Interactive Sketch Interpretation Using Arc-labeling and Geometric Constraint Satisfaction.

David Pugh

September, 1991
CMU-CS-91-181 2

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes the algorithms used by Viking, a solid modeling system whose user-interface is based on interactive sketch interpretation. Pencil and paper sketches are widely used early in the design process. Viking's user-interface lets the user create these sketches on the computer and automatically generate the corresponding three-dimensional interpretation. This user-interface combines the simplicity of pencil and paper sketches with the power of a solid modeling system.

Sketch interpretation combines arc-labeling and constraint satisfaction. Arc-labeling generates a surface topology for the interpretation. This algorithm extends the line-labeling methodology to non-trihedral objects and uses heuristics to generate interpretations in order of decreasing desirability to the user. Constraint satisfaction solves for the vertex positions in the interpretation. Constraints are either generated from implicit relations in the line-drawing or explicitly created by the user.

Keywords: solid modeling, sketch interpretation, line-labeling, arc-labeling, constraint satisfaction, geometric constraints

Interactive Sketch Interpretation Using Arc-labeling and Geometric Constraint Satisfaction.

David Pugh

Carnegie-Mellon University
Pittsburgh, PA 15213, USA.

September 29, 1991

Abstract

This paper describes the algorithms used by Viking, a solid modeling system whose user-interface is based on interactive sketch interpretation. Pencil and paper sketches are widely used early in the design process. Viking's user-interface lets the user create these sketches on the computer and automatically generate the corresponding three-dimensional interpretation. This user-interface combines the simplicity of pencil and paper sketches with the power of a solid modeling system.

Sketch interpretation combines arc-labeling and constraint satisfaction. Arc-labeling generates a surface topology for the interpretation. This algorithm extends the line-labeling methodology to non-trihedral objects and uses heuristics to generate interpretations in order of decreasing desirability to the user. Constraint satisfaction solves for the vertex positions in the interpretation. Constraints are either generated from implicit relations in the line-drawing or explicitly created by the user.

1 Introduction

Sketches have long played an important role in design. Even now, despite the wide availability of solid modeling systems, designers will often do the preliminary design using pencil and paper instead of using the computer. This paper describes a solid modeling system, called Viking, which uses sketches created by the user to determine an object's shape. Topologic and geometric constraints are derived from the sketch and a three-dimensional object description consistent with those constraints is generated. The resulting object, despite the ambiguity inherent in a sketch, is normally close enough to the intended object that the user's momentum is maintained and the design process can continue without pause.

Viking uses arc-labeling and constraint satisfaction allow the user to create complex, precisely dimensioned, models. The task of generating a new object description has two parts: arc-labeling is used to find the surface topology, and constraint satisfaction is used to solve for the vertex positions. Arc-labeling extends Huffman-Clowes line-labeling [5] [9] to non-trihedral objects. Constraint satisfaction uses a satisfaction algorithm to simultaneously solve systems of non-linear equations. The constraints are generated from the line-drawing (e.g. a hidden line must cross behind a visible line), the world constraints (e.g. faces are planar polygons) and user constraints. The latter class of constraints represent geometric constraints placed on the model by the user (e.g. two sides have equal lengths).

Viking's combination of sketch interpretation and constraint satisfaction creates a surprisingly effective user-interface. At each step in the creation of a model, the designer is simply changing the sketch to make it look right: each step is obvious from context. This leaves the designer free to concentrate on the design itself and not how to convey it to the computer. Sketches, however, lack the precision needed later in the design process. Viking circumvents this problem by letting the designer place geometric constraints on the object thereby allowing the designer to turn a rough sketch into a precisely dimensioned object.

2 Related work

Viking is a solid modeling system whose user-interface is based on interactive sketch interpretation system. In this respect, Viking is unique: this is the first system to do sketch interpretation interactively. Viking, however, does use two well established techniques to create its user-interface: line-labeling and constraint satisfaction. Line-labeling systems have been around for over 20 years [9] [5]. Constraint satisfaction systems have been around even longer [14].

2.1 Line-labeling systems

The first line-labeling systems were developed separately by Huffman [9] and Clowes [5]. These algorithms reduced the problem of determining whether a line-drawing represented a trihedral object[1] to one of finding mutually consistent labelings for each intersection in the line-drawing. Since then, this basic algorithm has been extended to a wide variety of objects and line-drawings: scenes containing shadows [15], line-drawings in which hidden-lines are visible [12], objects with curved surfaces [4], and thin-shell objects [10]. Also, Waltz [15] introduced an efficient algorithm for filtering out obviously inconsistent labelings, and Sugihara [13] developed a way to test the feasibility of an interpretation by using linear programming.

2.2 Constraint satisfaction systems

Constraint solvers generally fall into two broad categories: incremental solvers and simultaneous solvers. The first type, perhaps best represented by ThingLab [2], essentially modify one variable at a time, propagating the results through the constraint network. These systems work well when dealing with acyclic networks of constraints, but may have problems with networks of cyclic constraints, iterating indefinitely on some networks. The second type, which describes Viking and Juno [11] among others, attempts to find a solution which simultaneously satisfies all of the constraints, typically using some form of gradient descent to minimize the total error. These systems have the problem that they require an initial position, and that they are not guaranteed to find a solution. For example, they may become trapped in local minima and never find a solution that satisfies all of the constraints. Given an initial position close to a solution, however, these system will normally find an exact solution, even if there are cyclic constraints.

3 Using Viking

Viking is an implementation of a what you see is what you get user-interface for solid modeling. What Viking's users see is a line-drawing, and what the users get is a three-dimensional object description. Viking first creates a line-drawing from the current object description and the current view transform. The users describe what they want to see by changing the line-drawing. Viking then uses sketch interpretation to generate a new object description which is consistent with the modified line-drawing. This object description, if it is acceptable to the user, replaces the current object description and a new line-drawing is created. The user can then modify the new line-drawing, starting a new cycle of modification and interpretation.

3.1 Modifying the line-drawing

A substantial portion of Viking is devoted to creating and modifying line-drawings. Creating the line-drawing is straight-forward: apply a view transform to the current object and generate a hidden-lines-dashed image. The algorithms used to create and display this image differ from the standard algorithms only in that special symbols are used to display information about the object's topology: thick, thin, or double lines are used to represent, respectively, edges adjacent to zero, one or two faces.

The user can modify the line-drawing in a variety of ways. For example, the user can click on a line-segment to toggle its visibility flag, making a hidden line-segment visible or vis versa. When the modified

[1] One in which each vertex is adjacent to exactly three faces.

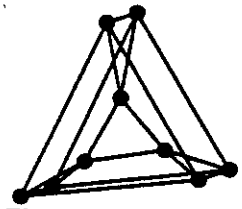

Figure 1a: Draw the wire-frame.

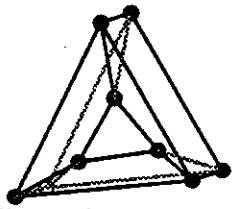

Figure 1b: Modify the line-drawing.

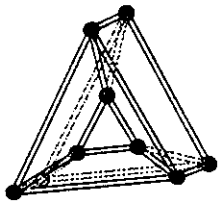

Figure 1c: Find an interpretation.

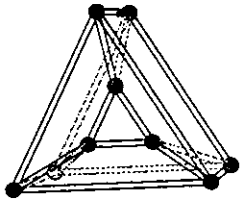

Figure 1d: Add geometric constraints.

line-drawing is displayed, the selected line-segment will be drawn with its new visibility, even though this is inconsistent with the underlying object description.

3.2 Interpreting a modified line-drawing

Once the user has modified the line-drawing, Viking must find a new interpretation. Unfortunately, line-drawings can have multiple valid interpretations. For example, a line-drawing of a block could represent either a solid block or a box which has an opening on the far side. For interactive systems, this topologic ambiguity presents a serious problem: users have both a particular topology in mind and little patience. In order for an interface to be successful, it must generate the desired topology quickly and easily.

Viking uses heuristics to control the order in which interpretations are generated, with the intent that the first interpretation is, at least, close to the interpretation desired by the user. These heuristics assign a cost to each interpretation based on the user's stated preferred object type (solid, thin-shell or no preference) and the current object. Interpretations are then generated in order of increasing cost. Once an interpretation has been generated, the user can either accept or reject it. In the latter case, the user can guide the search for subsequent interpretations by indicating where the current interpretation is incorrect.

