VANTAGE
A Frame-Based Geometric Modeling System
Programmer/User's Manual V2.0

B. Kumar, J.C. Robert, R. Hoffman, K. Ikeuchi, T. Kanade

CMU-RI-TR-91-31

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

December 1991

Table of Contents

List of Figures

List of Tables

Abstract

Geometric modeling systems allow users to create, store, and manipulate models of three-dimensional (3-D) solid objects. These geometric modeling systems have found many applications in CAD/CAM and robotics areas. Graphic display capability which rivals photographic techniques allows realistic visualization of design and simulation. Capabilities to compute spatial and physical properties of objects, such as mass property calculation and static interference check, are used in the design and analysis of mechanical parts and assembly. Output from the geometric modelers can be used for automatic programming of NC machines and robots.

These geometric modeling systems are powerful in many application domains, but have severe limitations to be used for tasks such as model-based computer vision. Among others,

1. There is no explicit symbolic representation of the two-dimensional (2-D) information obtained by the projection of the 3-D model. The output image displayed on the screen is a set of pixel intensity values, with no knowledge of the logical grouping of points, lines and polygons. Also, the relationship between 3-D and 2-D information is not maintained properly.

2. Most of the current 3-D geometric modeling systems are designed with a closed architecture, with a minimum of documentation describing the internal data structures. Moreover, some of the data structures are packed into bit-fields, making understanding and modification difficult.

3. They run as stand-alone interactive systems and cannot easily be interfaced to other programs.

To address these shortcomings, we have developed the VANTAGE geometric modeling system. VANTAGE uses a consistent object space representation in both the 3-D and 2-D domains, which makes it suitable for computer vision and other advanced robotics applications. Its open architecture design allows for easy modification and interface to other software. This paper discusses the design goals and methodology for the VANTAGE geometric modeler.

1. Introduction

Geometric modeling systems allow users to create, store, and manipulate models of three-dimensional (3-D) solid objects. These geometric modeling systems have found many applications in CAD/CAM and robotics areas. Although powerful in many application domains, there are some limitations of these geometric modeling systems, which make them difficult to be used for tasks such as model-based computer vision. Among others,

1. There is no explicit symbolic representation of the two-dimensional (2-D) information obtained by the projection of the 3-D model. The image data is a set of pixel intensity values, with no knowledge of the logical grouping of points, lines and polygons.
2. They are designed with a closed architecture, with a minimum of documentation describing the internal data structures. Worse, some of the data structures are packed into bit-fields, making understanding and modification difficult.
3. They run as stand-alone interactive systems and cannot easily be interfaced to other programs.

To address these shortcomings, we have developed the VANTAGE geometric modeling system. VANTAGE uses a consistent object space representation in both the 3-D and 2-D domains, which makes it suitable for computer vision and other advanced robotics applications . Its open architecture design allows for easy modification and interface to other software. The problems involving model based vision are the main driving force behind this work and VANTAGE has applications in computer vision and advanced robotics research.

The current version of VANTAGE is reasonably debugged and has decent graphic routines and user interface. This manual covers the following areas:

- General concepts and terminology
- Overview of design and implementation
- Primitive solids and coordinate transformations
- Operations on solids
- 3-D boundary representation
- Light-sources, cameras
- Scenes and 3-D properties
- Images and 2-D properties

VANTAGE is currently supported on SUN running Lucid Common Lisp or Allegro Common Lisp. Most of the code is portable to other lisp environments except for the graphic and user-interface routines. VANTAGE can also run under X-Window system. Please direct all enquires to vantage@cs.cmu.edu.

2. Terminology

- Primitives: The basic solids provided by the system. They can be defined by giving required dimensions. The primitives are cube, cone, truncated-cone, cylinder, sphere, 2.5prism, 2.5cone, triangular-prism and right-angle-prism.

- Boolean Operations: The operations allowed on the solids to move, modify or create a resulting solid. The operations defined are union, intersection and difference.

- CSG-Definition : The definition of solids is stored in the form of a tree structure. The leaf nodes are either primitives or nil. A parent node corresponds to a boolean operation applied to its corresponding children nodes. When creating the boundary representation for a particular node, the boundary representation for all the children of the node is also created recursively.

- Boundary representation: A solid is represented as a collection of faces, a face as a collection of edges, and an edge is defined by its two end vertices. The vertices are defined by their x, y and z coordinates. Topological information is included in the form of *winged-edge* representation.

- Winged-edge representation: A way of defining the topological relationship. There are eight slots for each edge which define the two end vertices (P-vertex, N-vertex), four neighboring edges (PCW, PCCW, NCW , NCCW) and two adjacent-faces (P-face, N-face).

- 3D-hierarchical structure: In the boundary representation the cylindrical, conical and spherical faces are approximated by a finite number (user specified) of planar faces. After any sequence of boolean operations the faces derived from a same primitive surface are grouped so that they can be treated as a single entity. Also there is provision to group faces that are tangent to each other across a common edge.

3. VANTAGE - Concepts and Design

3.1. Motivations for Developing VANTAGE

Currently available geometric modeling systems have critical limitations to be used in applications which require flexible, explicit, and user-specified access, attachment, and modification of the information within the systems. For example, developing a model-based vision system based on an object model requires analyzing how the object features, such as faces, edges, *etc.*, will appear as the camera positions vary. However, though the graphic display of the object generates beautiful "images", usually no *explicit* symbolic representations of the two-dimensional information are computed in the projection of a 3-D object model. The graphical output image is a set of pixel intensity or color values, with no knowledge of the logical grouping of points, lines, and polygons, or pointers to the original object features. Humans can interpret the image, but the explicit information that a vision program may require is not available.

Rarely do the built-in functions of a geometric modeling system satisfy all the representational and computational capabilities that a user needs for his own new application. Theoretically, writing or modifying a few modules to access and manipulate the information hidden in the system or adding a few representational capabilities to the existing ones will bring about the required capabilities. In practice, however, such modifications and additions are very difficult and painful, if not impossible. Most systems are designed with a closed architecture, with a minimum of documentation describing the internal data structures. Worse, due to their implementation in such languages as Fortran, many of the data structures are packed into bit-fields, making understanding and modification above a certain level of sophistication impractical.

Recognizing these limitations of currently available geometric modeling systems, we have decided to develop a new flexible geometric modeler, VANTAGE, so that

- Both 3-D and 2-D information of objects can be explicitly represented by symbolic data structures.
- A user can easily modify the system, add new capabilities, and interface his programs to it.

Our primary application of VANTAGE will be in the area of model-based computer vision, but we expect the flexibility and modifiability of VANTAGE will allow it to be used as a tool for many other

advanced robotics applications.

3.2. Open Architecture

Many existing modelers act as a black box for the application programs and discourage sharing data. The approach we take emphasizes direct interaction between the modeling system and the application program. The collection of all data items, such as surface, edge, camera, light-source, *etc.*, describing objects and their relationships form the *geometric database*. The *geometric engine*, consisting of both the system-defined and user-defined functions, can access and manipulate it. This implies that all the system and application programs are at the same level of hierarchy in terms of accessing information, with minimal distinction between them. Figure 3-1 illustrates the open architecture.

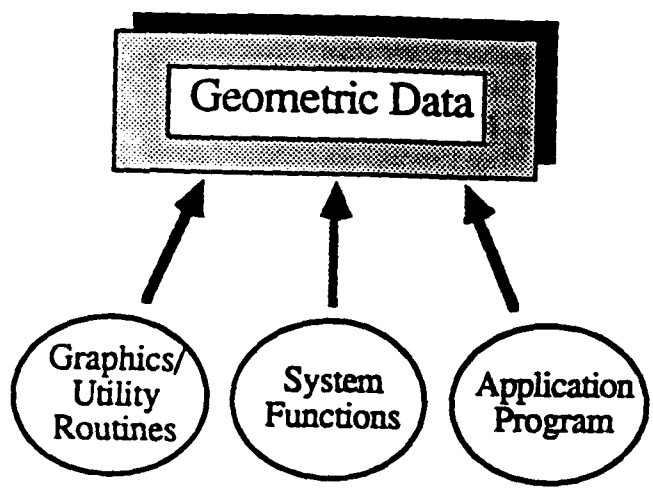


Figure 3-1: Open Architecture of VANTAGE

3.3. Lisp and frame-based Representation

VANTAGE avoids complex and heterogeneous data structures. All data in VANTAGE are represented in a standardized manner by the use of *frames*. Frames are analogous to schema or concepts as defined in other knowledge representation languages. A frame is composed of *slots*, *facets* and *fillers*. For example, a frame structure defining a face may look like:

```
(BOTTOM-FACE
    (is-a
        (value   3-D-face))
    (area
        (value   140)
        (if-added (update-max-area-face)))
    (face-of
```

```
        (value   my-cube))
(edge-list
        (value   edge-a edge-b edge-c edge-d)))
```

In this example, BOTTOM-FACE is the name of the frame. The slots are used to represent various attributes of a frame, such as **is-a**, **area**, **face-of**, and **edge-list**. A slot can have multiple facets such as **value** and **if-added**. *3-D-face, 140, update-max-area-face* are fillers defining contents of different facets.

Frames are like record structures in conventional programming languages, but have much more flexibility and features. Frames, slots, facets, and fillers can be added or erased at any time. Frames also provide a mechanism to automatically select and execute procedures and functions attached to a frame depending on the operation performed on a particular slot. These functions are called *demons*. In the above example, *update-max-area-face* is a demon which is fired automatically to update, if necessary, the variable *maximum-area-face* when a value for the slot **area** is added. The frame structure is omnipresent throughout the system. The flexibility of frames provides an effective means to allow smooth interface to user supplied programs.

VANTAGE is implemented in the COMMON LISP language. LISP combines symbolic processing with features from traditional computing. By writing VANTAGE in LISP, it inherits all the merits of the LISP language such as interaction, incremental building, symbolic representation *etc.* We believe that VANTAGE will have successful applications in the A.I. world.

3.4. Solid and Boundary Representation

We have selected the Constructive Solid Geometry approach for representing the shapes of objects in VANTAGE . VANTAGE provides basic solid primitives like cube, cylinder, *etc.* The user creates new solids by making boolean operations (union, difference and intersection) on these primitives.

A 3-D boundary representation of each object is maintained within the system. This contains lists of faces, edges and vertices. Vertices contain their respective coordinate values, and edges join these vertices. The faces are planar polyhedra and represented by a collection of connected edges.

The neighborhood information or *topology* relates the edges, faces and vertices of the solid. This information in VANTAGE is stored in the form of *winged-edge* representation. An edge has two end vertices, four neighboring edges and two faces defining a strict relationship.

3.5. Relations Between 2-D and 3-D

A model-based vision system attempts to recognize objects in images by matching features in the image with those expected from the model of the scene. Many current geometric modelers are good at synthesizing images of a scene with a given viewing position and lighting condition. It is, however, very difficult to extract symbolic representations of expected appearances of the object so that they can be used in designing recognition strategies. One of the VANTAGE design goals is to provide a capability to explicitly represent the relationships between 3-D information (such as shape, surface, and lighting) in the scene and 2-D information (such as visibility, shadow, and projected shape) in the image.

For a particular viewing condition, the 3-D faces are projected on the viewing plane and the visible portion of the projections result in 2-D face-regions. These 2-D face-regions are a collection of 2-D arcs and the 2-D arcs connect 2-D joints. The hierarchy of 2-D face-regions, 2-D arcs and 2-D joints is the same as faces, edges and vertices at the 3-D level.

Image formation in general is dictated by the physical properties of the considered surfaces. A 3-D face contains some slots for representing various properties like color, surface-roughness, shadows, etc. They are classified according to the cause. For example, a particular 3-D face can have shadows due to different light sources such as *light-source-a* and *light-source-b*. All these shadow conditions interact to produce the final image. The 3-D face properties are projected on the 2-D regions, which are divided into *property-regions*.

3-D level	2-D level
3-D face	2-D region
3-D edge	2-D arc
3-D vertex	2-D joint
3-D property	2-D property-region

Table 3-1: Relation between 3-D and 2-D level

Table 3-1 explains the relation between the two levels. The information at the 2-D and 3-D levels have a correspondence and can be referenced back and forth. In addition, VANTAGE maintains topological information not only at the 3-D level but also at the 2-D level.

3.6. Organization

Figure 3-2 shows the overall organization of VANTAGE.

