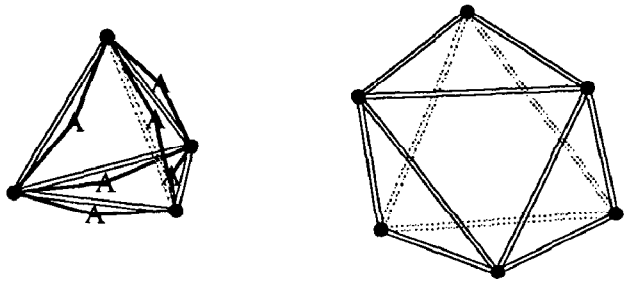

Figure 3e: Making an equilateral tetrahedron.

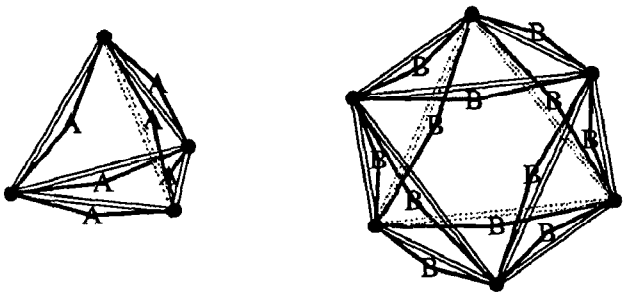

Figure 3f: Making an equilateral octahedron.

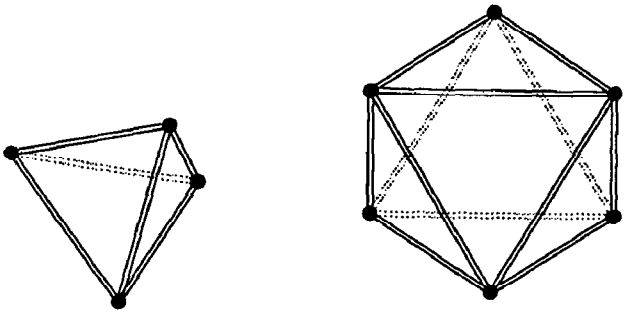

Figure 3g: Viewing from another direction.

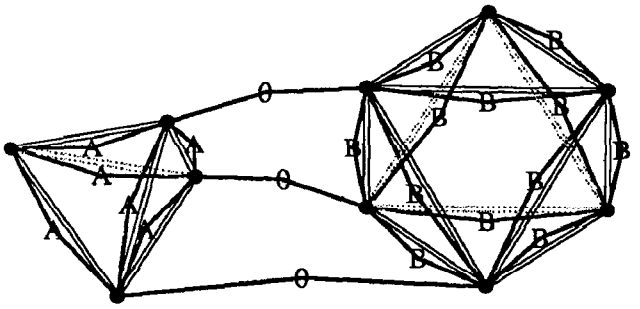

Figure 3h: Before solving the coincidence constraints.

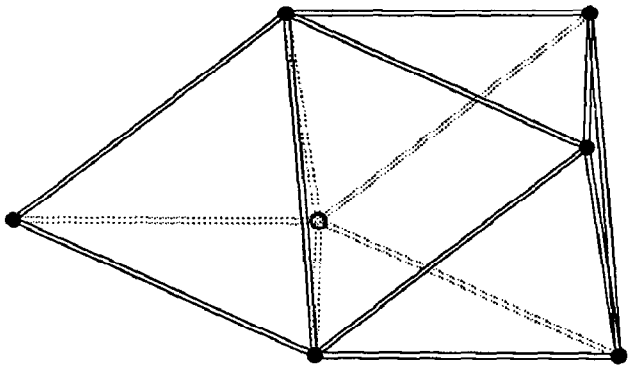

Figure 3i: After solving the coincidence constraints.

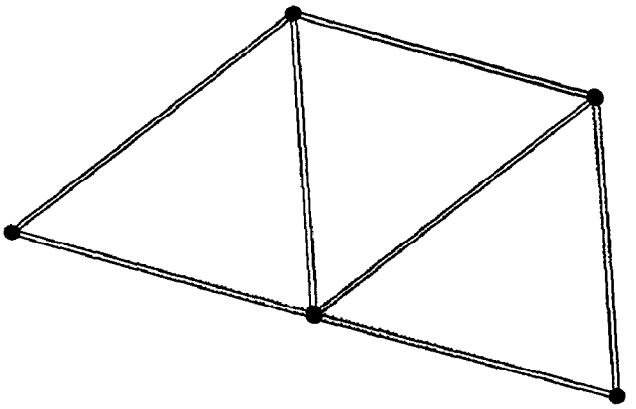

Figure 3j: An "edge-on" view.