Once an acceptable interpretation of the line-drawing has been found, geometric constraint satisfaction is used to find the position of the vertices. Constraints fall into three classes: world constraints, image constraints and user-specified constraints. World constraints force faces on the new object to be planar polyhedra. Image constraints force the hidden parts of the new object to be behind the visible parts, with respect to the current view transform. User-specified constraints force the new object to have the geometric properties desired by the user.

The positions of the vertices in the current object description are used as the initial positions for the vertices in the new object description. The vertices are then moved to satisfy the system of constraints using a non-linear satisfaction algorithm developed by Bullard and Biegler [3]. Although there is no explicit constraint that a solution be close to the initial values, motion in the problem space is well damped and solutions tend not to differ unnecessarily from the initial values.

3.3 Examples

Figures 1a-1d shows four steps in the process of creating a triangular torus. In Figure 1a, the user has drawn three triangles (each in a different plane) and connected the appropriate vertices to form a wire-frame for the torus. Figure 1b was created by hiding the line-segments which would be hidden if the drawing represented a solid object. Figure 1c is the first interpretation found for Figure 1b. The position of some of the vertices has shifted slightly in order to satisfy the constraint that each face is a planar polygon. Figure 1d shows the interpretation found after constraining every triangle in Figure 1c to be an equilateral triangle.

Figures 2a-2d provide an example of cutting a notch into a block. Figure 2a is the block before any changes have been made. In Figure 2b, the wire-frame for the notch has been added. Figure 2c is the second interpretation found for Figure 2b (in the first interpretation, line ab was adjacent to two surfaces). Figure 2d is the interpretation found after deleting the unneeded lines and adding the constraint that the angle at the bottom of the notch is 45°.

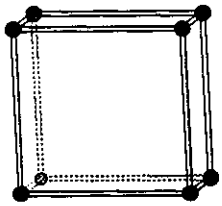

Figure 2a: Start with a simple block.

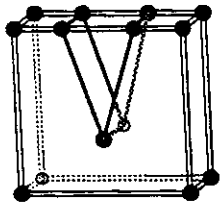

Figure 2b: Draw in a notch.

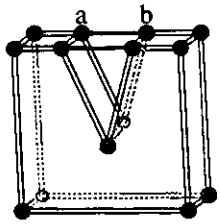

Figure 2c: Find an interpretation.

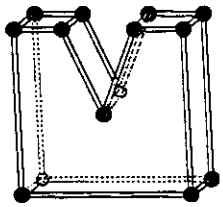

Figure 2d: Clean-up and add geometric constraints.

4 Finding the valid labelings for an arbitrary intersection

The first step in analyzing a line-drawing is to generate the valid labelings for each intersection in the drawing. A labeling represents one configuration of surfaces around an intersection and is simply the line-labels selected for each line adjacent to the intersection. A line's label describes the surfaces adjacent to the line. In traditional line-labeling, there are three possible labels: convex (where the line lies along a ridge formed by the two surfaces), concave (where the line lies at the bottom of valley formed by the two surfaces) and occluding (where both surfaces extend to one side of the line). In arc-labeling, a label tells how many surfaces extend to either side of a line and what the other bounding line is for each surface. The labels used by arc-labeling can represent lines with any number of adjacent surfaces, but they do not distinguish between convex and concave arrangements of adjacent surfaces.

The surfaces described in the labeling can be thought of as a solid arc extending left from one line around the intersection to the right of another line, and are called surface-fragments throughout the rest of the paper. A labeling is valid for an intersection if a physical model of the labeling can be constructed whose appearance matches the appearance of the intersection. Most line-labeling algorithms can only generate the valid labelings for intersections of two or three lines[2]. Arc-labeling can find the valid labelings for intersections with any number of adjacent lines.

With both labeling schemes, the goal is to find valid, mutually consistent labelings for each intersection in the line-drawing. In line-labeling, a line is consistently labeled if it has the same label at each end. In arc-labeling, a line is consistently labeled if the same number of surfaces extend to each side at both ends. If a consistent labeling can not be found, then the line-drawing represents an impossible object: one which does not have a physical interpretation.

Both line-labeling and arc-labeling assume that the line-drawing represents a general view of the object [12]. In essence, this is an assumption that a small change in the observer's view point will produce a correspondingly small change in the line-drawing. This assumption has two corollaries in Viking:

1. Every face is drawn as a closed polygon with a non-zero area.

2. Every edge is drawn as a line with a non-zero length.

These corollaries eliminate a lot of special cases (in particular, what is the left and right of a line with a length of zero?), but make it impossible to use Viking to analyze most engineering drawings since these drawings typically violate one or both of the corollaries.

4.1 Evaluating the validity of a labeling

A model of an intersection's labeling can be created by assigning a position to the endpoint of each adjacent line. Using this model, the visibility of each adjacent line can be calculated and the surface-fragments can be tested to see if they intersect anywhere except along a common line. If the endpoints can be positioned such that the appearance of the model is the same as the appearance of the intersection in the line-drawing and none of the surface-fragments intersect, then a physical model of the labeling exists and labeling is valid.

[2] In practice, the labelings are pre-calculated and stored in an intersection library (also called a junction library).

$$
\begin{array}{rcl}
a, b, \ldots & = & \text{lines} \\
\mathcal{X}, \mathcal{Y}, \ldots & = & \text{surface-fragments} \\
V & = & \text{Set of all visible lines adjacent to the intersection.} \\
H & = & \text{Set of all hidden lines adjacent to the intersection.} \\
S & = & \text{Set of all surface-fragments adjacent to the intersection.} \\
\mathrm{l}_{\mathcal{X}} & = & \text{The line forming the left boundary of } \mathcal{X}. \\
\mathrm{r}_{\mathcal{X}} & = & \text{The line forming the right boundary of } \mathcal{X}. \\
(a \propto \mathcal{X}) & = & \begin{cases} \text{True} & \text{If } \mathcal{X} \text{ spans } a \text{ } (a \text{ lies between the bounding lines of } \mathcal{X}, \\ & & a \neq \mathrm{l}_{\mathcal{X}} \text{ and } a \neq \mathrm{r}_{\mathcal{X}}). \\ \text{False} & \text{otherwise} \end{cases} \\
(a \Uparrow \mathcal{X}) & = & \begin{cases} \text{True} & \text{If } a \text{ is in front of the plane defined by } \mathcal{X}. \\ \text{False} & \text{otherwise} \end{cases} \\
(a \Downarrow \mathcal{X}) & = & \begin{cases} \text{True} & \text{If } a \text{ is behind the plane defined by } \mathcal{X}. \\ \text{False} & \text{otherwise} \end{cases} \\
(\mathcal{X} \otimes \mathcal{Y}) & = & \begin{cases} \text{True} & \mathcal{X} \text{ and } \mathcal{Y} \text{ do not intersect except along a common line.} \\ \text{False} & \text{otherwise} \end{cases}
\end{array}
$$

Table 1: Notation key for validity expressions.

Using the notation given in Table 1, first order logic expressions can be written which correspond to the criteria given above. These expressions (along with their English translations) are as follows:

- $(\forall v \in V, (\forall \mathcal{X} \in S \mid (v \propto \mathcal{X}), (v \Uparrow \mathcal{X})))$ (1)
  Every visible line is in front of every spanning surface-fragment.
- $((V = \emptyset) \vee (\forall h \in H, (\exists \mathcal{X} \in S \mid (h \propto \mathcal{X}), (h \Downarrow \mathcal{X}))))$ (2)
  Either the entire intersection is hidden by a non-adjacent surface or
  every hidden line is behind at least one spanning surface-fragment.
- $(\forall \mathcal{X} \in S, (\forall \mathcal{Y} \in S \mid (\mathcal{Y} \neq \mathcal{X}), (\mathcal{X} \otimes \mathcal{Y})))$ (3)
  No two distinct pairs of surface-fragments intersect anywhere except along a common line.

If all three of these expressions can be satisfied, then the labeling is considered valid because it is possible to create a model of the labeling which matches the appearance of the intersection in the line-drawing.

Of all the predicates used to write these expressions, only the $\Uparrow$ and $\Downarrow$ present significant problems. While the $\otimes$ predicate is fairly complex, it can be rewritten in terms of the other predicates, as shown in Table 2. Table 2 was generated by enumerating the possible ways in which two surface-fragments can share a common point without intersecting one another. Figures 3a - 11 show all non-intersecting arrangements of two surface-fragments. The remaining predicates can be evaluated based on the angle each line makes with the horizontal, which is known.

