Genesis
Reference Manual

J.A. Heisserman

EDRC 48-23-91

Genesis
Reference Manual

Jeff Heisserman

August 1991

Engineering Design Research Center
Carngie Mellon University
Pittsburgh, PA 15213

Abstract

Boundary solid grammars is a rule-based formalism for generating complex models of rigid solid objects. Solids are represented by their boundary elements, i.e. vertices, edges, and faces, with coordinate geometry associated with the vertices. Non-geometric data may be associated with any of these elements. Rules are used to match conditions of a solid or collection of solids and then modify them or create additional solids. A boundary solid grammar consists of an initial solid and a set of rules. It produces a language or family of solid models.

The boundary solid grammar formalism is implemented in the **Genesis** boundary solid grammar interpreter. **Genesis** provides facilities for representation and display of solids, match conditions, solid modeling operations, rule and grammar definition, and searching through the language of a grammar.

This reference manual details the facilities provided by the **Genesis** boundary solid grammar interpreter.

Contents

Chapter 1

Introduction

Boundary solid grammars [4] is a rule-based formalism for generating complex models of rigid solid objects. Solids are represented by their boundary elements, i.e. vertices, edges, and faces, with coordinate geometry associated with the vertices. Non-geometric data may be associated with any of these elements. Rules are used to match conditions of a solid or collection of solids and then modify them or create additional solids. A boundary solid grammar consists of an initial solid and a set of rules, and produces a language or family of solid models.

The boundary solid grammar formalism is implemented in the **Genesis** boundary solid grammar interpreter. **Genesis** provides facilities for representation and display of solids, match conditions, solid modeling operations, rule and grammar definition, and searching through the language of a grammar. **Genesis** consists of a logic programming language interpreter allowing keyboard input and the description of solid rules and grammars, a solid modeling database, a label database for non-geometric data, and interactive graphics for display and manipulation of solid models.

The current version of **Genesis** is constructed on IBM's compiler-based implementation of the **CLP**($\mathcal{R}$) programming language [5]. Models in the solid modeling and label databases are constructed, modified and accessed with built-in predicates connected within the **CLP**($\mathcal{R}$) compiler. Graphics routines are called as built-in predicates and access these databases directly to display the models. Additional predicates for matching features of the models, applying operations, and applying rules are written in **CLP**($\mathcal{R}$).

This reference manual details the facilities provided by the **Genesis** boundary solid grammar interpreter.

Chapter 2

Running Genesis

For the purposes of this discussion, **Genesis** is composed of two parts: the CLP*(TZ)* interpreter and **Genesis** modeler and graphics routines (written in C); and **Genesis** predicates (written in CLP(7£)). To run **Genesis** , the user must run the CLP(7£) / **Genesis** executable code, then load the **Genesis** predicates. This chapter describes this process.

2.1 Setting the Environment

In order for the CLP(TJ) system to find the correct initialization file, set the CLPRLIB environment variable to the directory containing the CLP(TJ)initialization file **init.clpr.** The initialization file is generally located in the **/usr/misc/ .genesis/prolog** directory, and the following command will set the correct path:

setenv CLPRLIB **/usr/misc/.genesis/prolog**

This command may be inserted in your .**login** file.

2.2 Loading Genesis

The CLP(7£) / **Genesis** executable code is generally located in

/usr/misc/ .genesis/bin/genesis

When this is executed, the CLP(TS) interpreter will respond with the following header and prompt:

```
CLP(R) Version 1.0
(c) Copyright International Business Machines Corporation 1989
All Rights Reserved

1 ?-
```

At this point the user should set any desired options (described in the next section), and load the **Genesis** predicates with `load_load` followed by `load(Rules)`.

`load_load`
: Loads the predicates that load the rest of the **Genesis** system (**CLP($\mathcal{R}$)** portion).

`load(Rules)`
: Loads the **Genesis** system including the given rule set. `load(Rules)` uses the `graphics` and `backtrack` predicates to load the appropriate files. `Rules` may be a grammar from the misc collection or a user created file specified relative to the users current directory (e.g. `load('./myrules/this_grammar.clpr').`).

2.3 Options

Genesis can run with or without graphics, and with forward solid rule application or with undo capability for searching the derivations of a grammar. The predicates in this section allow the user to load the **Genesis** system with the desired settings.