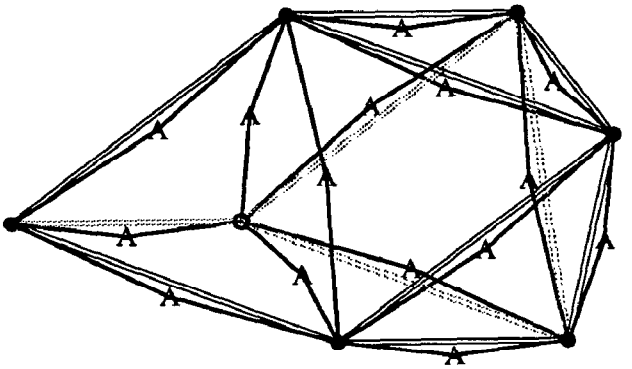

Figure 3k: The resulting seven-sided polyhedra.

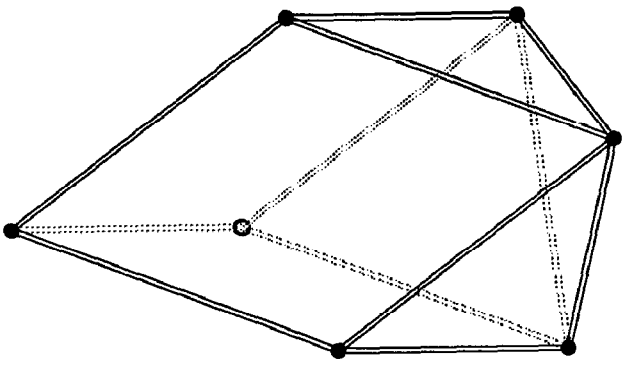

Figure 3l: Figure 3k with the constraints hidden.

5 FUTURE WORK

Viking's user-interface has some significant weaknesses. Some of these are problems should not be difficult to solve. Others do not seem to have easy solutions. These problems are presented in the order that they will be addressed in future research.

CAD modeling interface

Currently, *Viking* provides few of the capabilities found in conventional solid modeling systems. For example, *Viking* can neither calculate the mass of an object nor find the intersection of two objects. Combining conventional solid modeling capabilities and interactive sketch interpretation should not be difficult: *Viking*'s underlying object description is equivalent to the boundary representation description used by some solid modelers.

Explicit constraints specification

Viking's users must explicitly specify every geometric constraint. Other constraint based design systems, such as Snap-Dragging [1] [5], provide mechanisms for defining constraints implicitly. Incorporating similar mechanisms into *Viking* could alleviate one of the more tedious aspects of *Viking*'s user-interface.

Planar faces and straight edges

Viking can, currently, only interpret line-drawings of objects with planar faces. The sketch interpretation algorithm can be easily extended to objects with non-planar faces. Modifying the rest of *Viking* however, is more difficult: planar faces provides one of the better implicit constraints and designing a good user-interface for letting the user specify which faces are non-planar and controlling the shape of a non-planar face is not easy.

Quadhedral vertices

Viking can only analyze line-drawings in which every vertex is adjacent to four or fewer edges. This is because *Viking*'s sketch interpretation algorithm must match every intersection in the line-drawing to an entry in a fixed intersection library. This library contains all possible intersections of two, three and four lines. The program used to generate *Viking*'s intersection library, however, is already capable of generating entries for intersections of five or more lines. Adding this capability to *Viking* should not be difficult.

Simple polygonal faces

Faces in *Viking* must be simple, planar polygons: no internal holes or repeated edges or vertices. It should be possible to extend the algorithm to allow more complicated faces, although it may not be worth the extra processing time required. The current version of *Viking* lets the user simulate holes and the like by using artifact edges.

Explicit topology specification

Viking's sketch interpretation algorithm uses the presence of hidden lines-segments to automatically reject inconsistent interpretations. The downside of this is that the user must correctly indicate which line-segments are hidden. This can be a tedious and time-consuming process.

Viking lets the user generate a blind interpretation, in which the visibility cues are ignored and, therefore, the user does not have to indicate which line-segments are hidden. Blind interpretations are slower and less discriminating than conventional interpretation, since visibility cues can not be used to reject unwanted topologies. Despite this, it is often easier to generate a blind interpretation and manually reject unwanted topologies than it is to indicate which line-segments are hidden and generate a standard interpretation.

5.1 OPEN PROBLEMS

The following section describes problems that do not seem to have easy solutions.

General view

Viking's sketch interpretation algorithm can only interpret line-drawings that correspond to a general view of an object. A general view is one in which a small change in the view direction makes correspondingly small change in the line-drawing [11]. So, for example, a general view could not contain any faces that are "edge-on" to the viewer (such as Figure 3j).