4.1.1 Evaluating the $\Uparrow$ and $\Downarrow$ predicates

The satisfiability of an expression containing the $\Uparrow$ and $\Downarrow$ predicates can be evaluated if the expression can be written in the following form:

$$C = (a_1 \Uparrow \mathcal{X}_1) \wedge (a_2 \Uparrow \mathcal{X}_2) \wedge \cdots \wedge (a_n \Uparrow \mathcal{X}_n) \wedge (a_{n+1} \Downarrow \mathcal{X}_{n+1}) \wedge \cdots \wedge (a_m \Downarrow \mathcal{X}_m) \quad (4)$$

$$
\begin{array}{rcll}
\angle a & = & \text{The angle } a \text{ makes with the horizontal.} & \\
\angle ab & = & \text{The angle counterclockwise from } a \text{ to } b. & \\
\angle \mathcal{X} & = & \text{The angle between the bounding lines of } \mathcal{X}. & \\
(a \not\propto \mathcal{X}) & = & \neg(a \propto \mathcal{X}) & \\
 & & & \\
(\mathcal{X} \otimes \mathcal{Y}) & = & \eta(\mathcal{X},\mathcal{Y}) \vee \kappa(\mathcal{X},\mathcal{Y}) \vee \varrho(\mathcal{X},\mathcal{Y}) \vee \tau(\mathcal{X},\mathcal{Y}) \vee \psi(\mathcal{X},\mathcal{Y}) \vee \omega(\mathcal{X},\mathcal{Y}) \vee & \\
 & & \eta(\mathcal{Y},\mathcal{X}) \vee \kappa(\mathcal{Y},\mathcal{X}) \vee \varrho(\mathcal{Y},\mathcal{X}) \vee \tau(\mathcal{Y},\mathcal{X}) \vee \psi(\mathcal{Y},\mathcal{X}) \vee \omega(\mathcal{Y},\mathcal{X}) & \\
 & & & \\
\eta(\mathcal{X},\mathcal{Y}) & = & (\mathrm{l}_{\mathcal{X}} \propto \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{X}} \propto \mathcal{Y}) \wedge (\mathrm{l}_{\mathcal{Y}} \propto \mathcal{X}) \wedge (\mathrm{r}_{\mathcal{Y}} \propto \mathcal{X}) \wedge (\eta_1(\mathcal{X},\mathcal{Y}) \vee \eta_2(\mathcal{X},\mathcal{Y})) & \\
\eta_1(\mathcal{X},\mathcal{Y}) & = & (\angle\mathcal{X} < 180°) \wedge & \\
 & & (((\mathrm{l}_{\mathcal{X}} \Uparrow \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{X}} \Uparrow \mathcal{Y}) \wedge (\mathrm{l}_{\mathcal{Y}} \Downarrow \mathcal{X}) \wedge (\mathrm{r}_{\mathcal{Y}} \Downarrow \mathcal{X})) \vee & \text{[Figure 3a]} \\
 & & ((\mathrm{l}_{\mathcal{X}} \Downarrow \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{X}} \Downarrow \mathcal{Y}) \wedge (\mathrm{l}_{\mathcal{Y}} \Uparrow \mathcal{X}) \wedge (\mathrm{r}_{\mathcal{Y}} \Uparrow \mathcal{X})) \vee & \text{[Figure 3b]} \\
 & & ((\mathrm{l}_{\mathcal{X}} \Uparrow \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{X}} \Downarrow \mathcal{Y}) \wedge (\mathrm{l}_{\mathcal{Y}} \Downarrow \mathcal{X}) \wedge (\mathrm{r}_{\mathcal{Y}} \Uparrow \mathcal{X})) \vee & \text{[Figure 3c]} \\
 & & ((\mathrm{l}_{\mathcal{X}} \Downarrow \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{X}} \Uparrow \mathcal{Y}) \wedge (\mathrm{l}_{\mathcal{Y}} \Uparrow \mathcal{X}) \wedge (\mathrm{r}_{\mathcal{Y}} \Downarrow \mathcal{X}))) & \text{[Figure 3d]} \\
\eta_2(\mathcal{X},\mathcal{Y}) & = & (\angle \mathrm{r}_{\mathcal{X}}\mathrm{l}_{\mathcal{Y}} < 180°) \wedge (\angle \mathrm{r}_{\mathcal{Y}}\mathrm{l}_{\mathcal{X}} < 180°) \wedge & \\
 & & (((\mathrm{l}_{\mathcal{X}} \Downarrow \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{X}} \Uparrow \mathcal{Y}) \wedge (\mathrm{l}_{\mathcal{Y}} \Uparrow \mathcal{X}) \wedge (\mathrm{r}_{\mathcal{Y}} \Downarrow \mathcal{X})) \vee & \text{[Figure 4a]} \\
 & & ((\mathrm{l}_{\mathcal{X}} \Uparrow \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{X}} \Downarrow \mathcal{Y}) \wedge (\mathrm{l}_{\mathcal{Y}} \Downarrow \mathcal{X}) \wedge (\mathrm{r}_{\mathcal{Y}} \Uparrow \mathcal{X}))) & \text{[Figure 4b]} \\
 & & & \\
\kappa(\mathcal{X},\mathcal{Y}) & = & (\mathrm{l}_{\mathcal{X}} \propto \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{X}} \propto \mathcal{Y}) \wedge (\mathrm{l}_{\mathcal{Y}} \not\propto \mathcal{X}) \wedge (\mathrm{r}_{\mathcal{Y}} \not\propto \mathcal{X}) \wedge & \\
 & & (\angle\mathcal{X} < 180°) \wedge & \\
 & & (((\mathrm{l}_{\mathcal{X}} \Uparrow \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{X}} \Uparrow \mathcal{Y})) \vee & \text{[Figure 5a]} \\
 & & ((\mathrm{l}_{\mathcal{X}} \Downarrow \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{X}} \Downarrow \mathcal{Y}))) & \text{[Figure 5b]} \\
 & & & \\
\varrho(\mathcal{X},\mathcal{Y}) & = & (\mathrm{l}_{\mathcal{X}} \propto \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{X}} \not\propto \mathcal{Y}) \wedge (\mathrm{l}_{\mathcal{Y}} \not\propto \mathcal{X}) \wedge (\mathrm{r}_{\mathcal{Y}} \propto \mathcal{X}) \wedge (\varrho_1(\mathcal{X},\mathcal{Y}) \vee \varrho_2(\mathcal{X},\mathcal{Y})) & \\
\varrho_1(\mathcal{X},\mathcal{Y}) & = & (\mathrm{l}_{\mathcal{Y}} \neq \mathrm{r}_{\mathcal{X}}) \wedge (\angle \mathrm{r}_{\mathcal{Y}}\mathrm{l}_{\mathcal{X}} < 180°) \wedge & \\
 & & (((\mathrm{l}_{\mathcal{X}} \Uparrow \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{Y}} \Downarrow \mathcal{X})) \vee & \text{[Figure 6a]} \\
 & & ((\mathrm{l}_{\mathcal{X}} \Downarrow \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{Y}} \Uparrow \mathcal{X}))) & \text{[Figure 6b]} \\
\varrho_2(\mathcal{X},\mathcal{Y}) & = & (\mathrm{l}_{\mathcal{Y}} = \mathrm{r}_{\mathcal{X}}) \wedge (\angle\mathcal{X} < 180°) \wedge (\angle\mathcal{Y} < 180°) \wedge & \\
 & & (((\mathrm{l}_{\mathcal{X}} \Uparrow \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{Y}} \Downarrow \mathcal{X})) \vee & \text{[Figure 7a]} \\
 & & ((\mathrm{l}_{\mathcal{X}} \Downarrow \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{Y}} \Uparrow \mathcal{X}))) & \text{[Figure 7b]} \\
 & & & \\
\tau(\mathcal{X},\mathcal{Y}) & = & (\mathrm{l}_{\mathcal{X}} \propto \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{X}} \not\propto \mathcal{Y}) \wedge (\mathrm{l}_{\mathcal{Y}} \not\propto \mathcal{X}) \wedge (\mathrm{r}_{\mathcal{Y}} \not\propto \mathcal{X}) \wedge & \\
 & & (\angle\mathcal{X} < 180°) \wedge & \\
 & & ((\mathrm{l}_{\mathcal{X}} \Uparrow \mathcal{Y}) \vee & \text{[Figure 8a]} \\
 & & (\mathrm{l}_{\mathcal{X}} \Downarrow \mathcal{Y})) & \text{[Figure 8b]} \\
 & & & \\
\psi(\mathcal{X},\mathcal{Y}) & = & (\mathrm{l}_{\mathcal{X}} \not\propto \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{X}} \propto \mathcal{Y}) \wedge (\mathrm{l}_{\mathcal{Y}} \not\propto \mathcal{X}) \wedge (\mathrm{r}_{\mathcal{Y}} \not\propto \mathcal{X}) \wedge & \\
 & & (\angle\mathcal{X} < 180°) \wedge & \\
 & & ((\mathrm{r}_{\mathcal{X}} \Uparrow \mathcal{Y}) \vee & \text{[Figure 9a]} \\
 & & (\mathrm{r}_{\mathcal{X}} \Downarrow \mathcal{Y})) & \text{[Figure 9b]} \\
 & & & \\
\omega(\mathcal{X},\mathcal{Y}) & = & (\mathrm{l}_{\mathcal{X}} \not\propto \mathcal{Y}) \wedge (\mathrm{r}_{\mathcal{X}} \not\propto \mathcal{Y}) \wedge (\mathrm{l}_{\mathcal{Y}} \not\propto \mathcal{X}) \wedge (\mathrm{r}_{\mathcal{Y}} \not\propto \mathcal{X}) \wedge & \\
 & & ((\mathrm{l}_{\mathcal{X}} \neq \mathrm{l}_{\mathcal{Y}}) \vee & \text{[Figure 10]} \\
 & & ((\mathrm{l}_{\mathcal{X}} = \mathrm{l}_{\mathcal{Y}}) \wedge (\angle\mathcal{X} = 180°))) & \text{[Figure 11]}
\end{array}
$$