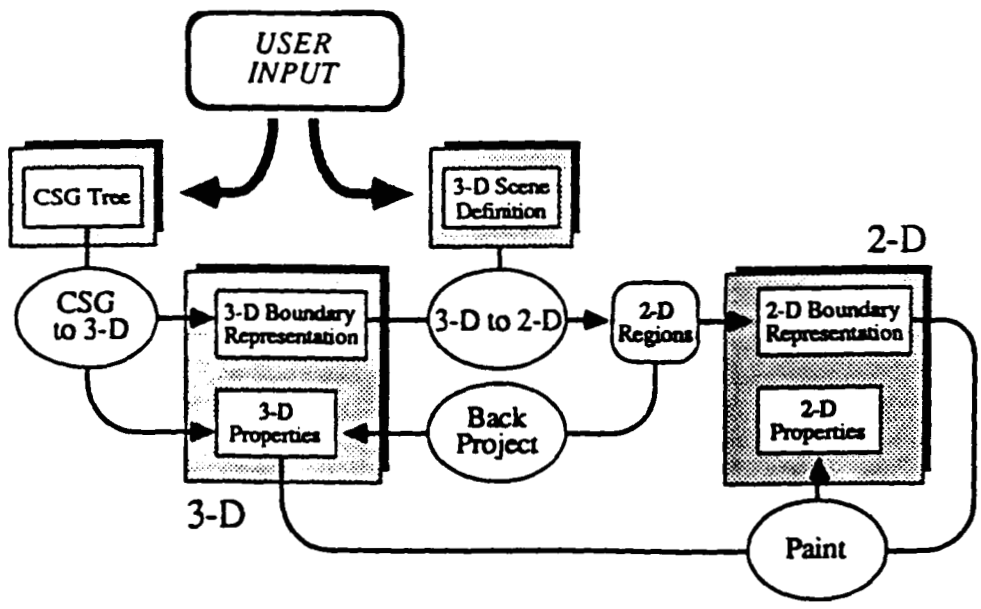


Figure 3-2: Organization of VANTAGE

1. The user creates a solid by applying operations on primitives solids. The definition of the solid is stored in a Constructive Solid Geometry (CSG) Tree.
2. The 3-D boundary representation of the solid is generated from the CSG tree in the 3D geometric database. The face properties (*e.g.* color, shadow, and visibility) are also maintained in the 3D geometric database.
3. The user defines a 3-D Scene that contains a collection of solids, environmental conditions (*e.g.* lighting conditions), and a viewing condition.
4. The 3-D scene is projected to generate a 2-D Image, for which VANTAGE creates a complete explicit representation (2-D boundary representation and 2-D Properties), which contains geometric and topological information for all visible regions, as well as back-pointers to the 3-D boundary representation in the 2D geometric database.

The subsequent sections detail the different parts of the system.

3.6.1. Solid Definition

In Constructive Solid Geometry, an object is generated by applying successive operations (union, intersection, difference, move, mirror) on a set of primitive solids (cube, cylinder, cone, sphere). A CSG tree represents internally this CSG definition. A leaf node of a CSG tree defines a primitive solid. An intermediate node specifies an operation to be performed on its descendants, and corresponds to the solid resulting from the operation. Figure 3-3 shows an example of a CSG tree.

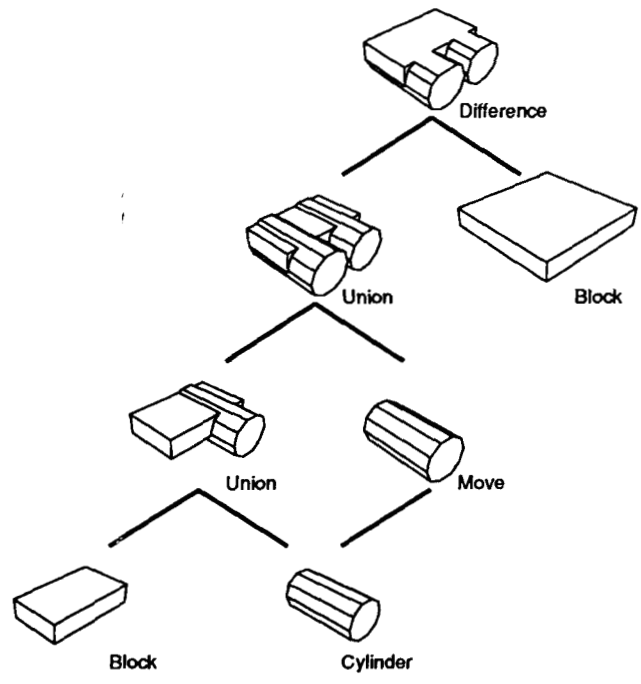


Figure 3-3: CSG-Tree

3.6.2. 3-D Boundary Representation

A 3-D boundary representation of each object is maintained within the system. It consists of a four level hierarchy of frames: body, face, edge, and vertex. A body is made of faces, a face is defined by its edges, and an edge has two end vertices. Each element contains some geometric properties (coordinates of the vertices, equations of the faces, position of the body in space). In addition, VANTAGE maintains a complete representation of topological relationships in the form of a winged-edge representation that lists the end vertices, neighboring edges and faces of every edge (see figure 3-4).

Although VANTAGE includes non-polyhedral primitives such as cylinders and cones, all non-planar surfaces (cylindrical, conical or spherical surfaces) are approximated by a finite number of planar faces. In the same way, all curves are represented by a collection of linear edges. The number of planar faces used to represent a non-planar surface is entered by the user.

VANTAGE stores the exact geometric definition of each surface and curve as a separate frame. VANTAGE also maintains a pointer from every planar face that approximates a non-planar surface to the corresponding surface, and similarly from the approximating edges to the corresponding curves.

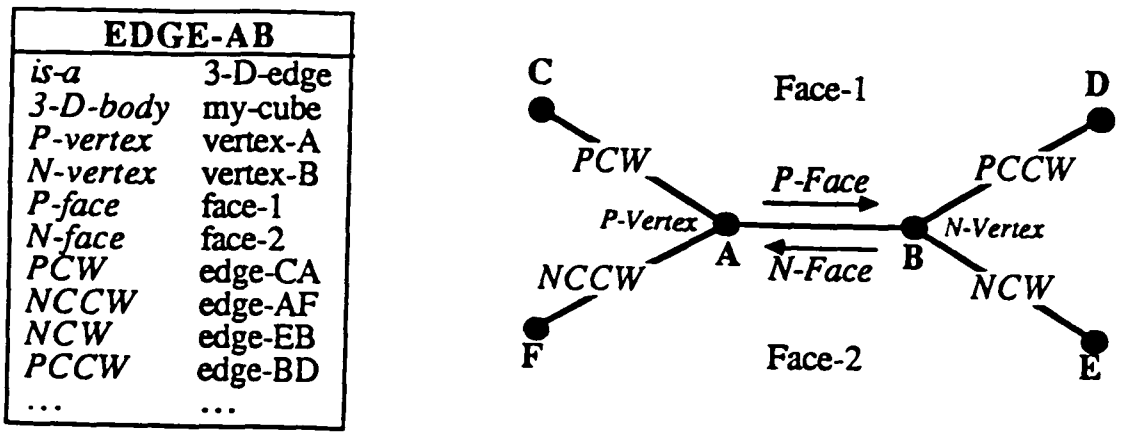


Figure 3-4: Winged-edge representation and associated frame

The surface/curve frames are used for grouping faces/edges that approximate the same surface/curve. Also, any operation that requires the exact geometric definition of surfaces and curves (*e.g.* generation of parametric equations) can be performed using the surface and curve frames.

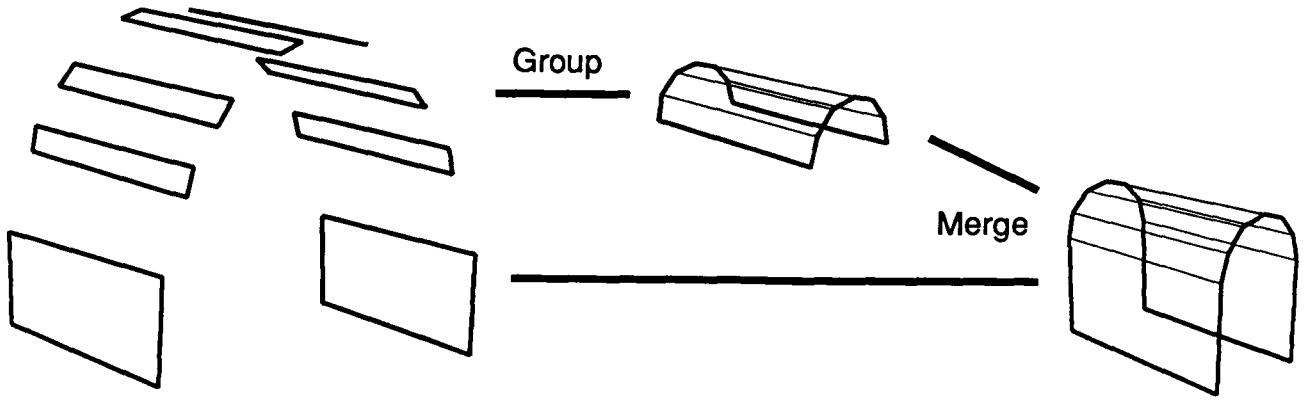


Figure 3-5: Grouping and Merging Operations

VANTAGE builds two more levels of representation based on surface properties. Figure 3-5 illustrates the grouping and merging operations for this purpose. First, using references to the surface frames, VANTAGE can group a set of adjacent planar faces that approximate the same curved surface into a *curved* face frame. Similarly, connected linear edges that approximate the same curve are also grouped into a *curved* edge frame. Second, faces that are tangent across an edge, that is, C1 continuous, are also merged into one. Since detecting the C1 continuity is sometimes ambiguous due to the finite precision of floating point calculation, VANTAGE provides an interactive graphic interface that allows the user to select any pair of adjacent faces he desires to merge.

Importantly, the topological relationships of grouped and merged surfaces are also maintained in a winged-edge representation. This feature is very useful for computer vision applications, where a continuous surface must often be treated as a single surface.

3.6.3. 3-D Face Properties

Property descriptions of the faces of solids can be attached to the 3-D boundary representation. There are two types of face properties:

- Physical properties inherent to the solid itself (*e.g.* color, texture).
- Properties that result from the environment of the solid (*e.g.* cast shadow for a given light-source). These properties are computed at the time of projection.

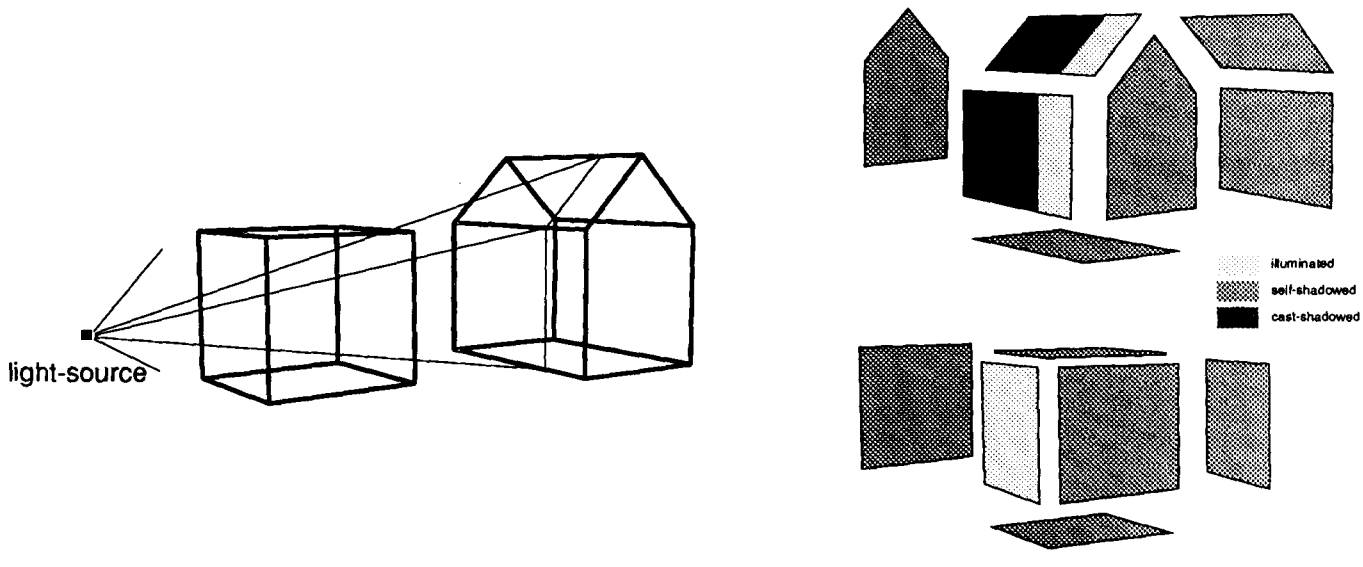


Figure 3-6: 3-D Face Properties for a Light-source

Figure 3-6 illustrates the 3-D property representations in the case of color and shadow. These divisions due to different properties are stored in the property frame of the 3-D faces. They are also classified according to the cause. For example, a particular 3-D face can have shadows due to different light-sources such as *light-source-a* and *light-source-b*.

3.6.4. 3-D Scene

A 3-D scene is a portion of the world for which we can create an image. It is composed of:

- A collection of solids
- A selection of physical properties of the solids (*e.g.* color).
- A set of environmental conditions, that can include:
  - A set of light-sources (one or several).

• A sensor, which is used to generate a 2-D image of the scene (*e.g.* a camera).

Each scene condition (*e.g.* a light-source) is defined as a separate frame containing all necessary information: location, color of light, point/extended light-source, sensor characteristics, etc.

3.6.5. 2-D Image

The 2-D representation of a 3-D scene is an explicit symbolic representation of the image that is obtained by projecting the scene using the specified sensor (see Figure 3-7). VANTAGE provides a capability to explicitly represent the relationships between 3-D information (such as shape, surface, and lighting) in the scene and 2-D information (such as visibility, shadow, and projected shape) in the image. For a particular viewing condition, the 3-D faces are projected on the viewing plane and the visible portion of the projections result in 2-D face-regions. These 2-D face-regions are a collection of 2-D arcs and the 2-D arcs connect 2-D joints. The hierarchy of 2-D face-regions, 2-D arcs and 2-D joints is the same as faces, edges and vertices at the 3-D level.