`graphics`
: Is the graphics switch turned on? If so, the system will return "Yes", otherwise, it will return "No".

`set_graphics(X)`
: Turns the graphics switch on (X = 1) or off (X = 0).

`backtrack`
: Is backtracking of the solid rules turned on?

`set_backtrack(X)`
: Turns solid rule backtracking switch on (X = 1) or off (X = 0).

The `graphics` and `backtrack` switches must be set prior to loading the **Genesis** predicates, and will have no effect if changed later during the session. Once the `graphics` and `backtrack` switches are set properly, the user should execute a `load_load`, followed by `load(Rules)`.

2.4 Exiting Genesis

The user may exit from **Genesis** and terminate the process using the **halt** predicate, as is standard for Prolog and logic programming systems. The `^D` (Control D) keystroke serves as an abbreviated form of the **halt** predictate.

It is occasionally useful to terminate a query during its execution with the `^C` (Control C) keystroke. `^C` will return control to the top level of the interpreter.

Chapter 3

Constructing and Using Grammars

Users interact with **Genesis** in several ways. They may use **Genesis** as a boundary representation solid modeler, using local operations, unary shape operations, and Boolean operations. **Genesis** provides facilities for defining matching conditions and operations, defining rules, constructing initial solids, applying rules, and searching for derivations in the language of a grammar. These facilities allow the user to experiment with generative grammars, creating alternative initial solids, defining new rules, and applying rules in different ways to generate interesting and useful models.

The focus of this chapter is on the construction of grammars, and the facilities provided by **Genesis** for exploring the space of derivations of these grammars.

3.1 Initial Solids

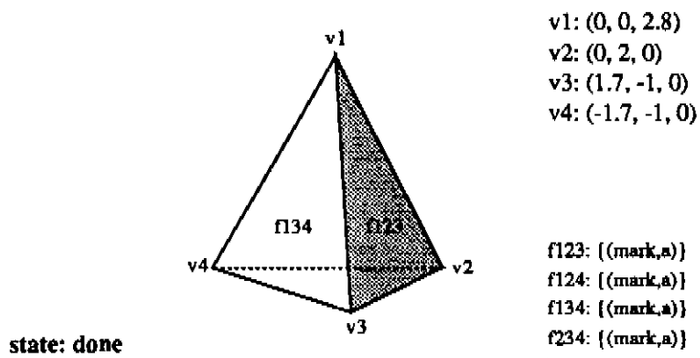

Figure 3.1: An initial solid.

The initial solid of a solid grammar is a topologically valid solid or collection of solids. An initial solid may be an empty solid (nothing there) and rules can be used to create solids. An initial solid may be created "by hand" with **Genesis** , using the operations presented in Chapters 6, 7, and 8. An initial solid created on a different solid modeling system, and loaded into **Genesis** using the facilities described in Chapter 10. A solid could be generated by one grammar and used as an initial solid in another grammar.

Modifications of the initial solid, and subsequent solids, may be accomplished by the application of the set of solid rules.

3.2 Reasoning About Solids

Conditions or features of solids are expressed as clauses in first order logic. Explicit conditions of a given solid correspond to axioms about the boundary representation. Clauses (in the form of Horn clauses) allow deduction of complex conditions of the solids from simpler conditions. In this way, arbitrarily complex conditions may be specified using deductive reasoning on the solid representation. Locating a condition of a solid then becomes a matter of satisfying a goal clause that specifies the desired condition.

A user may locate any of the existing topological elements using the provided predicates. For example, a user may locate any vertex using `vertex(V)`, where `V` becomes bound to the identifier of an existing vertex. A user may also find elements related by any of the topological adjacency relations. For example, a user may locate an edge-half associated with a given vertex using the `vertex_eh(V,E)` relation.

A large number of conditions are provided in **Genesis** . The queries on topological elements and topological adjacency relations are described in Chapter 4. The queries on the geometry associated with topological elements are described in Chapter 5. The queries on labels on topological elements and the current state are described in Chapter 8.

Using these conditions and operations, users can interactively locate features of solids, and modify solids with operations. They can construct their own conditions and operations. A user can use any of these conditions and operations to define solid rules.