Table 2: Boolean expansion of $(\mathcal{X} \otimes \mathcal{Y})$

This expression is obviously unsatisfiable if it contains $(a_i \Uparrow \mathcal{X}_i)$ and $(a_j \Downarrow \mathcal{X}_j)$ terms, where $a_i = a_j$ and $\mathcal{X}_i = \mathcal{X}_j$. Expressions which are potentially satisfiable are evaluated using an algorithm similar to the one developed by Sugihara [13] to test the feasibility of a line-drawing's interpretation.

The first step is to create vectors for each line and then to find the normal vector for each surface-fragment. These vectors and normals are created as follows:

For each $a \in \bigcup_{i=1}^{m} \{a_i\}$, create a vector:

$$\vec{p}_a = (\cos \angle a, \sin \angle a, z_a), \text{ where } z_a \text{ is unknown.}$$

($\vec{p}_a$ is the position of the endpoint of line a).

For each $\mathcal{X} \in \bigcup_{j=1}^{m} \{\mathcal{X}_j\}$, create a vector:

$$\vec{n}_{\mathcal{X}} = \begin{cases} (\vec{p}_{\mathrm{r}_{\mathcal{X}}} \times \vec{p}_{\mathrm{l}_{\mathcal{X}}}) & \text{if } (\angle \mathcal{X} < 180°) \\ (\vec{p}_{\mathrm{r}_{\mathcal{X}}} \times (\cos \angle \mathrm{r}_{\mathcal{X}} + 90°, \sin \angle \mathrm{r}_{\mathcal{X}} + 90°, \zeta_{\mathcal{X}})), \text{ where } \zeta_{\mathcal{X}} \text{ is unknown} & \text{if } (\angle \mathcal{X} = 180°) \\ (\vec{p}_{\mathrm{l}_{\mathcal{X}}} \times \vec{p}_{\mathrm{r}_{\mathcal{X}}}) & \text{otherwise} \end{cases}$$

($\vec{n}_{\mathcal{X}}$ is an upward-pointing normal for the plane defined by $\mathcal{X}$).

The dot product of $\vec{p}_i$ and $\vec{n}_{\mathcal{X}}$ can be used to determine whether $\vec{p}_i$ is above, on, or below $\mathcal{X}$. If the dot product is positive, $\vec{p}_i$ is above $\mathcal{X}$; if it is zero, $\vec{p}_i$ is on $\mathcal{X}$; otherwise, $\vec{p}_i$ is below $\mathcal{X}$. Using this, the predicates in C can be evaluated as follows:

For each predicate $(a_i \Uparrow \mathcal{X}_i)$ $(1 \leq i \leq n)$, create the equation: $\vec{n}_{\mathcal{X}_i} \cdot \vec{p}_{a_i} > 0$

For each predicate $(a_j \Downarrow \mathcal{X}_j)$ $(n < j \leq m)$, create the equation: $\vec{n}_{\mathcal{X}_j} \cdot \vec{p}_{a_j} < 0$

For each $\mathcal{X} \in \bigcup_{j=1}^{m} \mathcal{X}_j \mid \angle \mathcal{X} = 180°$, create the equation: $z_{r_{\mathcal{X}}} + z_{l_{\mathcal{X}}} = 0$

The last equation is based on the general view assumption (see Section 4). By that assumption, two bounding lines which appear parallel must be parallel.

Even though the first two equations are linear, they are not in a form which can be solved using linear programming. These equations can be converted for forms which are amiable to linear programming by replacing > 0 and < 0 with $\geq \varepsilon$ and $\leq -\varepsilon$ respectively (ε is any positive constant). This modified system of equations is equivalent to the original system[3]. If this modified system of equations has a solution, then there is a way to arrange the lines and surface-fragments so that the relations described in C are satisfied.

4.1.2 Evaluating the validity criteria

For any intersection, the validity of an arbitrary labeling can be determined as follows:

1. Generate boolean expressions corresponding to expressions (1) - (3), based on the intersection and the labeling.
2. Combine the three boolean expressions into a single expression (ANDing the three clauses together).
3. Simplify the boolean expression by:
   - rewriting the $\otimes$ predicate expressions in terms of the other predicates and
   - evaluating all except the $\Uparrow$ and $\Downarrow$ predicates.

[3]Proof: A solution to the modified system of equations is a solution to the original system. Any solution to the original system can be multiplied a large enough constant that it constitutes a solution to the modified system. Therefore, if a solution exists to the original system, a solution exists to the modified system

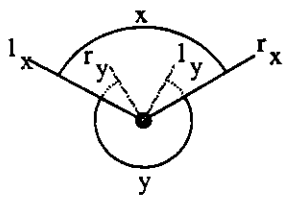

Figure 3a:
$((l_x \Uparrow \mathcal{Y}) \wedge (r_x \Uparrow \mathcal{Y}) \wedge$
$(l_y \Downarrow \mathcal{X}) \wedge (r_y \Downarrow \mathcal{X}))$

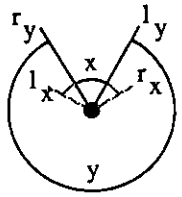

Figure 3b:
$((l_x \Downarrow \mathcal{Y}) \wedge (r_x \Downarrow \mathcal{Y}) \wedge$
$(l_y \Uparrow \mathcal{X}) \wedge (r_y \Uparrow \mathcal{X}))$

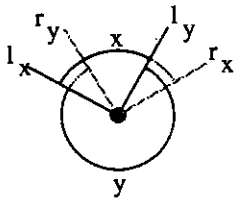

Figure 3c:
$((l_x \Uparrow \mathcal{Y}) \wedge (r_x \Downarrow \mathcal{Y}) \wedge$
$(l_y \Downarrow \mathcal{X}) \wedge (r_y \Uparrow \mathcal{X}))$

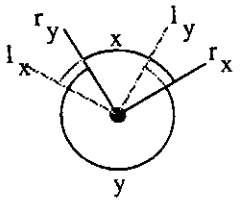

Figure 3d:
$((l_x \Downarrow \mathcal{Y}) \wedge (r_x \Uparrow \mathcal{Y}) \wedge$
$(l_y \Uparrow \mathcal{X}) \wedge (r_y \Downarrow \mathcal{X}))$

Figures 3a-3d: Double overlapping small-large surface-fragments.