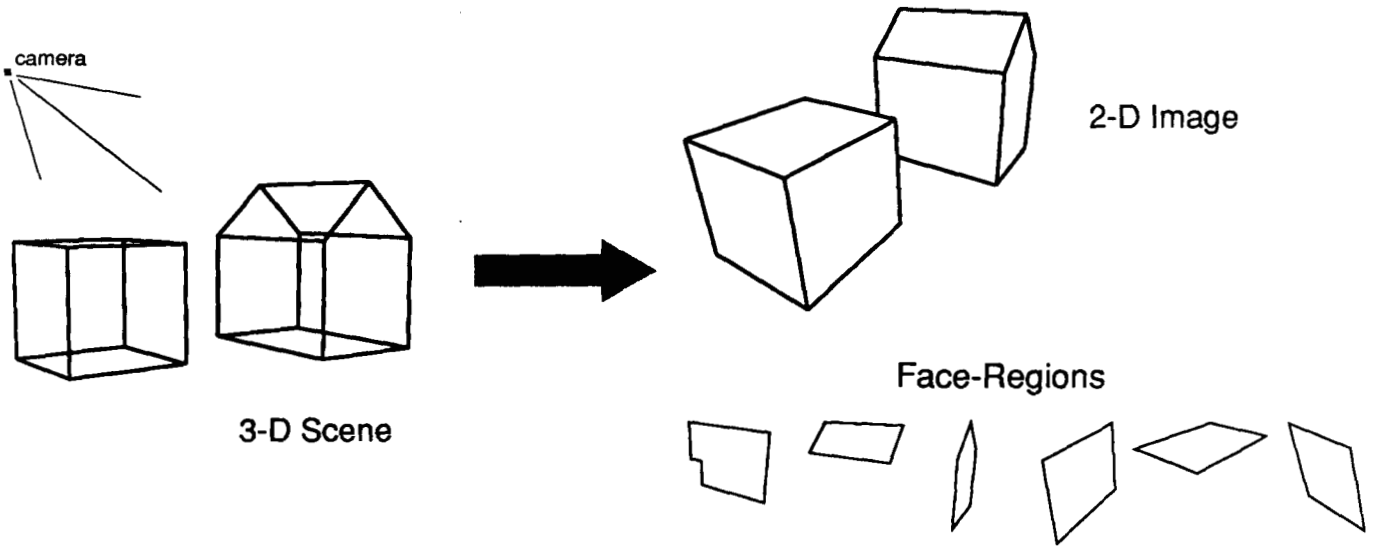


Figure 3-7: Projection of a 3-D scene, and regions of the resulting image

The algorithm to project a scene from 3-D to 2-D in addition generates all topological relationships among the face-regions (projections of the 3-D faces) of the image something similar to that of the 3-D level.

3.6.6. 2-D Property Regions

Different lighting conditions interact to produce the final image. To compute whether a face is illuminated by a light-source or shadowed, we take advantage of the correspondence between a light-source and a sensor, as shown in table 3-2. When projecting a scene using a light-source as viewpoint, the 2-D regions obtained represent the illuminated/cast-shadowed parts of the scene. These 2-D regions are back-projected on the 3-D faces of the solids, and stored as the 3-D face properties associated with the light-source. These 3-D face properties are projected on the 2-D face-regions , which are divided into *property-regions*.

Shading	Visibility
illuminated	visible
cast-shadowed	occluded
self-shadowed	back-face

Table 3-2: Light-source/Sensor

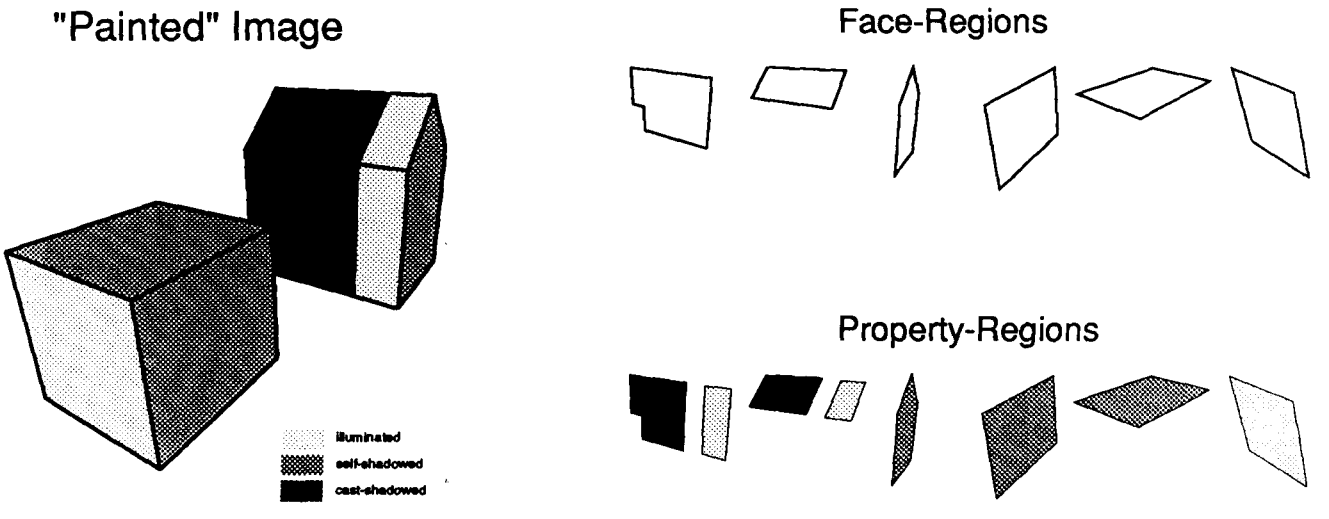


Figure 3-8: Property regions associated to a light-source

Figure 3-8 explains the *property-regions*, which are 2-D face properties of the 2-D face regions.

4. Invoking the system

The Lisp system should be invoked first. VANTAGE currently runs under LUCID and ALLEGRO common lisp environments. Then load the file "/usr/vantage/vantage-init.lisp". This will initialize the VANTAGE system.

```
> (load "/usr/vantage/vantage-init.lisp")
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;               VANTAGE SOLID MODELING SYSTEM  V1.0
;
;               VISION AND AUTONOMOUS SYSTEMS CENTER
;                      THE ROBOTICS INSTITUTE
;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;
;;;;;;; Loading Subsystem - Framekit
;;;;;;;                  - Solids
;;;;;;;                  - Windows
;;;;;;;                  - Utilities
;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;       Here you go.............
;       Send Comments and Bug reports to vantage@cs.cmu.edu
;
>
```

At the end it will open up a new graphic screen called "LISP-Screen". Inside there will be a default window named "vantage window". By clicking the left mouse button inside the window, a pop-up-menu interface can be invoked. Users interested in creating other widows on the Lisp screen are encouraged to go through the Chapter 12 of the "SUN Common Lisp User's Manual" titled "WINDOW TOOL KIT".

5. CSG Definition of a Solid

The creation of a CSG-node is performed by the macro **csgnode**. This macro allows the creation of primitive solids or the creation of solids by applying boolean operations on existing solids.

The following macro creates a solid:

(**csgnode** *solid-name type parameters*)

where *type* can be either a *primitive-type* (e.g. **cube, cylinder**, *etc.*), or an *operation* (e.g. **union, move**, *etc.*). The **csgnode** command can also take optional arguments that are defined below.

5.1. Primitives

They are defined by the macro:

(**CSGNODE** *solid-name primitive-type parameters* &key (*trans* *identity*)) *macro*

or the function:

(**CSGNODE*** *solid-name primitive-type parameters* &key (*trans* *identity*)) *function*

CSGNODE* is like CSGNODE, except that it evaluates its arguments.

primitive-type is one of the following types:

cube cylinder cone truncated-cone sphere iso-prism
right-angle-prism 2.5-prism 2.5-cone
or their abbreviated forms:
cu cub cy cyl co con tru sp sph iso rt 2.5p 2.5c

parameters is a list of numbers and depends on the primitive type.

- cube (*x-length y-length z-length*)
- cylinder (*radius height number-of-app-faces*)
- cone (*radius height number-of-app-faces*)
- truncated-cone (*bottom-radius top-radius height number-of-app-faces*)
- sphere (*radius approximation-number*)

- iso-prism (*base side height*)
- right-angle-prism (*side1 side2 height*)
- 2.5-prism (*height* (x_1 y_1) (x_2 y_2) (..) ...)
- 2.5-cone (*height* (x_{apex} y_{apex}) (*x1 y1*) (*x2 y2*) (..) ...)

trans is an optional parameter that defaults to the identity transformation. It specifies the rigid-motion attached to the node. It can be one of the following:

- *name* : name of an already defined rigid-motion.
- list of six float-numbers (*x y z roll pitch yaw*): the system will generate a motion matrix and will give a new name for it. The angles *roll*, *pitch*, and *yaw* can be entered in degrees or radians depending on the current value of the variable ***angle-mode*** (see page 29).
- list (*name x y z roll pitch yaw*): same as above but the given name is assigned.

See Appendix A for an example of each primitive.

5.2. Rigid-Motion

5.2.1. Moving a solid

(**CSGNODE** *solid-name* **move** *solid* &key (*trans* *identity*) (*fast* NIL)) *macro*

Defines a new solid obtained by applying to an existing solid the transformation specified by *trans*.

trans is as defined in the previous section.

fast specifies whether the boundary representation of the child nodes should be destructively affected or not, when generating the boundary representation for the specified node. *fast* takes one of the following values:

- **NIL**: the boundary representation of the child nodes will be copied and not destructed when generating the boundary representation of the specified node.
- **T**: the boundary representation of the child nodes will be destructed when generating the boundary representation of the specified node. They will not be copied, therefore saving computation time.
- **all**: all nodes" below" the specified node in the CSG-tree will have their *fast* flag set to **T**. Only the boundary representation of the specified node will remain.

(**CSGNODE*** *solid-name* **move** *solid* &key (*trans* *identity*) (*fast* NIL) *function*

CSGNODE* is like CSGNODE, except that it evaluates its arguments.

(**MOVE-CSG-NODE** *node-name trans*) *macro*

Moves an existing solid by applying to it the rigid-motion transformation specified by *trans*. The user is asked to confirm the move command when a boundary-representation exists for the node or when the node has parent nodes. If the user chooses to move the node anyway, the boundary-representations of the node and its possible parents are deleted. Note that from now on, the parents of the moved node will take into account the new location of the node.

(MOVE-CSG-NODE* *node-name trans*) *function*

MOVE-CSG-NODE* is like MOVE-CSG-NODE, except that it evaluates its arguments.

5.2.2. Defining a transformation

A rigid-motion can be defined at the time it is used, as explained above, or using one of the following functions:

(MK-MOTION-MATRIX *(name x y z roll pitch yaw)*) *function*

x, *y*, *z* define the position of the new origin, and *roll*, *pitch*, *yaw* define the rotations to perform about the initial z, y and x axis. If *name* is absent a system-generated name will be assigned.

(MK-ROTATION &key (*name* nil) (*rpy* '(0 0 0)) (*axis-angle* nil) (*center* '(0 0 0))) *macro*

Creates a rotation transformation, defined by the center of rotation, and either the roll, pitch and yaw coefficients, or the rotation axis vector plus the rotation angle. If *name* is absent a system-generated name will be assigned.

(MK-ROTATION* &key (*name* nil) (*rpy* '(0 0 0)) (*axis-angle* nil) (*center* '(0 0 0))) *function*

MK-ROTATION* is like MK-ROTATION, except that it evaluates its arguments.

(MK-TRANSLATION &key (*name* nil) (*xyz* '(0 0 0))) *macro*

Creates a translation transformation, defined by the translation vector. If *name* is absent a system-generated name will be assigned.

(MK-TRANSLATION* &key (*name* nil) (*xyz* '(0 0 0))) *function*

MK-TRANSLATION* is like MK-TRANSLATION, except that it evaluates its arguments.

(MK-COMBINED-TRANSFORMATION &key (*name* NIL) (*trans-list* nil)) *macro*

Creates a transformation resulting from several successive transformations. *trans-list* is the list of transformations to combine. The matrix of the new transformation is the product, from left to right, of the matrices of the transformations of *trans-list.* If *name* is absent, a system-generated name will be assigned.

(MK-COMBINED-TRANSFORMATION* &key (*name* NIL) (*trans-list* nil)) *function*

MK-COMBINED-TRANSFORMATION* is like MK-COMBINED-TRANSFORMATION, except that it evaluates its arguments.

5.3. Boolean Operations

Complex solids are created by applying boolean operations on other solids.

(CSGNODE *solid-name boolean-operation solids* &key (*trans* *identity*) (*fast* NIL)) *macro*

A new solid is generated by applying the specified operation to the specified solid(s), and then by transforming the resulting solid by the specified rigid-motion.

boolean operation is one of the following operations:

union difference intersection inverse

or their abbreviated form:

un uni di dif int inv

solids is a list of two solids, except for the **inverse** operation, in which case it is just one solid.

trans is specified as explained in the previous section.

fast is explained in the previous section.

(CSGNODE* *solid-name boolean-operation solids* &key (*trans* *identity*) (*fast* NIL)) *function*

CSGNODE* is like CSGNODE, except that it evaluates its arguments.