This is a problem, since engineering drawings often do not correspond to general views. However, it is not clear how significant this problem is. Engineering drawings often used specialized viewpoints because, historically, specialized views were easier to draw or because they illustrated a particular point. Specialized views are not, for the most part, easier to interpret than general views and both types of views are easy to generate using the computer.

One possibility for generating interpretations of specialized views is to use graph based algorithms [4] [7]. These algorithms do not depend on the viewpoint, generating a surface-topology by finding a planar embedding of an object's vertex-edge graph. Unfortunately, these algorithms probably could not be modified to use *Viking*'s search heuristics.

Sketch interpretation performance

Viking's sketch interpretation algorithm is not as fast as one might wish, taking almost three minutes to generate an interpretation of a line-drawing containing 100 points. The time required to generate an interpretation seems to be roughly proportional to the square of the number of points in the line-drawing. Although

faster workstations and more efficient algorithms may alleviate this problem, it is not realistic to expect that *Viking*'s sketch interpretation algorithm could be used on large objects (which might three or four orders of magnitude more complex than the objects created in Sections 4.1 or 4.2). It should, however, be possible to automatically partition a large object and use sketch interpretation on only the relevant parts.

Constraint satisfaction performance

Viking's constraint solver is used in two basic modes: when one or more constraints have been added and *Viking* must solve for a solution and when the user is moving a vertex by dragging it with the mouse and wishes to maintain the pre-existing constraints. The response time when dragging is far slower than desired, often taking several seconds to find a solution that satisfies all the constraints. It might be possible to use differential constraints [6] to improve response times when dragging.

6 CONCLUSIONS

Viking is a solid modeling system that uses interactive sketch interpretation to combine the simplicity of pencil and paper sketches with the power of a solid modeling system. *Viking* lets designers draw the object they wish to create and then modify it by changing the line-drawing to make it "look right." Each action is obvious from context, leaving the designer free to concentrate on the design itself and not how to convey it to the solid modeler.

This ease of use comes without sacrificing any of the capabilities intrinsic to solid modeling systems. As with other solid modeling systems, *Viking* lets the designer manipulate the underlying object description as if it were a solid object. This provides the designer with a powerful tool for visualizing an object's structure. For example, the designer can wiggle the object by dynamically changing the view transform or drag a translucent cutting plane through the object to see where vertices lie with respect to one another in three-dimensions. And, although *Viking*'s user-interface is based primarily on sketching, the designer can create precisely dimensioned models by using geometric constraints. This combination of sketching and solid modeling techniques creates an effective user-interface for developing ideas into practical designs.

REFERENCES

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

[2] L.G. Bullard and L.T. Biegler. Lp strategies for constraint simulation. In *AIChE '89 Conference Proceedings*, November 1989.

[3] M. B. Clowes. On seeing things. *Artificial Intelligence*, 2:79–116, 1971.

[4] S. Mark Courter and John A. Brewer III. Automated conversion of curvilinear wire-frame models to surface boundary models; a topological approach. In *SIGGRAPH '86 Conference Proceedings*, pages 171–178, 1986.

[5] Michael Gleicher and Andrew Witkin. Creating and manipulating constrained models. Technical Report CMU-CS-91-125, Carnegie Mellon University, 1991.

[6] Michael Gleicher and Andrew Witkin. Differential manipulation. *Graphics Interface*, pages 61–67, June 1991.

[7] Patrick M. Hanrahan. Creating volume models from edge-vertex graphs. In *SIGGRAPH '82 Conference Proceedings*, pages 77–84, 1982.

[8] D. A. Huffman. Impossible objects as nonsense sentences. *Machine Intelligence*, 6:295–323, 1971.

[9] S. Joy Mountford Michael Chen and Abigail Sellen. A study in interactive 3-d rotation using 2-d control devices. In *SIGGRAPH '88 Conference Proceedings*, pages 121–129, 1988.

[10] David Pugh. Interactive sketch interpretation using arc-labeling and geometric constraint satisfaction. Technical Report CMU-CS-91-181, Carnegie Mellon University, 1991.

[11] P. V. Sanker. A vertex coding scheme for interpreting ambiguous trihedral solids. *Computer Graphics and Image Processing*, 6:61–89, 1977.

[12] Ken Shoemake. Animating rotation with quaternion curves. In *SIGGRAPH '85 Conference Proceedings*, pages 177–186, 1985.

[13] Rob Woodbury. Searching for designs: Paradigm and practice. *Building and Environment*, 26(1):61–73, 1991.