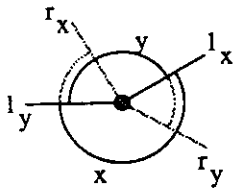

Figure 4a:
$((l_x \Downarrow \mathcal{Y}) \wedge (r_x \Uparrow \mathcal{Y}) \wedge$
$(l_y \Uparrow \mathcal{X}) \wedge (r_y \Downarrow \mathcal{X}))$

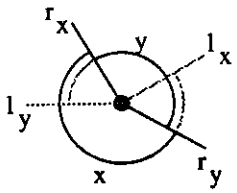

Figure 4b:
$((l_x \Downarrow \mathcal{Y}) \wedge (r_x \Uparrow \mathcal{Y}) \wedge$
$(l_y \Downarrow \mathcal{X}) \wedge (r_y \Uparrow \mathcal{X}))$

Figures 4a-4b: Double overlapping large-large surface-fragments.

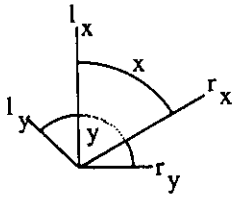

Figure 5a:
$(l_x \Uparrow \mathcal{Y}) \wedge (r_x \Uparrow \mathcal{Y})$

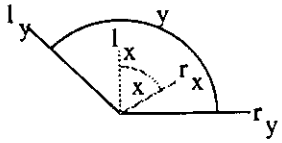

Figure 5b:
$(l_x \Downarrow \mathcal{Y}) \wedge (r_x \Downarrow \mathcal{Y})$

Figures 5a-5b: Fully overlapped surface-fragments.

4. Rewrite the boolean expression in disjunctive normal form. This expression will have the form:

$$B = C_1 \vee C_2 \vee \cdots \vee C_n \tag{5}$$

where each C_i is an expression of the form given by equation (4).

5. Determine the satisfiability of each C_i in B, using the algorithm described in Section 4.1.1. If any of the C_i's are satisfiable, then the labeling represents a valid arrangement of surface-fragments around the intersection. Otherwise, the labeling is invalid.

4.2 Intersection types

The set of valid labelings is relatively insensitive to the exact position of the lines around the intersection. Two different intersections will have the same valid labelings if:

- the corresponding pairs of lines around each intersection have the same angular sense (less than 180°, equal to 180° or greater than 180°) and
- the corresponding lines around each intersection are both either visible or hidden.

As a result, it is possible to create an intersection library. For each distinct arrangement of lines around an intersection, generate all possible labelings and use the algorithm described in the previous section to reject the invalid ones. Intersection libraries improve performance, since it takes less time to find the valid labelings in an intersection library than it does to generate them from scratch. Figures 12a-12k show all distinct arrangements of four or fewer lines around an intersection (unless mentioned otherwise, the angle between adjacent lines is less than 180°).

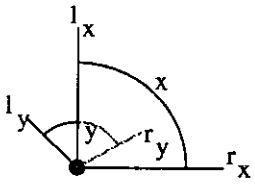

Figure 6a:
$(l_{\mathcal{X}} \Uparrow \mathcal{Y}) \wedge (r_{\mathcal{Y}} \Downarrow \mathcal{X})$

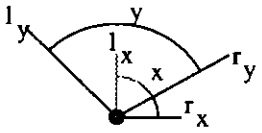

Figure 6b:
$(l_{\mathcal{X}} \Uparrow \mathcal{Y}) \wedge (r_{\mathcal{Y}} \Downarrow \mathcal{X})$

Figures 6a-6b: Partially overlapping surface-fragments.

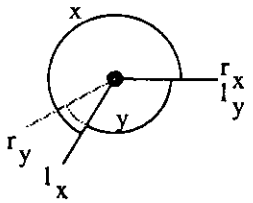

Figure 7a:
$(l_{\mathcal{X}} \Uparrow \mathcal{Y}) \wedge (r_{\mathcal{Y}} \Downarrow \mathcal{X})$

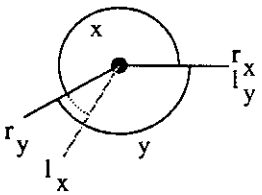

Figure 7b:
$(l_{\mathcal{X}} \Downarrow \mathcal{Y}) \wedge (r_{\mathcal{Y}} \Uparrow \mathcal{X})$

Figures 7a-7b: Partially overlapping joined surface-fragments.

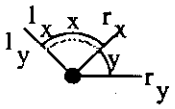

Figure 8a:
$(l_{\mathcal{X}} \Uparrow \mathcal{Y})$

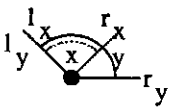

Figure 8b:
$(l_{\mathcal{X}} \Downarrow \mathcal{Y})$

Figures 8a-8b: Right folded surface-fragments.

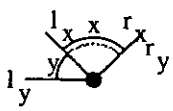

Figure 9a:
$(l_{\mathcal{X}} \Uparrow \mathcal{Y})$

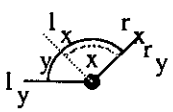

Figure 9b:
$(l_{\mathcal{X}} \Downarrow \mathcal{Y})$

Figures 9a-9b: Left folded surface-fragments.

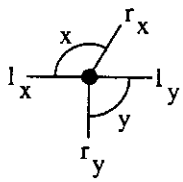

Figure 10: Disjoint surface-fragments.

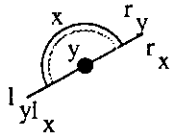

Figure 11: Folded half surface-fragments.

5 Generating a topology from a line-drawing

Generating a surface topology – a list of the vertices forming each face in the proposed object – from a line-drawing is a four step process. First, restrictions on the possible labels for each line are found using features in the line-drawing. Second, the valid labelings for each intersection in the line-drawing are found using the algorithm given in Section 4. Third, the entire line-drawing is consistently labeled using a branch and bound search. Fourth, a surface topology is found using the labeling found in the previous step.

Once a surface topology has been found, it is displayed to the user. The user can then either accept or reject the topology. If the user accepts it, the surface topology is used to create a new object. Otherwise, steps three and four are repeated and a new topology is generated.

5.1 Restricting the possible labelings for a line

For any given line-drawing, some labelings which might be valid based solely on an analysis of the intersections might be inappropriate based on global features in the line-drawing. In Figure 13, for example, line $\overline{ab}$ partially obscures two other lines. In order for this to occur, line $\overline{ab}$ must have one or more surfaces extending to its upper right and no surfaces extending to its lower left. As a result, any labeling for intersections a or b in which line ab fails to meet this condition is considered inappropriate. The following rules can be used to reject inappropriate labelings:

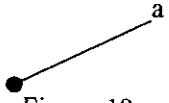

Figure 12a:
Arrow.1.

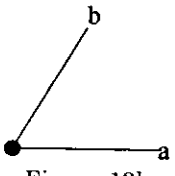

Figure 12b:
Arrow.2
($\angle ba > 180°$).

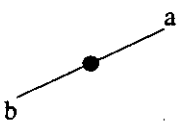

Figure 12c:
Straight.2
($\angle ba = 180°$).

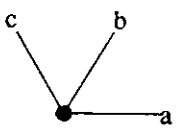

Figure 12d:
Arrow.3
($\angle ca > 180°$).

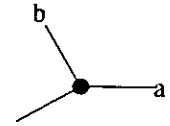

Figure 12e:
Fork.3.

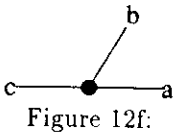

Figure 12f:
T.3
($\angle ca = 180°$ and $\angle ba > 180°$).

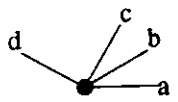

Figure 12g: Arrow.4 ($\angle da > 180°$).

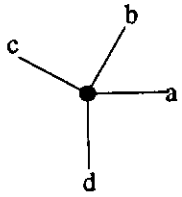

Figure 12h: Fork.4 ($\angle ca > 180°$ and $\angle bd > 180°$).

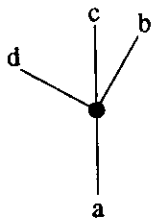

Figure 12i: Psi.4 ($\angle ca = 180°$ and $\angle db > 180°$).

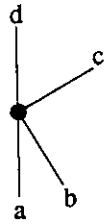

Figure 12j: K.4 ($\angle da = 180°$ and $\angle cb > 180°$).

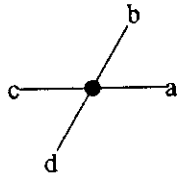

Figure 12k: X.4 ($\angle ca = 180°$ and $\angle bd = 180°$).

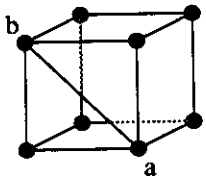

Figure 13: Discarding inappropriate labelings.