Example:

The following commands create the nodes that appear in the CSG-tree of Figure 3-3.

```
> (csgnode b1 cu (500 300 111.5) :trans (0 -68.1 49.25 0 0 0))
B1
> (csgnode b2 cyl (120 450 10) :trans (-130 6.9 6.5 0 0 -90))
B2
> (csgnode b3 mov b2 :trans (260 0 0 0 0 0))
B3
> (csgnode b4 cu (3000 3000 100) :trans (0 0 155 0 0 0))
B4
> (csgnode b5 uni (b1 b2))
B5
> (csgnode b6 uni (b5 b3))
B6
> (csgnode body1 dif (b6 b4) :fast all)
BODY1
>
```

5.4. Mirror Operation

This operation creates the symmetric solid of a specified solid relatively to a specified plane.

(**CSGNODE** *solid-name* **mirror** *parameters* &key (*trans* *identity*) (*fast* NIL)) *macro*

parameters is a list (*solid normal-x normal-y normal-z distance*) specifying the solid and the mirror-plane. The plane is defined by the x,y,z coordinates of its normal vector, and by its orthogonal distance to the origin.

(**CSGNODE*** *solid-name* **mirror** *parameters* &key (*trans* *identity*) (*fast* NIL)) *function*

CSGNODE* is like CSGNODE, except that it evaluates its arguments.

5.5. Redefining and Deleting solids

The definition of a node can be changed or deleted. If the affected node has ancestor nodes, then they are all affected as well. When defining a solid using **csgnode**, if the specified name is already used, then VANTAGE asks if it should use another name or replace the existing solid by the new one.

(**DELETE-CSG-NODE** *node-name*) *macro*

Deletes the node *node-name* and its boundary-representation (if it exists). Also deletes the parent csg-nodes of the node (if any), after confirmation from the user.

(DELETE-CSG-NODE* *node-name*) *function*

DELETE-CSG-NODE* is like DELETE-CSG-NODE, except that it evaluates its argument.

5.6. Information on the CSG-Tree

The CSG-Tree specifies how the solids are created, and stores all the node operations in a tree structure. Each node will correspond to a 3D-solid. The leaf nodes are primitive solids. The other nodes are obtained by applying an operation on its child nodes.

(CSG-TREE) *function*

Prints out information on the existing CSG-nodes.

(DESCRIBE-CSG-NODE *node-name*) *macro*

Prints out all the operations involved in the creation of the solid corresponding to *node-name*.

(DESCRIBE-CSG-NODE* *node-name*) *function*

DESCRIBE-CSG-NODE* is like DESCRIBE-CSG-NODE, except that it evaluates its arguments.

(DESCRIBE-CSG-NODES) *function*

Calls the function **describe-csg-node** for all the nodes.

6. Boundary Representation

This chapter describes the functions that generate the boundary-representation of a solid.

(BOUN-REP *node-name* **)** *macro*

Creates a complete 3d boundary-representation for the solid defined by the node. This representation consists of frames that represent the vertices, the edges, the faces, and the body (the name of the body-frame is *node-namez* (with suffix 'z')). If a boundary-representation already exists for the node, nothing is done.

A boundary-representation is generated for all nodes starting with the leaf-nodes (primitive solids) and going up the csg-tree until the specified node. All intermediate nodes that do not have a boundary-representation yet get one in the process, except those whose parent-node has the fast flag on, which get only a temporary boundary-representation that is destructively modified in the process and deleted. If a node already has a boundary representation, the existing representation is used for that node and VANTAGE does not generate a boundary-representation for the node and its child-nodes.

(BOUN-REP* *node-name* **)** *function*

BOUN-REP* is like BOUN-REP, except that it evaluates its arguments.

(3D-STRUCTURE *node-name* **)** *macro*

It has the same effect as *boun-rep*, but in addition the solid gets a *3D-Hierarchical structure* (grouping of faces approximating a same primitive surface...). If a boundary-representation already exists for the node, only the grouping of faces and edges is performed. If the grouping operations have also already been done, the function does not do anything.

(3D-STRUCTURE* *node-name* **)** *function*

3D-STRUCTURE* is like 3D-STRUCTURE, except that it evaluates its arguments.

(3D *node-name* **)** *macro*

Merges connected faces that are specified by the user. The new faces that are created are at the

top-level in the hierarchy of faces (see Figure 3-5). If a boundary-representation has not been generated yet, **boun-rep** is first called.

This main application of this function is to merge connected faces that have a continuous normal across the connecting edge. Since VANTAGE approximates all the higher order surfaces by planar polyhedra it is impossible to automatically detect those edges across which the *merge* should take place. So it requires interaction from the user through *mouse* input.

For merging some faces, some small faces may have to be created at the boundary of a surface, due to the approximation of the surface. Such configurations are first detected, and the user is asked to confirm any modification. Then the user can enter the faces he wants to merge. Any selected face is then considered as an approximated face and grouped with its neighbors to create the parent faces.

(3D* *node-name* **)** *function*

3D* is like 3D, except that it evaluates its arguments.

The boundary-representation of a node can be deleted using:

(DELETE-BOUN-REP *node-name* **)** *macro*

Deletes the boundary-representation of the node *node-name* (if it exists), with all its vertices, edges, faces.

(DELETE-BOUN-REP* *node-name* **)** *function*

DELETE-BOUN-REP* is like DELETE-BOUN-REP, except that it evaluates its argument.

7. Sensors

The definition of sensor applies to cameras and light-sources. In can also include combinations of cameras and light-sources (sensor-components) using AND and OR operations.

(CAMERA *name location* **key** (*target* '(0 0 0)) (*focal* **nil**) (*limit-angle* **nil**)) *macro*

Creates a camera that is positioned at *location* (= (*x y z*)) and that points toward *target*, with the specified focal-length.

limit-angle is the maximum angle (in degrees) between the normal of a face and the viewing direction, for which the face is visible. The default NIL value for limit-angle corresponds to a limit-angle of 90 degrees.

(CAMERA* *name location* **key** (*target* '(0 0 0)) (*focal* **nil**) (*limit-angle* **NIL**)) *function*

CAMERA* is like CAMERA, except that it evaluates its arguments.

(LIGHT-SOURCE *name location* **key** (*target* '(0 0 0)) (*focal* **nil**) (*limit-angle* **NIL**)) *macro*

Creates a light-source that is positioned at *location* (= (*x y z*)) and that points toward *target*, with the specified focal-length.

limit-angle is the maximum angle (in degrees) between the normal of a face and the lighting direction, for which the face is lit. The default NIL value for limit-angle corresponds to a limit-angle of 90 degrees.

(LIGHT-SOURCE* *name location* **key** (*target* '(0 0 0)) (focal **nil**) (*limit-angle* **NIL**)) *function*

LIGHT-SOURCE* is like LIGHT-SOURCE, except that it evaluates its arguments.

(MAKE-SENSOR-COMPONENT *name type params* **key** (focal *NIL*) (*limit-angle* **NIL**)) *function*

Creates a camera or a light-source with the given parameters.

type is either **camera** or **light**.

parameters is a list of six numbers describing x, y, z, roll, pitch and yaw. The camera points towards the negative z-axis given by the camera-coordinate system defined by the parameters.

focal specifies the focal distance of the perspective projection, or, when equal to NIL, characterizes an orthographic projection.

limit-angle is the maximum angle (in degrees) between the normal of a face and the projection direction, for which the face is lit. The default NIL value for limit-angle corresponds to a limit-angle of 90 degrees.

(ROTATE-CAMERA-AROUND-AXIS *camera angle* **)** *macro*

Rotates a camera around its viewing direction. The angle unit is given by ***angle-mode***.

(ROTATE-CAMERA-AROUND-AXIS* *camera angle* **)** *function*

ROTATE-CAMERA-AROUND-AXIS* is like ROTATE-CAMERA-AROUND-AXIS, except that it evaluates its arguments.

8. Scene and 3-D Properties

8.1. Scene

The following functions define a 3-D scene. The environmental properties applied to the scene (lighting conditions) are added to the definition of the scene at the time of calculation of the 3-D property regions of the scene for given light-sources (see section 8.2 and the IMAGE function, page 26)

(SCENE *name csg-node-list***)** *macro*

Defines a 3d-scene by a list of csg-nodes. A boundary-representation of all the bodies of the scene should exist before creating the scene.

(SCENE* *name 3d-body-list***)** *function*

SCENE* is like SCENE, except that it evaluates its arguments.

8.2. 3-D Properties

(PROJECT-AND-BACK-PROJECT *scene sensor* **optional** (*merge-shadows* **T**) **)** *macro*

Generates the 3-D properties of the specified *scene* for a given *sensor* (camera or light-source). A process of projection and back-projection is performed, as explained in paragraph 3.6.6, page 12. When the sensor is a camera, the regions generated on the 3-D faces of the scene are the visible, occluded, or back-oriented areas of the scene for the given camera. For a light-source, the illuminated, cast-shadowed and self-shadowed areas of the scene are obtained (see table 3-2). The names of the properties, which are also the names of the slots of the property-list frames of the faces (see the definition of a property-list frame, page 60), are built as in the following example: if the name of the sensor (camera or light) is S1, the properties will be called **visible-S1, occluded-S1, back-S1**. The property frames (see page 61) are automatically created or updated.
merge-shadows specifies whether the cast-shadowed regions corresponding to different occluding faces should be merged or not before back-projection to the faces of the scene (MERGE-LIGHT-PROPERTIES does the same merging operations, but after back-projecting to the scene).

(PROJECT-AND-BACK-PROJECT* *scene sensor*) *function*

PROJECT-AND-BACK-PROJECT* is like PROJECT-AND-BACK-PROJECT, except that it evaluates its arguments.

(MERGE-LIGHT-PROPERTIES *scene light-source*) *macro*

The property regions, obtained in a scene for a light-source using the previous function, are computed by a face-to-face technique, and therefore the occluded areas are split into regions characterized by the face that occludes them (the occluding face considered is the closest one to the sensor).

PROJECT-AND-BACK-PROJECT (see above) and IMAGE (see page 26) allow the user either to merge these split regions for each face before back-projecting them, or to back-project the split regions directly. In the latter case, the user can perform the merging operations later, using MERGE-LIGHT-PROPERTIES. This function makes the union, on every face of the scene, of the split occluded regions for the light-source, in order to get the full consolidated occluded (cast-shadowed) area. The old split regions are saved as a new property under a new name (for a light-source L1, the name is **split-occluded-L1**), and the new merged property-regions take their previous name (**occluded-L1**).

An example is showed in page 40.

(MERGE-LIGHT-PROPERTIES* *scene light-source*) *function*

MERGE-LIGHT-PROPERTIES* is like MERGE-LIGHT-PROPERTIES, except that it evaluates its arguments.

9. Image and 2-D Properties

9.1. Image

Given a 3-D scene and a sensor (camera), a 2-D image can be generated. The 2-D image consists of 2-D regions, arcs and joints. See paragraph 3.6.5, page 11, for a definition of a 2-D image, and page 62 for a description of a 2-D image frame.

(IMAGE *scene camera* **key** (*lights* **NIL**) (*image-name* **NIL**) (*merge-shadows* **T**)) *macro*

Generates a 2-D image for the given *scene*, using the given *camera*. If no *image-name* is given for the image, a name is automatically generated (*e.g.* image-1209). The complete 2-D representation of the image is computed, including regions, arcs, joints, winged-edge relations and back pointers to 3-D elements.

The generation of 3-D properties (back-face, shadow, illuminated) for the specified *lights* (if not NIL) is also performed (as with the PROJECT-AND-BACK-PROJECT function, page 24). *merge-shadows* specifies whether the cast-shadowed regions corresponding to different occluding faces should be merged or not before back-projection to the faces of the scene.

(IMAGE* *scene camera* **key** (*lights* **NIL**) (*image-name* 'image)) *function*

IMAGE* is like IMAGE, except that it evaluates its arguments.

An image can be deleted using:

(DELETE-IMAGE *image-name*) *macro*

Deletes the image *image-name*, with all its joints, arcs, regions.

(DELETE-IMAGE* *image-name*) *function*

DELETE-IMAGE* is like DELETE-IMAGE, except that it evaluates its argument.

9.2. 2-D Properties

(PAINT-PROPERTY-ON-IMAGE *image-name property-name*) *macro*

Projects the 3-D areas corresponding to the property *property-name* onto the 2-D regions of the image *image-name*. The property regions are first transformed using the camera that generated the image, then clipped by the regions of the image. The 2-D properties are stored in the property-list frames of the regions with the slot name *image-name* (see page 60: the format of a property-list frame is identical in 2-D and 3-D). The property frames (see page 61) are automatically updated.

(PAINT-PROPERTY-ON-IMAGE* *image-name property-name*) *function*

PAINT-PROPERTY-ON-IMAGE* is like PAINT-PROPERTY-ON-IMAGE, except that it evaluates its arguments.

10. Miscellaneous Functions and Variables

10.1. Functions dealing with Boundary representation

(PREVIOUS-EDGE *edge face* **)** *function*

Returns the edge that comes before *edge* on *face*.

(NEXT-EDGE *edge face* **)** *function*

Returns the edge that comes after *edge* on *face*.

(GET-VERTEX-LIST *face* **)** *function*

Returns the list of vertices of *face*. (The vertices are not ordered).

(GET-ORDERED-VERTICES *face* **)** *function*

Returns the ordered list of vertices of the outer boundary of *face*.

(GET-ALL-ORDERED-VERTICES *face* **)** *function*

Returns a list that contains the ordered lists of vertices of the boundaries of *face* (outer boundary and hole boundaries).

(NEIGHBOR-FACES *face* **)** *function*

Returns the list of faces that have at least in edge in common with *face*.