3.3 Constructing Rules

Solid rules are described with three components: a description, a left-hand side, and a right-hand side. The description is a textual description of the rule that will be presented to the user as the rule is matched. The left-hand side is the set of match conditions that must be satisfied in order to apply the rule. The right-hand side is the sequence of operations and match conditions that transform the matched solids.

`description(RuleName, Description)`
The description of a rule that will be presented to the user when the rule is applied.

`lhs(RuleName, VariableList, DisplayList)`
The match conditions of a rule. Free variables that are needed for the `rhs` are passed through `VariableList`. Elements may be highlighted to show the user where the rule is being applied. These elements should be listed in `DisplayList`.

`rhs(RuleName, VariableList)`
The operations of the rule. Free variables are passed via `VariableList` and operations are applied to transform the model(s).

`describe(Rule)`
Print the rule name and description to the screen.

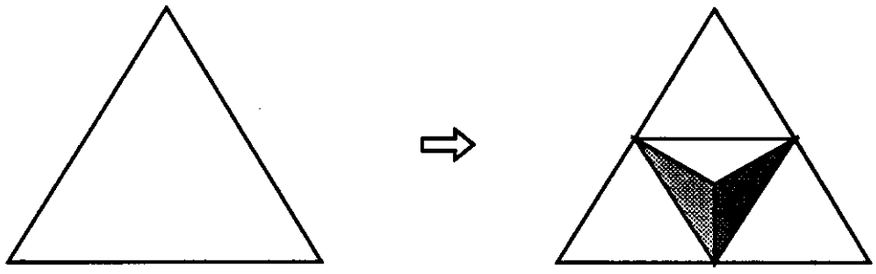

Figure 3.2: A simple solid rule.

A simple rule is presented below.

```
description(point_1, 'Build a point on a face.').

lhs(point_1, [F123], [F123]):-
        face(F123).

rhs(point_1, [F123]):-
        face_eh(F123, FirstEh),
        cw_non_colinear_eh(FirstEh, Eh12),
        cw_non_colinear_eh(Eh12, Eh23),
        eh_distance(Eh12, Eh23, Length),
        face_midpoint_esplit(F123),
        point_face(F123, Length/2.44949).
```

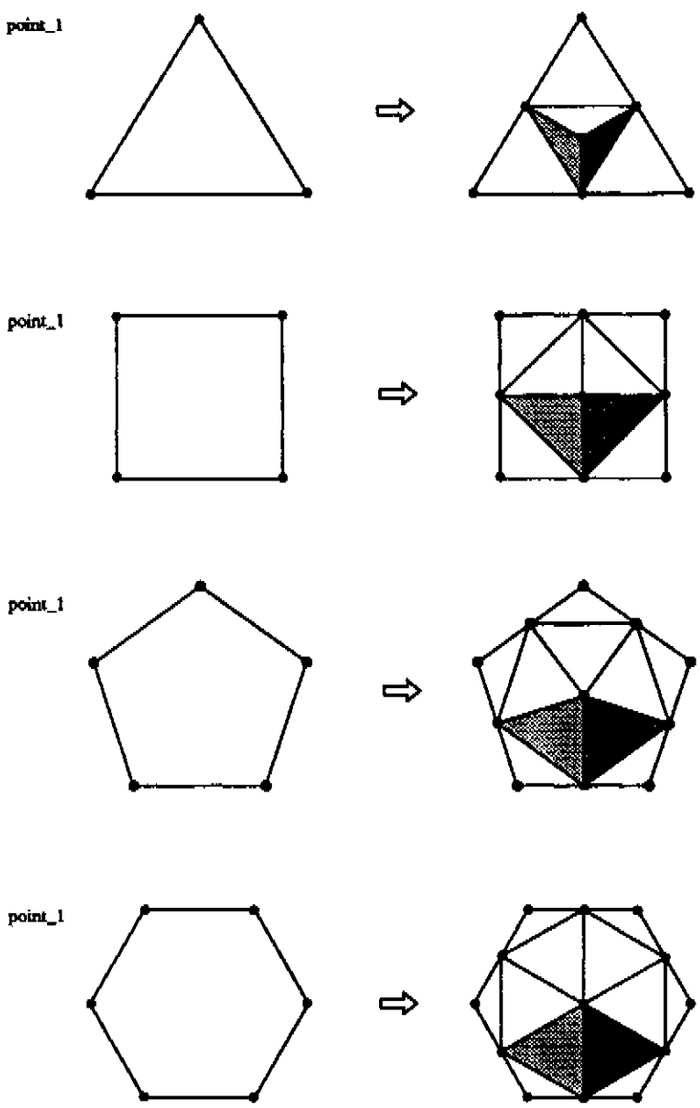

Figure 3.3: The generalized snowflake rule applied to various faces.