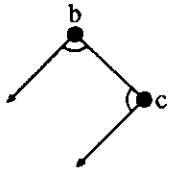

Figure 14a: Two surface-fragment merge

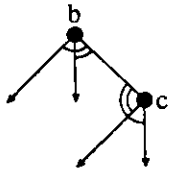

Figure 14b: Four surface-fragment merge

1. Crossing-rule 1
   If two lines cross and one of the line-segments adjacent to the crossing is hidden, then the obscuring line must have one or more adjacent surfaces extending over the obscured line-segment and no adjacent surfaces extending over the exposed line-segment.

2. Crossing-rule 2
   If two lines cross and every line-segment adjacent to the crossing is visible, then at least one of the lines must not be adjacent to any surfaces.

   These rules assume that intersections, where two or more lines meet at a point, can be distinguished from crossings, where two lines pass over one-another in the line-drawing. In Viking, this is easy to do because there is a one to one mapping between intersections in the line-drawing and vertices in the current object.

3. Perimeter-rule
   A line-segment on the perimeter of the line-drawing can only be adjacent to surfaces extending to the inside of the line-drawing.

   This rule assumes that the entire object is contained within the boundaries of the line drawing and that none of the object's faces contain holes.

5.2 Generating the correct topology

An object's topology normally is not uniquely specified by the line-drawing. For example, the line-drawing Figure 13 has 19 different topologies. Viking handles this problem by using heuristics to assign a cost to each possible labeling for each intersection. Topologies are then generated in order of increasing total cost.

5.2.1 Assigning costs to the labelings

Three heuristics are used to determine the cost of an intersection's labeling. The first heuristic penalizes labelings which are different from the labeling for the intersection in the current object description. The second heuristic factors in a bias towards topologies preferred by the user. For example the user can set a preference for solid topologies, in which case labelings which have non-continuous surfaces are penalized.

The third and final heuristic factors in a bias towards plausible configurations of surfaces. In particular, labelings are penalized in which one or more vertices must move in order to satisfy the image constraints.

5.2.2 Searching for a consistent labeling

Once a cost has been assigned to each labeling for each intersection, a search is conducted for the least expensive, consistent labeling for the entire line-drawing. This search is performed creating a branch and bound search tree. This tree is created by repeatedly expanding the leaf which has the lowest cost. There are two types of leaves: intermediate leaves and terminal leaves. The former correspond to partial labelings for the line-drawing. The latter are leaves in which a labeling has been assigned to every intersection in the line-drawing. Intermediate leaves are expanded by finding an unlabeled intersection and generating a new leaf for each of the intersection's labeling that is consistent with the labelings selected for the leaf's antecedents. Terminal leaves are expanded by generating the topology corresponding to the assigned labelings and allowing the user to accept or reject the topology.

A leaf's cost is an estimate of the total cost to label every intersection in the line-drawing. This cost can be calculated by adding the costs of the labelings already selected for a leaf's antecedents to an estimate of the remaining cost. This remaining cost is found by adding the cost of finding the least expensive labelings for the remaining intersections which are consistent with the leaf's antecedents (but which may be mutually inconsistent). This cost estimate is optimistic (i.e. the true cost will always be as high or higher than the estimate). As a result, this algorithm is guaranteed to generate consistent labelings for the entire line-drawing in order of increasing total cost[4].

5.2.3 Generating a topology from a labeling

The labeling selected for each intersection describes the intersection's adjacent surfaces. This description, however, is very limited: for each surface, the labeling contains only the left and right bounding lines for the surface at the intersection. Creating a surface topology requires merging the individual surface-fragments into closed, polygonal surfaces, which then become the object's faces. In Figure 14a, for example, the surface-fragments adjacent to intersections *b* and *c* can be merged to form a larger surface-fragment. If, as in Figure 14b, a line has two or more surfaces extending to a side, there are two possible ways to merge the surfaces. Viking solves this problem by trying both combinations. Only one of the combinations will lead to a valid set of surfaces (selecting the other combination will result in one or more open surfaces after all surface-fragments have been combined).

While this algorithm will always find the topology which corresponds to a labeling, not every topology is reasonable. If a topology contains a face in which an edge extends across the inside of the face between two of the face's vertices, then the topology is considered unreasonable and is rejected. The reason for this is that an edge between two of a face's vertices must be co-planar with the face. If the edge also extends across the inside of the face then it is impossible to satisfy either the *in front of* or the *behind* constraints used by the geometric constraint solver.

5.3 Selecting the topology

Once a reasonable topology has been found, it is displayed to the user and he or she is given a chance to accept or reject it. In the former case, an attempt is made to find a satisfactory object geometry for the topology. In the latter, a new topology is found. There are three ways in which the user can reject a topology. First, the user can simply reject the labeling, in which case the next least expensive labeling is found. Second, the user can reject the labeling, but accept the labeling for a single line. In this case, the next least expensive labeling is found in which the number of surfaces adjacent to the selected line does not change. Third, the user can reject the labeling for a single line, forcing the system to find the next

[4]Proof: The cost of a terminal leaf is the true cost for labeling the line-drawing. In order for a terminal leaf to be expanded, its cost must be as low or lower than the cost of any other leaf in the tree. And, since the cost of intermediate leaves are optimistic, the true cost of any other labeling must be at least as high as the cost of the terminal leaf.

least expensive labeling in which the number of surfaces adjacent to the selected line changes. The latter technique seems to be the most effective one for finding a desired topology.

6 Finding the geometry from a line-drawing

Once a topology has been accepted by the user, the next problem is to find appropriate positions for all of the vertices in the new object description. In Viking, this is done by solving a system of non-linear equations which correspond to geometric constraints on the new object. These geometric constraints fall into three broad categories:

World The constraint that each face in the proposed object description in planar.

Image The constraint that the visible/hidden relations implied by the image are satisfied.

- At any crossing (where two lines cross one another without physically intersecting), the visible line must be in front of the hidden or partially hidden line.
- At any visible intersection, visible lines must be in front of any spanning surfaces and hidden lines must be behind at least one spanning surface. This is similar to the first two validity criteria given in Section 4.1.

User Any geometric constraints placed by the user.

If a solution is found to the system of geometric constraints, then the new interpretation becomes the new current object and the user can continue the modify and interpret cycle. Otherwise, the user can either search for a new topology or abort the search for a geometry. In the latter case, the topology from the new interpretation is combined with the geometry from the current object (and, as a result, some of the geometric constraints will not be satisfied). The user can then modify the object (by moving a vertex closer to its desired position, for example) and re-run the constraint solver on the modified object.

6.1 Equations used to represent the constraints

Currently, Viking supports five different types of constraints. These constraints are listed below, along with the equations used to represent them. A constraint is satisfied when all of its corresponding equations are satisfied.

- Planarity constraint: Planar($\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n$)
  $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n$ all lie in the same plane.

  To add a planarity constraint, create four new variables: a, b, c, d,
  add the equation: $a^2 + b^2 + c^2 = 1$ and
  for each i such that $1 \leq i \leq n$, add the equation:
  $$ax_i + by_i + cz_i + d = 0.$$

- Line *in front of* line: LineFront($\vec{v}_1, \vec{v}_2, \vec{v}_3, \vec{v}_4, T$)
  Line $\vec{v}_1\vec{v}_2$ passes in front of line $\vec{v}_3\vec{v}_4$ after applying view transform T.

  To add a line *in front of* line constraint, add the equation: $p \geq \varepsilon$
  Where:
  p equals $(\vec{n} \cdot (\vec{v}_1 - \vec{v}_3))/(\vec{n} \cdot \vec{T}_z)$,
  n equals $((\vec{v}_2 - \vec{v}_1) \times (\vec{v}_4 - \vec{v}_3))$,
  T_z is vector pointing to the eye in T and
  ε is the minimum separation distance.

- Point *in front of* plane: PointFront($\vec{v}_p, \vec{v}_1, \vec{v}_2, \vec{v}_3, T$)
  $\vec{v}_p$ is in front of the plane defined by $\vec{v}_1, \vec{v}_2 \& \vec{v}_3$ after applying view transform T.

To add a point *in front of* plane constraint, add the equation: $p \geq \varepsilon$
Where:

p equals $(\vec{n} \cdot (\vec{v}_p - \vec{v}_2))/(\vec{n} \cdot \vec{T}_z)$,
n equals $((\vec{v}_1 - \vec{v}_2) \times (\vec{v}_3 - \vec{v}_2))$,
T_z is vector pointing to the eye in T and
ε is the minimum separation distance.