(FACEL-EDGEL-OF-VERTEX *vertex* **)** *function*

Returns a list that contains the list of edges that have *vertex* as an end and the list of faces that have *vertex* as a vertex.

(EDGE-LIST-OF-VERTEX *vertex* **)** *function*

Returns the list of edges that have *vertex* as an end.

10.2. Mathematical functions

(DEG) *function*

Sets the ***angle-mode*** variable to **deg**.

(RAD) *function*

Sets the ***angle-mode*** variable to **rad**.

ANGLE-MODE *variable*

Determines the unit (**deg** or **rad**) for the angles.

(SAVE-ANGLE-MODE) *function*

Saves the current ***angle-node***. To be used in conjunction with the **restore-angle-mode** function.

(RESTORE-ANGLE-MODE) *function*

Sets the ***angle-mode*** to the value it had when calling **save-angle-mode**.

(DEG-TO-RAD *deg-angle*) *function*

Returns the value in radians of an angle in degrees.

(CROSS-PRODUCT *vector1 vector2*) *function*

Returns the cross-product vector of *vector1* and *vector2*.

(DOT-PRODUCT *vector1 vector2*) *function*

Returns the dot-product of *vector1* and *vector2*.

(LENGTH-OF-VECTOR *vector*) *function*

Returns the length (norm) of *vector*

(NORM-OF-VECTOR *vector*) *function*

Divides *vector* by its norm. Returns the normalized vector.

(ANGLE-BETWEEN-VECTORS *vector1 vector2 direction* **)** *function*

Returns the angle in radians between *vector1* and *vector2*. The vector *direction* determines the sign of the angle.

(POINT-LINE-DISTANCE *xyz line-xyz-1 line-xyz-2*) *function*

Returns the orthogonal distance between a point and a line given by the coordinates of two points.

(HOMO-PROD &rest *transf-matrices*) *function*

Returns the matrix obtained by making the matrix-product of the specified transformation matrices.

10.3. Display functions

vantage-window is the default window where all the display actions take place. It has an active region attached to it. When the LEFT mouse button is clicked any where inside the *vantage-window*, the *pop-up-menu* system is invoked. (Figure 8).

Selection of an item will result in one the following:

- It will fire a particular function.

  Ex: "**Erase-Screen**" will clear the *vantage-window*.

- Further *pop-up-menus* will show up.

  Ex: "**choose-body**" will list all the solids defined and the chosen value will become the default for the display system.

- It may ask for some input values.

  Ex: "**show-corres-frame**" displays the following message. (Figure 9)

```
Please respond by clicking:
Complete description of the frame =  Left-mouse-button
Only the Name                     =  Middle-mouse-button
```

DISPLAY-CAMERA *frame*

Name of the frame that defines the camera used for display of a 3-D object. It is defined like any other camera in vantage, and can be redefined at will. Its default definition is created by the command:

(camera* 'display-camera '(3000 30000 3000)) which defines an orthogonal projection from the point (3000 3000 3000) pointing to the origin (0 0 0).

(DRAW-BODY *body-name* **)** *function*

Draws the specified body on the *vantage-window*. The *body-name* should correspond to the boundary representation.

(DRAW-FACE *face-name* **)** *function*

Draws the specified face on the *vantage-window*.

(DRAW-EDGE *edge-name* **)** *function*

Draws the specified edge on the *vantage-window*.

(DRAW-VERTEX *vertex-name radius* **)** *function*

Draws the specified vertex on the *vantage-window*.

(SHADE-FACE *face-name* **)** *function*

Shades the given face depending on the face normal.

(SHADE-POLYGON *ink* **&rest** *lists* **)** *function*

Shades a region given by the set of lists of vertices. The first one correspond to the outer boundary and the remaining ones are the holes.

(VERTEX-MATCH *x-position y-position* **&optional** *close* **)** *function*

Returns the nearest displayed vertex on the *vantage-window* with respect to the given x and y positions. The current position of the mouse is stored in ***mouse-x*** and ***mouse-y***. If more than one vertex is encountered within the range given by *close*, it will return one of them.

(EDGE-MATCH *x-position y-position* **&optional** *close* **)** *function*

Returns the nearest displayed edge on the *vantage-window* with respect to the given x and y positions. If more than one edge is encountered within the range given by *close*, it will return one of them.

(FACE-MATCH *edge1 edge2* **)** *function*

Returns the name of a face that has both *edge1* and *edge2* as edges.

(FLASH-FACE *face-name* **)** *function*

Highlights or erases the existing highlight on the specified face. This is a very useful debugging tool. Multiple calls to **flash-face** results in a blinking effect.

(FLASH-EDGE *edge-name* **&optional** *width* **)** *function*

Highlights or erases the existing highlight on the specified edge. This is a very useful debugging tool. Multiple calls to **flash-edge** results in a blinking effect.

(DRAW-IMAGE *image-name* **)** *function*

Draws the specified 2d-image on the *vantage-window*.

(DRAW-REGION *region-name* **)** *function*

Draws the specified 2d-region on the *vantage-window*.

(DRAW-ARC *arc-name* **)** *function*

Draws the specified 2d-arc on the *vantage-window*.

(DRAW-JOINT *joint-name radius* **)** *function*

Draws the specified 2d-joint on the *vantage-window*.

(SHOW-AXIS &optional *length* **)** *function*

Draws the current x, y and z axis on the screen.

CURRENT-BODY *variable*

Name of the last body that has been selected on the choose-body menu.

CURRENT-IMAGE *variable*

Name of the last image that has been selected on the choose-image menu.

(**FIT-SCREEN** &optional *solid-name* *current-body*) *macro*

Adjusts the display size so that the specified body is entirely inside the window.

(**FIT-SCREEN*** &optional *solid-name* *current-body*) *function*

FIT-SCREEN* is like FIT-SCREEN, except that it evaluates its argument.

(**IMAGE-FIT-SCREEN** &optional *image-name* *current-image*) *macro*

Adjusts the display size so that the specified image is entirely inside the window.

(**IMAGE-FIT-SCREEN*** &optional *image-name* *current-image*) *function*

IMAGE-FIT-SCREEN* is like IMAGE-FIT-SCREEN, except that it evaluates its argument.

(**WINDOW-ZOOM**) *function*

Redisplays the portion of the window selected by two successive middle-mouse-button clicks. The selected region will be enlarged to fit the *vantage-window*.

ZOOMF *variable*

Controls the scale of the image.

(**ZOOM** *x*) *function*

Changes the scaling factor of the display (variable ***zoomf***). The current value is multiplied by *x*. The value of *x* should be greater than 0.

DASH-LEVEL *variable*

Controls the length of the line segments used to draw dashed lines.

(DASH *x* **)** *function*

Changes the *dash-level* variable. The current value is multiplied by *x*. The value of *x* should be greater than 0.

SHADE-LENGTH *variable*

Controls the vertical distance between two dots used for shading.

(SHADEL *x* **)** *function*

Changes the *shade-length* variable. The current value is multiplied by *x*. The value of *x* should be greater than 0.

SHADE-WIDTH *variable*

Controls the horizontal distance between two dots used for shading.

(SHADEW *x* **)** *function*

Changes the *shade-width* variable. The current value is multiplied by *x*. The value of *x* should be greater than 0.

(DISPLAY-SCENE *scene sensor* **)** *macro*

Displays the scene *scene* as seen from the sensor *sensor*, with hidden parts hidden. Just paints polygons from back to front.

(DISPLAY-SCENE* *scene sensor* **)** *function*

DISPLAY-SCENE* is like DISPLAY-SCENE, except that it evaluates its arguments.

(DISPLAY-PROPERTY *image property-name* **)** *macro*

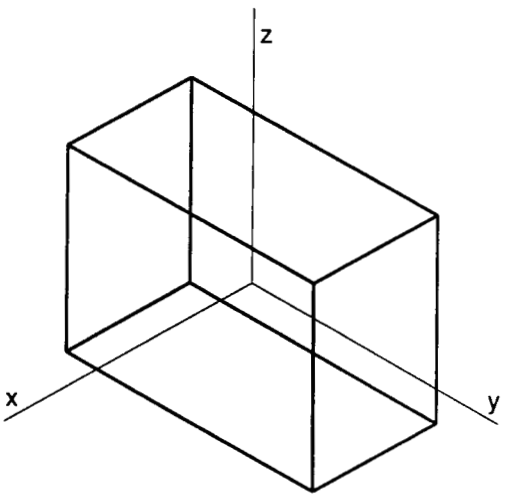
Displays and shades the property-regions of *image* for the property *property-name*.

(DISPLAY-PROPERTY* *image property-name* **)** *function*

DISPLAY-PROPERTY* is like DISPLAY-PROPERTY, except that it evaluates its arguments.

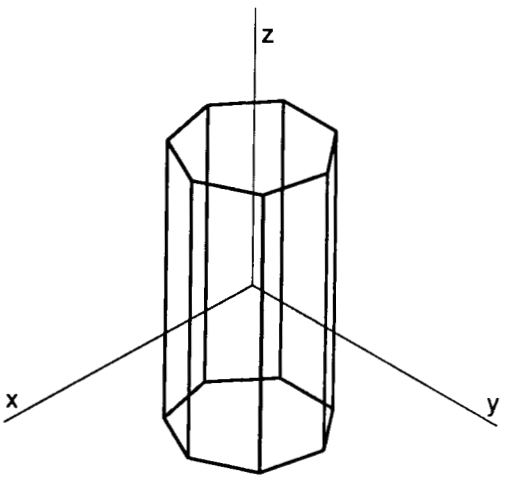
Appendix A
Primitive Solids

```
(csgnode primitive-1 cub (100 200 150))
```



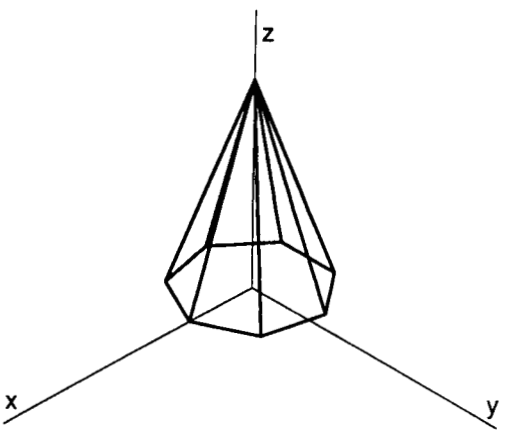

```
(csgnode primitive-2 cyl (50 200 7))
```

- 1 surface
- 2 curves



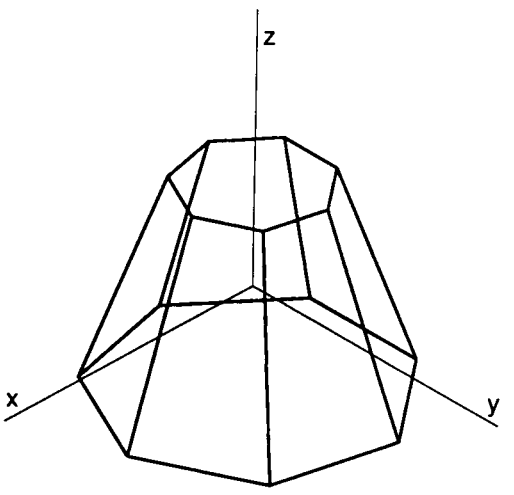

```
(csgnode primitive-3 con (50 150 7))
```

- 1 surface
- 1 curve



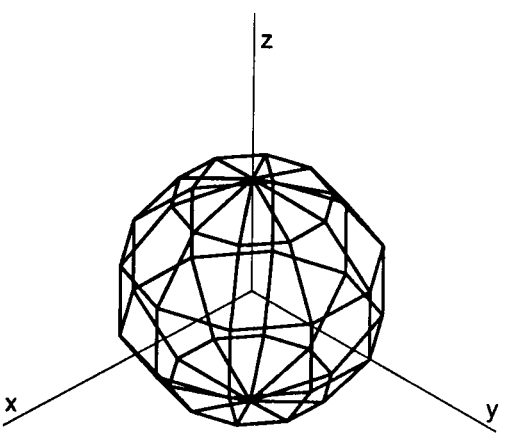
```
(csgnode primitive-4 tru (100 50 150 7))
```

- 1 surface
- 2 curves

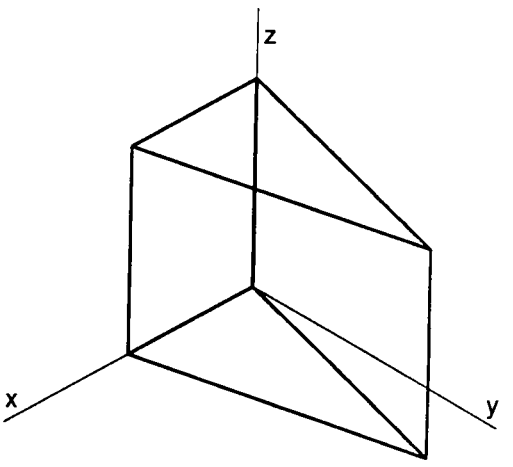



```
(csgnode primitive-5 sph (80 5))
```

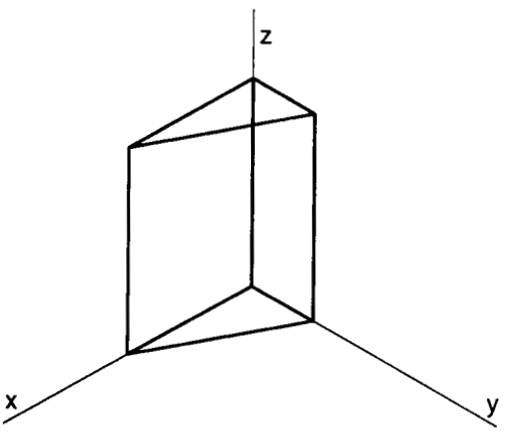
- 1 surface



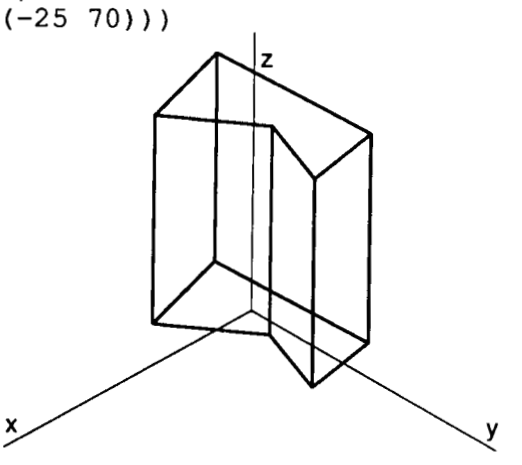

```
(csgnode primitive-6 iso (100 200 150))
```



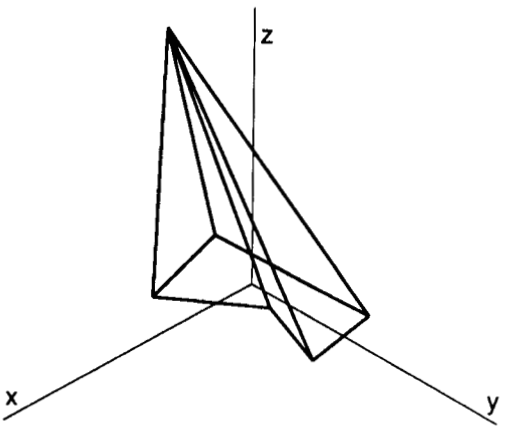
```
(csgnode primitive-7 rt (100 50 150))
```




```
(csgnode primitive-8 2.5p (150
                           (-20 -50) (50 -30)
                           (10 25) (30 80) (-25 70)))
```




```
(csgnode primitive-9 2.5c (150
                           (0 -70) (-20 -50) (50 -30)
                           (10 25) (30 80) (-25 70)))
```



Appendix B
Examples

This first example shows a lisp file which, when loaded, performs the following operations:

- generation of the CSG tree of an object (*body1*) from primitives
- creation of a boundary-representation of the object from the CSG tree
- display of the solid on the screen
- definition of a scene containing the object
- definition of a camera
- generation of an image of the scene using the camera
- display of the image on the screen

```
;; Save current angle unit (degrees or radians)
(save-angle-mode)
;;
;; Set angle unit to degrees
(setq *angle-mode* 'deg)
;;
;; Define csg representation for body1
(csgnode* 'b1 'cu '(500 300 111.5) :trans '(0 -68.1 49.25 0 0 0))
(csgnode* 'b2 'cyl '(120 450 10) :trans '(-130 6.9 6.5 0 0 -90))
(csgnode* 'b3 'mov 'b2 :trans '(260 0 0 0 0 0))
(csgnode* 'b4 'cu '(3000 3000 100) :trans '(0 0 155 0 0 0))
(csgnode* 'b5 'uni '(b1 b2))
(csgnode* 'b6 'uni '(b5 b3))
(csgnode* 'body1 'dif '(b6 b4) :fast 'all)
;;
;; Set the angle unit to its previous value
(restore-angle-mode)
;;
;; Generate boundary representation for body1 (called body1z)
(boun-rep* 'body1)
;;
;; Compute scaling and translation factors so that body1 fits on
;; the display window, and draw body1. The camera is the current
;; "display-camera", which has a default definition, but which can be
;; redefined
(fit-screen* 'body1)
;;
;; Define 3d scene
(scene* 'my-scene '(body1))
;;
;; Define a camera
(camera* 'cam1 '(2000 1000 500) :focal 5)
;;
;; Generate 2D description
(image* 'my-scene 'cam1 :image-name 'my-image)
;;
;; Compute scaling and translation factors so that my-image fits on
;; the display window, and draw my-image
(image-fit-screen* 'my-image)
;;
```

This second example shows an interactive session in which operations similar to the ones of the previous example are performed, plus the following operations:

- definition of a light-source
- generation of the 3D properties of the scene for the light-source
- generation of the 2D properties of the image by projection of the 3D properties of the scene
- display of the 2D property regions

Figures B-1 and B-2 show the resulting image and the property-regions associated with the light-source.

```
> (csgnode a1 cu (100 100 100))
A1
> (csgnode a2 cu (80 80 200))
A2
> (csgnode a3 cu (80 200 80))
A3
> (csgnode a4 cu (200 80 80))
A4
> (csgnode a5 dif (a1 a2))
A5
> (csgnode a6 dif (a5 a3))
A6
> (csgnode a7 dif (a6 a4) :fast all)
A7
> (csgnode ground cu (1000 1000 10) :trans (0 0 -100 0 0 0))
GROUND
> (boun-rep a7)
A7
> (boun-rep ground)
GROUND
> (scene s1 (a7 ground))
S1
> (camera c1 (300 -200 450) :focal 1)
C1
> (light-source l1 (-200 300 450) :focal 1)
L1
> (image s1 c1 :lights (l1) :image-name i1 :merge-shadows nil)
I1
> (image-fit-screen i1)
I1
> (paint-property-on-image i1 back-l1)
BACK-L1
> (paint-property-on-image i1 occluded-l1)
OCCLUDED-L1
> (paint-property-on-image i1 visible-l1)
VISIBLE-L1
> (merge-light-properties s1 l1)
L1
> (paint-property-on-image i1 occluded-l1)
OCCLUDED-L1
> (display-property i1 back-l1)
BACK-L1
> (display-property i1 split-occluded-l1)
SPLIT-OCCLUDED-L1
> (display-property i1 occluded-l1)
OCCLUDED-L1
> (display-property i1 visible-l1)
VISIBLE-L1
```

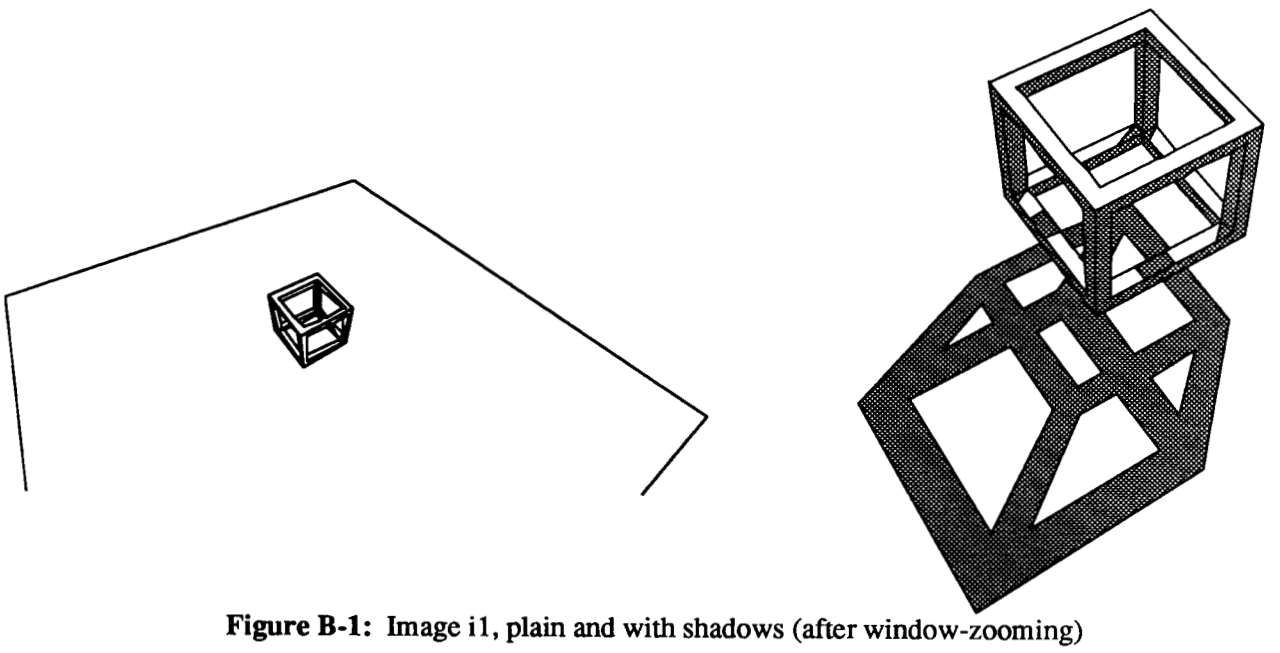


Figure B-1: Image i1, plain and with shadows (after window-zooming)

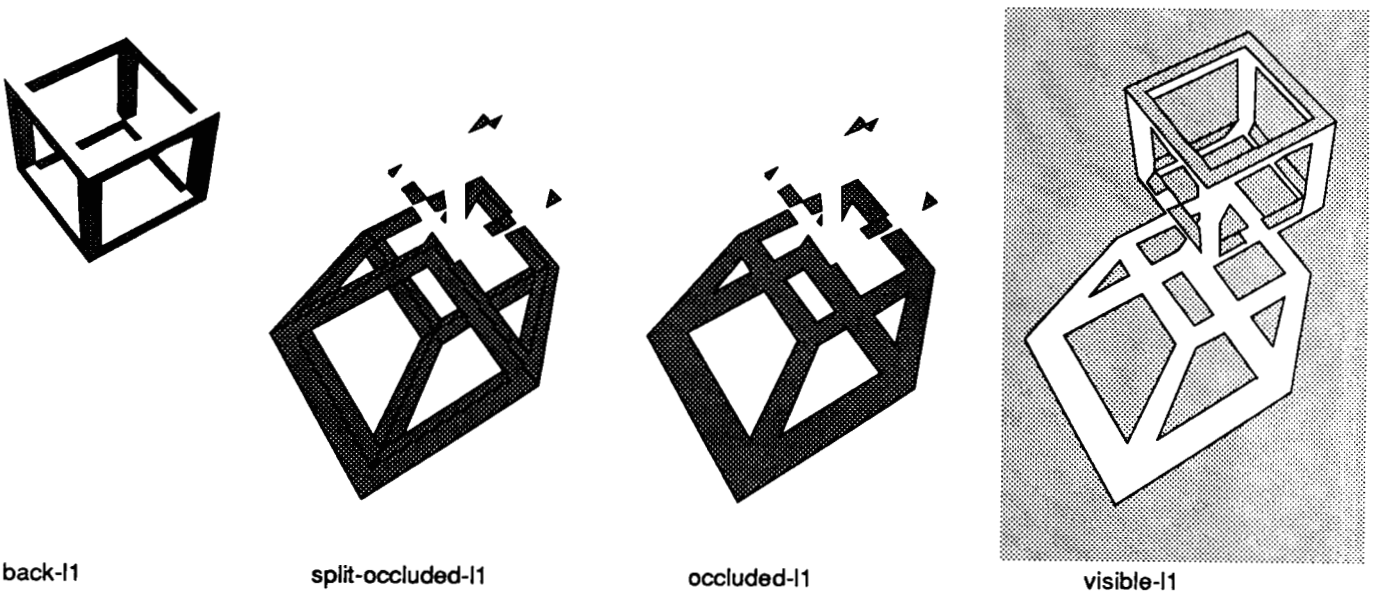


Figure B-2: I1 property-regions projected on i1

Appendix C
Standard Frames

The next pages give the definition of the frames used in VANTAGE. For each type of frame, the corresponding slots are listed, along with a brief description of each. Optional slots are marked with a "*".

CSG-NODE

Description: Representation of a CSG-node.

Frame:

(CSG-NODE-NAME
(is-a)
(class)
(type)
(parameters)
(rigid-motion)
(node-used-by-list)
(boundary-rep)
(group-approximate)
(merge)
* (node-left)
* (node-right)
* (fast)
* (surface-list)
* (curve-list))

Slots:

is-a
Value: **csg-node.**

class
Value: either **primitive, or operation.**
Specifies whether the node is a leaf-node (primitive solid) or results from an operation performed on its child node(s).

type
Value: either a primitive type (*e.g.* **cube. cylinder, cyl**, *etc.*) or an operation name (*e.g.* **union, dif, move, mir**, *etc.*).
This slot specifies the type of primitive or the type of operation the node corresponds to.

parameters
Value: List of float numbers.
For a primitive solid, the parameters that define it.

rigid-motion
Value: *motion-matrix.*
Defines the coordinate system attached to the node.

node-used-by-list
Value: list of *csg-nodes.*
The list contains the csg-nodes that are defined using the current node, *i.e.* the parent nodes of the node in the csg-tree.

boundary-rep
Value: *3d-body.*
Inverse: **body-csg-node.**
Points to the boundary-representation of the body defined by the node.

group-approximate

Value: **T**.
Is T when the grouping of the approximated faces and edges has been done.

merge

Value: **T**.
Is T when some faces have been merged.

node-left

Value: *CSG-NODE*.
For a solid that results from an operation, the left child of the node.

node-right

Value: *CSG-NODE*.
For a solid that results from an operation, the right child of the node.

fast

Value: **T**.
When it is **T**, then the boundary representation of the child nodes of the node will not be copied/saved before performing on them the operation that will create the boundary representation of the node.

surface-list

Value: list of *surfaces*.
List of the non-planar surfaces contained in the body.

curve-list

Value: list of *curves*.
List of the non-linear curves contained in the body.