- Distance constraint: $D(k_1, \vec{v}_1, \vec{v}_2, \ldots, k_n, \vec{v}_{2n-1}, \vec{v}_{2n}, k_{n+1}, \vec{v}_{2n+1}, \vec{v}_{2n+2}, \vec{v}_{2n+3}, \ldots, k_{n+m}, \vec{v}_{2n+3m-2}, \vec{v}_{2n+3m-1}, \vec{v}_{2n+3m}) \begin{cases} \leq & d \\ = & d \\ \geq & d \end{cases}$

  The distances between the given pairs of vertices must satisfy the linear equation shown below.

  To add a distance constraint, add the following equation:

  $k_1 \mid \vec{v}_1 - \vec{v}_2 \mid + \ldots + k_n \mid \vec{v}_{2n-1} - \vec{v}_{2n} \mid +$
  $k_{n+1} L(\vec{v}_{2n+1}, \vec{v}_{2n+2}, \vec{v}_{2n+3}) + \ldots +$
  $k_{n+m} L(\vec{v}_{2n+3m-2}, \vec{v}_{2n+3m-1}, \vec{v}_{2n+3m}) \begin{cases} \leq & d \\ = & d \\ \geq & d \end{cases}$

  Where:

  $L(v_a, v_b, v_c)$ equals $\mid (v_a - v_b) \times (v_c - v_b) \mid / \mid v_c - v_b \mid$
  ($L(v_a, v_b, v_c)$ is the distance between v_a and the line defined by v_b and v_c).

- Angular constraint: $\text{Angle}(\vec{v}_p, \vec{v}_1, \vec{v}_2, \vec{v}_3) = a$
  The angle between lines $\vec{v}_1\vec{v}_2$ and $\vec{v}_3\vec{v}_4$ is a.

  To add an angular constraint, add the following equation: $(\vec{v}_{12} \cdot \vec{v}_{34}) - \cos(a) \mid \vec{v}_{12} \mid \cdot \mid \vec{v}_{34} \mid = 0$
  where:

  $\vec{v}_{12}$ equals $(\vec{v}_2 - \vec{v}_1)$ and
  $\vec{v}_{34}$ equals $(\vec{v}_4 - \vec{v}_3)$.

6.2 Solving a system non-linear equations

An iterative algorithm is used to solve for a solution to a system of non-linear equations. In each cycle of the iteration, the following steps are performed:

1. a displacement is found by creating and solving a linear optimization problem,
2. an optimal displacement is then found, and
3. the positions of the vertices is updated.

The iteration continues until either all of the equations are satisfied (to within an error tolerance) or no more progress is being made towards a solution.

6.2.1 Finding the displacement

The goal of each iteration cycle is to reduce the global error in the system of non-linear equations. The first step in this process is to find a displacement in which the global error decreases. This is done by creating

and solving the following linear optimization problem:

$$\begin{array}{lll} \text{Minimize:} & \sum_{i=1}^{n_e} \mid e_i \mid + \sum_{j=1}^{n_i} s_j + v \sum_{k=1}^{n} (w_k * \mid d_k \mid) & \\ \text{Such that:} & h_i(\vec{x}) + \nabla h_i(\vec{x}) \cdot \vec{d} + e_i = 0 & 0 \leq i < n_e \\ & g_j(\vec{x}) + \nabla g_j(\vec{x}) \cdot \vec{d} + s_j \geq 0 & 0 \leq j < n_i \\ & s_j \geq 0 & 0 \leq j < n_i \\ & \mid d_k \mid \leq (b/w_k) & 0 \leq k < n \\ \text{Solve for:} & \vec{d}, \vec{e}, \vec{s} & \end{array}$$

Where:

- n is the number of variables.
- n_e is the number of non-linear equations of the form: $h_i(\vec{x}) = 0$.
- n_i is the number of non-linear equations of the form: $g_j(\vec{x}) \geq 0$.
- $\vec{x}$ is the initial position vector.
- $\vec{d}$ is the displacement vector.
- $\vec{e}$ and $\vec{s}$ are the remaining error variables.
- w_k is the weight for variable k (typically, $1 \leq w_k \leq 10$).
- v is the displacement bias (currently, 10^{-3}).
- b is the maximum step size (currently, 5% the length of the diagonal of the object's bounding box).

This LP is similar to the one developed by Bullard and Biegler [3] to solve systems of non-linear equations[5] This LP is not in standard form, since the absolute value function is used in the objective function. It is, however, easy enough to convert this LP to standard form by:

- replacing e_i with $q_i - r_i$,
- replacing $\mid e_i \mid$ with $q_i + r_i$ and
- adding the equations $q_i \geq 0$ and $r_i \geq 0$.

And using a similar substitution for $\vec{d}$.

Since the displacement is found using a linear approximation of a non-linear system, it may not be optimal for reducing the global error. Indeed, for some systems, the global error will increase if the vertices are moved by the displacement. Viking uses two techniques to mitigate this problem. First, because the linear approximations are more accurate for small displacements, the objective function favors solutions with smaller displacements. Second, Viking searches along the displacement vector to find the distance which produces the smallest global error.

[5]The only significant differences are that, in Bullard and Biegler's LP, $v = 0$ and a different strategy is used to find the optimal displacement.

6.2.2 Finding the "optimal" displacement

A binary search along the vector defined by the displacement is used to find the displacement with the smallest global error. The displacement found by the search is not strictly optimal, since the search is limited to searching along a single vector which may not contain the true optimal displacement. The algorithm used to perform the binary search is given below (where $\mu(x)$ is the global error at x):

Starting with: $l \Leftarrow 0$
$h \Leftarrow 1$

Loop:

$m \Leftarrow (l+h)/2$
if $((\mu(x+h\vec{d}) < \mu(x+l\vec{d}) \wedge (\mu(x+h\vec{d}) < \mu(x+m\vec{d}))$ [1]
    exit loop, $(h\vec{d})$ is the optimal displacement.
else if $(\mu(x+l\vec{d}) < \mu(x+h\vec{d}))$
    $h \Leftarrow m$ [2]
else
    $l \Leftarrow m$ [3]

The benefit of using the binary search is most clearly seen when x is close to a solution for the system of equations. In this situation, the initial displacement may be too large and, without the binary search, the vertices would be displaced beyond the solution. On the next iteration, the displacement may overshoot the solution again and the constraint solver will oscillate around the solution without finding it. Using the binary search described above mitigates this problem: if the displacement steps too far beyond the solution, then the global error will not decrease, the test on line [1] will be false, and the search will continue on either the first half of the displacement vector (line [2]) or the second half (line [3]). Normally only two or three iterations of the loop are required to find an optimal displacement.

6.2.3 Updating the position of the vertices

Once an optimal displacement has been found, updating the positions of the vertices is trivial: simply add the corresponding displacement to each coordinate of each vertex. Determining whether sufficient progress is being made is slightly more complicated. Viking currently bases progress on the slope of the change in global error over the past ten iterations. If this slope is close to zero or positive (which would indicate an increasing global error), the search for a solution aborts.

7 Performance

Two critical aspects of an interactive system are how quickly it can react to the user's actions and how performance degrades as the problem size increases. Viking's performance was tested by creating several faceted spheres, such as the one shown in Figure 15, and modifying the line-drawing by making all of the line-segments obscured by one face visible. Table 3 lists the times required to perform the steps required to find a surface-topology corresponding to the modified line-drawing.

Currently, Viking's algorithm for sketch interpretation is uncomfortably slow, often taking one or two minutes to find a surface-topology. Most of this time, however, seems to be spent allocating memory and page-swapping. Switching to more efficient data-structures could dramatically reduce the memory requirements and improve performance. That and improved workstation performance should allow future versions of Viking to provide the fast response times needed for an interactive system, especially when used with relatively small objects.

8 Future work

Viking was written as a test-bed for developing a user-interface based on sketch interpretation. As such, it is an uneven implementation of a solid modeler. Many of the capabilities found in conventional

# of Vertices	9	16	25	36	49	64	81	100
# of leaves in tree	15	34	56	107	164	233	311	399
Total memory (MB)	0.7	1.3	2.0	3.8	6.3	8.4	11.0	14.6
Check time	0.04	0.04	0.09	0.13	0.19	0.24	0.33	0.41
Load time	0.21	0.60	1.10	2.46	4.19	5.75	7.58	9.98
Filter time	0.05	0.24	0.47	0.67	0.78	0.95	1.03	1.09
Setup time	0.07	0.15	0.30	0.80	1.58	2.22	3.00	4.21
Search time	0.06	0.15	0.23	0.38	0.62	0.90	1.27	1.89
Total time	0.44	1.18	2.19	4.36	7.36	10.06	11.23	17.58

# of leaves	is the total number of leaves in the search tree.
Total memory	is the amount of memory used for Viking during the search.
Check time	time required to check for obvious inconsistencies.
Load time	time required to load the valid labelings for each intersection.
Filter time	time required to perform Waltz-filtering [15].
Setup time	time required to initialize the search tree.
Search time	time required to search for the surface topology.
Total time	total time required to find the surface topology.

All times are in CPU seconds on a Sun SPARC station 1+

Table 3: Time requires to generate a surface topology.

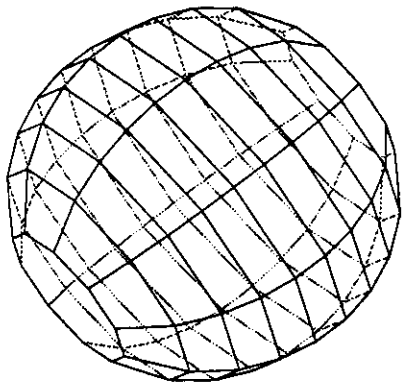

Figure 15: 100 vertex object for testing system performance.

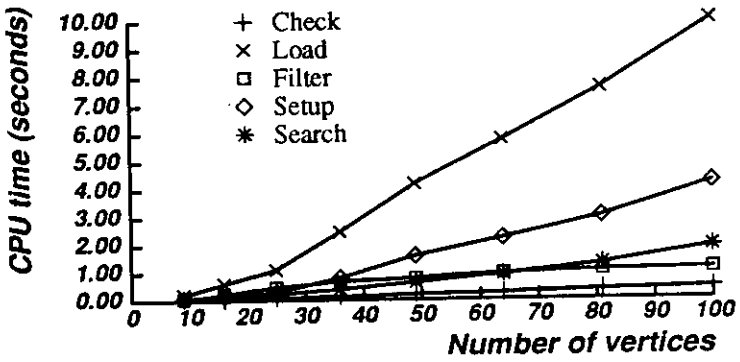

Figure 16: Graph of CPU time vs. number of vertices

solid modelers are not present in Viking. For example, Viking does not support constructive solid geometry using boolean operations. Adding support for conventional solid modeling to Viking will allow for a direct comparison between interactive sketch interpretation and conventional solid modeling techniques.

8.1 Sketch interpretation

Viking uses an intersection library which contains all distinct intersections of four or fewer lines. Currently there is no provision for generating the valid labelings for more complex intersections, although this will change in future versions of Viking. The program used to generate Viking's intersection library is able to find the valid labelings for arbitrarily complex intersections. It is a simple matter to move this algorithm into Viking and use it to find the valid labelings for intersections with more than four adjacent lines.

Another change planned for Viking is to allow objects with non-planar faces. Viking's sketch interpretation algorithm is robust enough to analyze sketches of objects containing non-planar faces with only minor changes. For example, the algorithm used to generate the valid labelings for an intersection assume that surface-fragments are planar in only two places: the expansion of the $\otimes$ predicate and the equations used to

represent the $\Uparrow$ and $\Downarrow$ predicates in Section 4.1.1. Non-planar surface-fragments can be accommodated by simply modifying the $\otimes$ predicate and assuming that any $\Uparrow$ and $\Downarrow$ predicate involving them can be satisfied.

8.2 Constraint satisfaction

Two major changes are planned for Viking's constraint satisfaction user-interface. First, several new types of constraints will be supported. These new constraints will allow the user to control the shape of a group of vertices. For example, the user could place a rectangular constraint on four vertices, and the vertices will be constrained to form a planar rectangle. Second, the way in which dragging by the user is done when the constraints are initially satisfied will be changed to use the constrained dynamics techniques developed by Witkin and Welch [16]. Constrained dynamics is faster and more robust than the non-linear constraint solver currently used.

8.3 Constraint inference

One problem with Viking is that it can be exceedingly tedious to make a sketch and then place all of the appropriate constraints. The Snap-dragging user-interface developed by Bier [1] and modified by Gleicher and Witkin [7] allows the user to place constraints with no extra work as the model is being created. Incorporating Snap-dragging into Viking could simplify the task of creating constrained models.

9 Summary

This paper describes a solid modeler, Viking, whose user-interface is based on interactive sketch interpretation. This interface is designed around a cycle in which the user repeatedly:

- modifies a line-drawing of the current object and
- generates an interpretation of the modified line-drawing. This interpretation replaces the current object.

The user can modify the line-drawing by adding or removing lines, by changing the visibility of a line-segment, and by placing geometric constraints on the object represented by the line-drawing. Viking generates a new interpretation by searching for one which is consistent with the modified line-drawing. Since line-drawings typically have multiple valid interpretations, Viking uses heuristics to generate interpretations in order of decreasing desirability to the user. These heuristics are so effective that the interpretation desired by the user is normally the first interpretation generated by Viking.

The job of finding an interpretation of a modified line-drawing is broken into two parts: finding a surface topology using arc-labeling and finding the vertex positions using constraint satisfaction. Arc-labeling combines an algorithm for finding the valid labelings of arbitrary intersections with heuristics for evaluating the desirability of a labeling to the user. Constraint satisfaction uses a non-linear satisfaction algorithm to solve systems of geometric constraints, where the constraints are either explicitly placed by the user or inferred from the line-drawing. The combination of these two techniques creates a fast and flexible sketch interpretation system.

References

[1] Eric Bier (1989) *Snap-Dragging: Interactive Geometric Design in Two and Three Dimensions* Technical report EDL-89-2, Xerox Palo Alto Research Center, 1989

[2] Alan Borning (1979) *ThingLab - A Constraint-Oriented Simulation Laboratory* PhD thesis, Stanford University, March 1979

[3] L.G. Bullard and L.T. Biegler (1989) *LP Strategies for Constraint Simulation*, AIChE '89 conference Proceedings, AIChE, San Francisco, November 1989.

[4] R. T. Chien and Y.H. Chang (1974) *Recognition of curved objects and object assemblies* Proceedings of the 2^{nd} International Conference on Pattern Recognition, pp. 496-510

[5] M. B. Clowes (1971) *On seeing things*, Artificial Intelligence, vol 2, pp. 79-116

[6] Theresa Farrah and Alan Borning *A User Interface for ThingLab Based on Direct Manipulation of Physical Objects* University of Washington, Technical report number 86-09-02

[7] Michael Gleicher and Andrew Witkin (1991) *Creating and Manipulating Constrained Models* Published as CMU School of Computer Science technical report: CMU-CS-91-125.

[8] James Gosling (1983) *Algebraic Constraints* PhD thesis, Carnegie-Mellon University, May 1983. Published as CMU Computer Science Department tech report CMU-CS-83-132.

[9] D. A. Huffman (1971) *Impossible objects as nonsense sentences*, B. Mettzer, D. Michie (Eds.) Machine Intelligence 6, pp 295-323 (Edinburgh University Press, Edinburgh, 1971)

[10] Takeo Kanade *A Theory of Origami World* Artificial Intelligence, vol 13, pp. 279-311

[11] Greg Nelson (1985) *Juno, a Constraint-Based Graphics System* In B.A. Barsky (Ed.) *SIGGRAPH '85 conference Proceedings*, pp. 235-243. ACM, San Francisco, July 1985.

[12] P. V. Sanker (1977) *A vertex coding scheme for interpreting ambiguous trihedral solids* Computer Graphics and Image Processing, vol 6, pp. 61-89

[13] Kokichi Sugihara (1986) *Machine Interpretation of Line Drawings*, The MIT Press: Cambridge, Massachusetts

[14] Ivan Sutherland (1963) *Sketchpad: A Man Machine Graphical Communications System* PhD Thesis, Massachusetts Institute of Technology, Jan 1963

[15] David Waltz (1972) *Generating semantic descriptions from drawings of scenes with shadows* MAC AI-TR-271, MIT (1972) Also in: P. Winston (Ed.) *The Psychology of Computer Vision* (McGraw Hill, New York 1975)

[16] Andrew Witkin and William Welch (1990) *Fast Animation and Control of Non-Rigid Structures* Computer Graphics, vol 24, 1990