3D-BODY

Description: Representation of a solid or body.

Frame:

(**3D-BODY-NAME**
(is-a)
(body-csg-node)
(body-rigid-motion)
(body-face-list)
(body-edge-list)
(body-vertex-list)
(body-cfg-list)
* (body-app-grouped-faces)
* (body-app-grouped-edges)
* (body-merged-faces))

Slots:

is-a
Value: **3d-body**.

body-csg-node
Value: *csg-node*.
Inverse: **boundary-rep**
Points to the csg-node that defines the body.

body-rigid-motion
Value: *motion-matrix*.
The transformation gives the location of the current body-coordinates frame in the world-coordinates.

body-face-list
Value: list of *3d-faces*.
Inverse: **face-body**.
Lists all the faces of the body.

body-edge-list
Value: list of *3d-edges*.
Inverse: **edge-body**.
Lists all the edges of the body.

body-vertex-list
Value: list of *3d-vertices*.
Inverse: **vertex-body**.
Lists all the vertices of the body.

body-cfg-list
Value: list of lists of *3d-faces*.
Lists all the closed groups of connected faces (cfg) of the body.

body-app-grouped-faces
Value: list of *3d-faces*.
Lists all the faces of the body that have been obtained by grouping a set of

connected faces that approximate a same surface.

body-app-grouped-edges

Value: list of *3d-edges*.

Lists all the edges of the body that have been obtained by grouping a set of connected edges that approximate a same curve.

body-merged-faces

Value: list of *3d-faces*.

Lists all the faces of the body that have been obtained by merging connected faces.

3D-FACE

Description: Representation of a face of a body.

Frame:

(**3D-FACE-NAME**
(is-a)
(face-body)
(face-type)
(face-geometry)
(out-boun-list)
(hole-boun-list)
* (face-surface)
* (face-class)
* (face-parent)
* (face-subdivision)
* (face-children)
* (face-properties)
* (app-grouped-out-boun-list)
* (app-grouped-hole-boun-list))

Slots:

is-a
Value: **3d-face**.

face-body
Value: *3d-body*.
Inverse: **body-face-list**.
Points to the 3d-body that contains the face.

face-type
Value: either **plane, cyl, sph, con**
Gives the type of the surface that contains the face.

face-geometry
Value: list of parameters.
For a planar face, lists the coordinates of the normal vector of the face and the orthogonal distance between the face and the origin.

out-boun-list
Value: list of lists of *3d-edges*.
Lists the outer boundaries of the face. For a planar face, there is only one outer-boundary.

hole-boun-list
Value: list of lists of *3d-edges*.
Lists the boundaries of the holes of the face.

face-surface
Value: *surface*.
Points to the surface that contains the face.

face-class

Value: either **global, app,** or **merge.**
Indicates whether and how this face was combined with other faces to generate a parent face. A **global** face has no parent face. An **app** face has a parent face obtained by grouping a set of connected faces that approximate a same surface. A **merge** face has a parent face obtained by merging a set of connected faces.

face-parent

Value: *3d-face.*
Inverse: **face-children.**
Points to the parent face of this face.

face-subdivision

Value: either **merge** or **app.**
Indicates whether and how this face is divided into children faces. It is **app** if the face was obtained by grouping a set of connected faces that approximate a same surface. It is **merge** if the face was obtained by merging a set of connected faces.

face-children

Value: list of *3d-faces.*
Inverse: **face-parent.**
Points to the list of faces that generated this face.

face-properties

Value: *property-list.*
Points to the frame that lists the properties of the face.

app-grouped-out-boun-list

Value: list of lists of *3d-edges.*
Lists the outer boundaries of the face, replacing every set of connected approximated edges by their parent edge.

app-grouped-hole-boun-list

Value: list of lists of *3d-edges.*
Lists the boundaries of the holes of the face, replacing every set of connected approximated edges by their parent edge.

3D-EDGE

Description: Representation of an edge of a body.

Frame:

(**3D-EDGE-NAME**
(is-a)
(edge-body)
(edge-type)
(p-face)
(n-face)
(pcw)
(nccw)
(pccw)
(ncw)
(p-vertex)
(n-vertex)
* (edge-curve)
* (edge-kind)
* (edge-class)
* (edge-parent)
* (edge-subdivision)
* (edge-children)
* (app-grouped-p-face)
* (app-grouped-n-face)
* (app-grouped-pcw)
* (app-grouped-nccw)
* (app-grouped-pccw)
* (app-grouped-ncw))

Slots:

is-a
Value: **3d-edge**.

edge-body
Value: *3d-body*.
Inverse: **body-edge-list**.
Points to the 3d body that contains the edge.

edge-type
Value: either **line**, **cir**, or any combination of 2 surfaces chosen among **plane**, **cyl**, **con**, **sph** (*e.g.* **plane-cyl**, **con-con**, **cyl-sph**, *etc*).
Gives the type of the curve that contains the edge, or the types of the 2 surfaces whose intersection contains the edge.

p-face
Value: *3d-face*.
Points to the p-face in the winged-edge representation of this edge.

n-face
Value: *3d-face*.
Points to the n-face in the winged-edge representation of this edge.

pcw
Value: *3d-edge*.
Points to the pcw-edge in the winged-edge representation of this edge.

nccw
Value: *3d-edge*.
Points to the nccw-edge in the winged-edge representation of this edge.

pccw
Value: *3d-edge*.
Points to the pccw-edge in the winged-edge representation of this edge.

ncw
Value: *3d-edge*.
Points to the ncw-edge in the winged-edge representation of this edge.

p-vertex
Value: *3d-vertex*.
Points to the p-vertex in the winged-edge representation of this edge.

n-vertex
Value: *3d-vertex*.
Points to the n-vertex in the winged-edge representation of this edge.

edge-curve
Value: *curve*.
Points to the curve that contains the edge.

edge-kind
Value: **aux**.
aux indicates that the edge is an auxiliary edge which divides a curved surface into auxiliary planar surfaces for the purpose of approximation.

edge-class
Value: either **global** or **app**.
Indicates whether this face was combined with other faces to generate a parent face. A **global** edge has no parent edge. An **app** face has a parent face obtained by grouping a set of connected faces that approximate a same curve.

edge-parent
Value: *3d-edge*.
Inverse: **edge-children**.
Points to the parent edge of this edge.

edge-subdivision
Value: **app**.
This slot indicates whether this edge is divided into children edges. It is **app** if the edge was obtained by grouping a set of connected edges that approximate a same curve.

edge-children
Value: list of *3d-edges*.
Inverse: **edge-parent**.
Points to the list of edges that generated this edge.

app-grouped-p-face
Value: *3d-face*.
Parent-face of the p-face of the edge.

app-grouped-n-face
Value: *3d-face*.
Parent-face of the n-face of the edge.

app-grouped-pcw
Value: *3d-edge.*
Parent-edge of the pcw-edge of the edge.

app-grouped-nccw
Value: *3d-edge.*
Parent-edge of the nccw-edge of the edge.

app-grouped-pccw
Value: *3d-edge.*
Parent-edge of the pccw-edge of the edge.

app-grouped-ncw
Value: *3d-edge.*
Parent-edge of the ncw-edge of the edge.

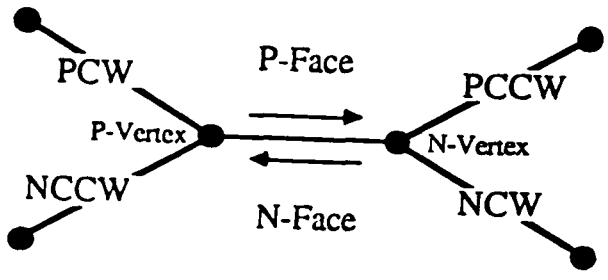


Figure C-1: Winged-edge representation

3D-VERTEX

Description: Representation of a vertex of a body.

Frame:

(**3D-VERTEX-NAME**
(is-a)
(vertex-body)
(one-of-the-edges)
(xyz-values)
* (display-xy))

Slots:

is-a
Value: **3d-vertex.**

vertex-body
Value: *3d-body.*
Inverse: **body-vertex-list.**
Points to the 3d body that contains the vertex.

one-of-the-edges
Value: *3d-edge.*
Points to one of the 3d edges that have this point as an end point.

xyz-values
Value: list of three float numbers.
Gives the list (x-coordinate, y-coordinate, z-coordinate) of the vertex with respect to the body coordinates.

display-xy
Value: list of two integers.
Gives the coordinates of the vertex with respect to the screen coordinates.

MOTION-MATRIX

Description: Representation of a motion matrix.

Frame:

(**MOTION-MATRIX-NAME**
(is-a)
(matrix-name)
(first-row)
(second-row)
(third-row))

Slots:

is-a
Value: **motion-matrix.**

matrix-name
Value: array (3 4) of float numbers.
Points to the matrix, defined by the **make-array** function.

first-row
Value: list of 4 float numbers.
Lists the elements of the first row of the matrix.

second-row
Value: list of 4 float numbers.
Lists the elements of the second row of the matrix.

third-row
Value: list of 4 float numbers.
Lists the elements of the third row of the matrix.

SURFACE

Description: Definition of a surface.

Frame:

(SURFACE-NAME
(is-a)
(type)
(parameters)
(rigid-motion))

or:

(SURFACE-NAME
(is-a)
(type)
(move)
(rigid-motion))

or:

(SURFACE-NAME
(is-a)
(type)
(mirror)
(mirror-plane)
(rigid-motion))

Slots:

is-a
Value: **surface.**

type
Value: either **cyl, con,** or **sph.**
Geometric type of the surface.

parameters
Value: list of float numbers defining the surface.
The list contains the radius for a cylinder and a sphere, and a radius and a height for a cone.

rigid-motion
Value: *motion-matrix.*
The motion-matrix gives either the position of the surface in the world coordinates frame, or, when the **move** or **mirror** slot exists, the transformation to apply to the specified surface.

move
Value: *surface.*
If specified, indicates that the surface is obtained by applying the specified rigid-motion to the specified surface.

mirror

Value: *surface*.
If specified, indicates that the surface is obtained by applying the specified mirror operation to the specified surface, and then by applying the specified rigid-motion.

mirror-plane

Value: list of 4 float numbers.
Defines the mirror plane by listing the coordinates of its normal vector and the orthogonal distance between the plane and the origin.

CURVE

Description: Definition of a curve

Frame:

(**CURVE-NAME**
(is-a)
(type)
(parameters)
(rigid-motion)
(inter))

or:

(**CURVE-NAME**
(is-a)
(type)
(move)
(rigid-motion))

or:

(**CURVE-NAME**
(is-a)
(type)
(mirror)
(mirror-plane)
(rigid-motion))

Slots:

is-a
Value: **curve**.

type
Value: either **cir** or any combination of 2 surfaces chosen among **plane**, **cyl**, **con**, **sph** (*e.g.* **plane-cyl**, **con-con**, **cyl-sph**, *etc*).
Gives the type of the curve, or the types of the 2 surfaces whose intersection generate the curve.

parameters
Value: list of float numbers defining the curve.
When the type of the curve is **cir**, it lists the radius and elevation (z-coordinate) of the curve.

rigid-motion
Value: *motion-matrix*.
The motion-matrix gives either the position of the curve in the world coordinates frame, or, when the **move** or **mirror** slot exists, the transformation to apply to the specified curve.

move
Value: *curve*.

If specified, indicates that the curve is obtained by applying the specified rigid-motion to the specified curve.

mirror

Value: *curve*.
If specified, indicates that the curve is obtained by applying the specified mirror operation to the specified curve, and then by applying the specified rigid-motion.

mirror-plane

Value: list of 4 float numbers.
Defines the mirror plane by listing the coordinates of its normal vector and the orthogonal distance between the plane and the origin.

inter

Value: list containing 2 *surfaces* or one *surface* and one plane.
Points to the 2 surfaces whose intersection defines the curve. If one surface is a plane, it is specified by a list containing the coordinates of its normal vector and the orthogonal distance between the plane and the origin.

3D-SCENE

Description: Definition of a 3d scene.

Frame:

(**3D-SCENE-NAME**
(is-a)
(csg-node-list)
(light-list)
(x-max)
(y-max)
(z-max)
(x-min)
(y-min)
(z-min))

Slots:

is-a
Value: **3d-scene**.

csg-node-list
Value: list of *csg-nodes*.
Lists the bodies that are in the scene.

light-list
Value: list of *sensors*.
Lists the light-sources that have been used in the scene.

x-max, y-max, z-max, x-min, y-min, z-min
Value: float numbers.
Maximum and minimum x, y, z coordinates of the scene.

SENSOR

Description: Definition of a sensor.

Frame:

(**SENSOR-NAME**
(is-a)
(type)
(parameters)
(rigid-motion)
* (focal-length)
* (limit-angle))

Slots:

is-a
Value: **sensor**.

type
Value: either **camera** or **light**.

rigid-motion
Value: *rigid-motion* Defines the frame of coordinates of the sensor. The z-axis of this frame is the "viewing" direction, pointing from the object to the sensor.

parameters
Value: list of 6 float numbers.
Lists the parameters (*x y z roll pitch yaw*) that define the frame of coordinates of the sensor.

focal
Value: float number.
Focal-length of the sensor. Gives the distance between the projection-plane and the sensor. For a light-source, no specified focal-length means a parallel light-source, whereas a specified focal-length corresponds to a perspective light-source.

limit-angle
Value: float number, or **NIL**.
Maximum angle (in degrees) between the normal of a face and the projection direction, for which the face is visible. **NIL** corresponds to a limit-angle of 90 degrees.

PROPERTY-LIST

Description: Representation of properties for a particular 3D face or 2D region.

Frame:

(PROPERTY-LIST-NAME
(is-a)
(*property-1*)
(*property-2*)
.
.
...)

Slots:

is-a
Value: **property-list.**

property-1*, *property-2*, *etc.
Each slot represents a different property specified by the name of the slot. Value: list of polygons, where a polygon is a list of boundaries (outer, then hole(s)), and where a boundary is a list of vertices (each one represented by a list containing its x and y coordinates (in 2-D), or x, y and z coordinates (3-D)), or **T.**
Defines the region(s) of the face where the property applies. If **T**, the property applies to the whole face.

PROPERTY

Description: Each *property* frame has a name that is the name of a property (*e.g.* **occluded-L1**), and lists the 3-D faces and the 2-D regions that have that property.

Frame:

(PROPERTY-NAME
(is-a)
(*3d-faces*)
(*2d-regions*))

Slots:

is-a
Value: **property**.

3d-faces
Value: list of faces (in one or several scene(s)) that have the property.

2d-regions
Value: list of regions (in one or several image(s)) that have the property.

2D-IMAGE

Description: Definition of a 2d-image.

Frame:

(2D-IMAGE-NAME
(is-a)
(image-3d-scene)
(image-camera)
(image-light-source-list)
(image-region-list)
(image-arc-list)
(image-joint-list)
(image-bounding-box))

Slots:

is-a
Value: **2d-image.**

image-3d-scene
Value: *3d-scene.*
Specifies the 3d scene that is projected on the image.

image-camera
Value: *sensor.*
Specifies the camera that is used to generate the image.

image-region-list
Value: list of *2d-regions.*
Inverse: **region-image.**
Lists all the regions of the image that result from the projection of a 3d-face.

image-arc-list
Value: list of *2d-arcs.*
Inverse: **arc-image.**
Lists all the arcs (line-segments) of the image.

image-joint-list
Value: list of *2d-joints.*
Inverse: **joint-image.**
Lists all the joints (vertices) of the image.

image-bounding-box
Value: list of 4 float numbers.
Lists the minimum and maximum coordinates of the image (*x-min y-min x-max y-max*).

2D-REGION

Description: Representation of a region of a 2d-image

Frame:

(**2D-REGION-NAME**
(is-a)
(region-image)
(3d-face)
(region-bounding-box)
(region-out-boun-list)
(region-hole-boun-list)
* (region-properties))

Slots:

is-a
Value: **2d-region.**

region-image
Value: *2d-image.*
Inverse: **image-region-list.**
Points to the 2d-image that contains the region.

3d-face
Value: *3d-face.*
Points to the 3d-face that generated the region.

region-bounding-box
Value: list of 4 float numbers.
Lists the minimum and maximum coordinates of the region on the image (*x-min y-min x-max y-max*).

region-out-boun-list
Value: list of *2d-arcs.*
Lists the outer boundary of the region.

region-hole-boun-list
Value: list of lists of *2d-arcs.*
Lists the boundaries of the holes of the region.

region-properties
Value: *property-list.*
Points to the frame that lists the properties of the region.

2D-ARC

Description: Representation of an arc of a 2d-image.

Frame:

(**2D-ARC-NAME**
(is-a)
(arc-image)
(3d-edge)
(p-joint)
(n-joint)
(p-region)
(n-region)
(pcw)
(nccw)
(pccw)
(ncw))

Slots:

is-a
Value: **2d-arc.**

arc-image
Value: *2d-image.*
Inverse: **image-arc-list.**
Points to the 2d image that contains the arc.

3d-edge
Value: *3d-edge.*
Points to the 3d-edge that generated the arc.

p-joint
Value: *2d-joint.*
Points to the p-joint in the winged-edge representation of this arc.

n-joint
Value: *2d-joint.*
Points to the n-joint in the winged-edge representation of this arc.

p-region
Value: *2d-region.*
Points to the p-region in the winged-edge representation of this arc.

n-region
Value: *2d-region.*
Points to the n-region in the winged-edge representation of this arc.

pcw
Value: *2d-arc.*
Points to the pcw-arc in the winged-edge representation of this arc.

nccw
Value: *2d-arc.*
Points to the nccw-arc in the winged-edge representation of this arc.

pccw

Value: *2d-arc*.
Points to the pccw-arc in the winged-edge representation of this arc.

ncw

Value: *2d-arc*.
Points to the ncw-arc in the winged-edge representation of this arc.

2D-JOINT

Description: Representation of a joint of an image.

Frame:

(2D-JOINT-NAME
(is-a)
(joint-image)
(x)
(y)
* (3d-vertex-list)
* (display-xy))

Slots:

is-a
Value: **2d-joint.**

joint-image
Value: *2d-image.*
Inverse: **image-joint-list.**
Specifies the *2d-image* that contains the joint.

x
Value: float number.
Gives the x-coordinate of the joint with respect to the camera coordinates.

y
Value: float number.
Gives the y-coordinate of the joint with respect to the camera coordinates.

3d-vertex-list
Value: list of *3d-vertices.*
Specifies the 3d-vertex or 3d-vertices that generated the joint.

display-xy
Value: list of two integers.
Gives the coordinates of the joint with respect to the screen coordinates.

Appendix D
Framekit+ functions

FRAMEKIT is a frame-based knowledge representation, written in COMMON LISP, that provides the basic mechanisms of frames, inheritance, demons and views. It has been developed by Center for Machine Translation, Carnegie Mellon University. This section explains briefly some of the important and heavily used **FRAMEKIT** functions. This is by no means a complete description. **[Warning::Some of the key word arguments to the functions and the like are omitted here.]** The users are advised to go through the separate document titled **"The FRAMEKIT User's Guide"**.

D.1. Frames

A *frame* is a multi-level data structure, much like a record structure in traditional programming languages, that is used to store information used by COMMON LISP programs. A large collection of frames is sometimes called a *knowledge base*. Because frames also support *demons* and *inheritance* they are particularly useful for representing the knowledge in AI programs.

Frames are abstract data types comprised of slots, facets, views and fillers. Each frame can have any number of slots . Each slot can have any number of facets , and each facet can have any number of views and each view can have any number of fillers . Frames differ from traditional record structures in that slots, facets and views can be allocated and removed at run time. There are some facets that are pre-defined by FRAMEKIT to handle demons and inheritance.

The general structure of a frame is as follows.

```
(FrameName
   (SlotName   (VALUE          (VIEW list-of-values)
                               (VIEW list-of-values))
               (IF-ADDED       demon-list)
               (IF-NEEDED      demon-list)
               (IF-ERASED      demon-list)
               (IF-ACCESSED    demon-list)
               (RESTRICTIONS   predicate-list)
               (DEFAULT        list-of-values)
               ....
               (user-defined   .........)
               (...            .........))
(SlotName   ....)
```

```
....)
```

Please refer to FRAMEKIT manual for the complete syntax.

D.2. Frame creation

(CREATE-FRAME *frame* **)** *function*

The argument to CREATE-FRAME must be a symbol. The symbol is checked to see if a frame of that name already exists; if not, a new frame is created and added to *FRAME-LIST*. The frame name is returned if the creation took place, otherwise NIL is returned.

(CREATE-SLOT *frame slot* **)** *function*

If the frame already exists, CREATE-SLOT checks to see if the slot is already present; if not, a new slot is created. If the frame doesn't exist, it will either automatically create the frame or print a warning message. The slot name is returned if a new slot is created; otherwise NIL is returned.

(CREATE-FACET *frame slot facet* **)** *function*

If the frame and slot already exist, CREATE-FACET checks to see if the facet is already present; if not, a new facet is created. If either the frame or the slot doesn't exist, it will either automatically create them or print a warning message. The facet name is returned if a new facet is created; otherwise NIL is returned.

Examples:

```
> (create-frame 'dog)
> (create-slot 'dog 'weight)
> (create-facet 'tiger 'weight 'value)
> (create-facet 'cat 'race 'if-needed)
```

(MAKE-FRAME *frame-name* &rest *fullframe* **)** *macro*

MAKE-FRAME is a macro for defining frames in a file or at the Lisp top-level. The first argument is interpreted as the name of the frame to create; the rest of the arguments are interpreted as fully-specified slot definitions. The frame name is returned. For example:

```
> (make-frame my-frame
     (slot1 (facet1 (view1 filler-list1)
                    (view2 filler-list2))
```

```
            (facet2 (view3 filler-list2)))
      (slot2 (facet3 (view4 filler-list3))
            (facet4 (view1 filler-list4))))

my-frame
>
```

Any number of slots may be defined, each with any number of facets. Each facet may contain any number of views, each with any number of fillers.

(**MAKE-FRAME*** *frame-name fullframe*) *function*

MAKE-FRAME* is like MAKE-FRAME, except that it evaluates it arguments, and the slot definitions must be specified as a single list.

(**MK-FRAME** *frame-name* &rest *fullframe*) *macro*

MK-FRAME is another macro for defining frames in a file or at the Lisp top-level; unlike MAKE-FRAME, MK-FRAME accepts slot definitions in abbreviated form:

```
(mk-frame my-frame2
     (slot1 value-list1)
     (slot2 value-list2)
     ...
     ...
     (slotn value-listn))
```

Each filler is placed in the COMMON view of the VALUE facet of the specified slot.

(**MK-FRAME*** *frame-name* &rest *fullframe*) *function*

MK-FRAME* is like MK-FRAME, except that it evaluates it arguments, and the slot definitions must be specified as a single list.

D.3. Update Functions

(**ADD-VALUE** *frame slot filler*) *function*

Adds a filler to the VALUE facet in the specified slot, unless that filler already exists.

(**ADD-VALUES** *frame slot filler-list*) *function*

ADD-VALUES is just like ADD-VALUE, except that it accepts a list of fillers to add to the

VALUE facet all at once.

(ADD-FILLER *frame slot facet filler* **)** *function*

Similar to ADD-VALUE but operates on the specified facet instead of the VALUE facet.

(ADD-FILLERS *frame slot facet filler-list* **)** *function*

Similar to ADD-VALUES but operates on the specified facet instead of the VALUE facet.

(ERASE-VALUES *frame slot* **)** *function*

Erases all the fillers of the VALUE facet of the specified slot.

(ERASE-FILLER *frame slot facet filler* **)** *function*

Removes the given filler from the specified facet of the slot.

(ERASE-FACET *frame slot facet* **)** *function*

Deletes the named FACET from the named SLOT of the frame.

(ERASE-SLOT *frame slot* **)** *function*

The given SLOT is removed from the frame. If the slot is a relation, the inverse link will be erased in the corresponding frame.

(ERASE-FRAME *frame* **)** *function*

The frame is erased, by erasing slots one by one (thus eliminating any inverse links and then erasing the frame itself). Returns nil.

(ERASE-FRAMES *frame-list* **)** *function*

ERASE-FRAME is applied on each element of the frame-list.

(REPLACE-VALUE *frame slot filler* **)** *function*

A composition of ERASE-VALUES and ADD-VALUE. Erases the VALUE facet fillers for the specified view, and adds the given filler to the VALUE facet.

(REPLACE-FILLER *frame slot facet filler* **)** *function*

A composition of ERASE-FILLER and ADD-FILLER. Erases the facet fillers and adds the given filler to that facet.

Examples:

```
> (add-filler 'edge123 'pcw 'value 'edge86)
> (add-value 'dog 'color 'white)
> (replace-value 'dog 'color 'brown)
> (erase-values 'dog 'color)
> (erase-facet 'dog 'race 'if-needed)
> (erase-slot 'edge123 'pcw)
> (erase-frame 'dog)
```

D.4. Access Functions

(GET-VALUES *frame slot* **)** *function*

Returns a list of fillers of the VALUE facet in the specified slot.

(GET-FILLERS *frame-name slot facet* **)** *function*

Returns a list of fillers of the specified facet.

(SLOT-NAMES *frame* **)** *function*

Returns a list containing the names of the slots in the specified frame.

(FACET-NAMES *frame slot* **)** *function*

Returns a list containing the names of the facets in the specified slot.

D.5. Miscellaneous Functions and Variables

(FRAME-P *frame* **)** *function*

Returns the frame name if it exists otherwise nil.

FRAME-LIST *variable*

When FRAMEKIT creates a frame, its name is added to *FRAME-LIST*. When a frame is erased, it is removed from *FRAME-LIST*. The order of the frames in *FRAME-LIST* indicates from left to right the order in which they were created.

FKTRACE *variable*

If *FKTRACE* is non-NIL, FRAMEKIT will print trace information concerning each FRAMEKIT action that is evaluated. Although this results in a lot of output, it is useful for debugging purposes, since operations that are not always evident to the user (like demon invocation and automatic structure creation) become visible when tracing is enabled. Initial value is NIL.

FKWARN *variable*

If *FKWARN* is non-NIL, FRAMEKIT will inform the user about warning conditions that are non-fatal, but require some user notification (e.g., trying to add a filler to a facet when that filler is already present). Initial value is nil.

!FRAME, !SLOT, !FACET and **!FILLER** are the special variables which store the current frame, slot, facet and filler respectively at the time a *demon* mechanism is invoked. They can be used by the functions fired at that moment.

Index