3.4 Applying Rules

There are several ways to generate solids using the rules that users have defined. The primary facilities are describe below.

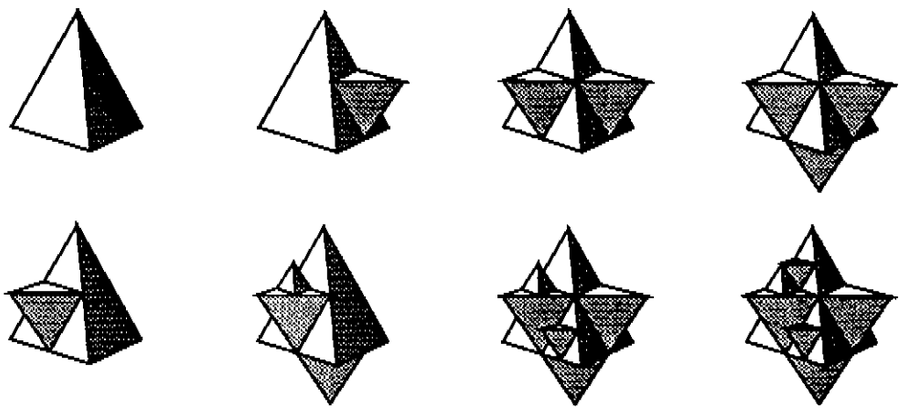

Figure 3.4: A portion of the language of snowflakes.

- Apply a specific rule once and display the results. This is useful primarily for debugging new rules.
- Apply the applicable rules to the current solids. **Genesis** displays the current solids with the matched elements highlighted for a given match, and the user is queried to allow or disallow the application of the rule.
- Apply any applicable rules a given number of times (depth-first) and display the result after each rule application.

`apply`
Apply the rules to the (previously loaded) initial solid(s).

`slowest_apply`
Show everything and check with user at every step - debugging.

`faster_apply`
Ask no questions, but update the display after each rule application.

`fastest_apply`
No questions asked, display only when completely done.

`apply_rule(Rule)`
Apply the given rule once and display results.

`apply_n(N)`
Apply rules `N` times and display the result after each rule application.

3.5 Searching

Genesis provides several ways to search through the space of derivations of a grammar.

- Search depth-first for a member of the language of the grammar (which occurs when the current state equal to `done`).
- Search depth-first for a derivation of the grammar with has a given state `State` as the current state.
- Search randomly for a member of the language of the grammar (with the current state equal to `done`). A given rule is applied if a random number is (between 0 and 1) is less than `Probability`.
- Search randomly for a derivation of a grammar which has a given state `State` as the current state.

The predicates in this section generate derivations from the grammar.

`search`
: Explore the search space of solutions and find a derivation of the grammar (in the `done` state).

`search(State)`
: Explore the search space of solutions and find a derivation of the grammar with `State` as the current state.

`random_search(Probability)`
: Find a random derivation of a grammar. A given rule is applied if a random number is (between 0 and 1) is less than `Probability`.

`random_search(Probability, State)`
: Find a random derivation of a grammar with `State` as the current state. A given rule is applied if a random number is (between 0 and 1) is less than `Probability`.

3.6 Generating Models

A user may mix these various interaction modes. For example, one may apply rules to generate a model, then manually modify the model using solid modeling operations. The new model may then be used as an initial solid for further detailing and development with another rule set. A user may also generate a model with another solid modeler, read it into **Genesis** from a file, and use it as an initial solid in a grammar to generate additional detail, or related solids.

Chapter 4

Matching on Topology

The queries described here match on the topological elements and adjacencies of the generalized split edge data structure [4].

4.1 Matching on Topological Elements

A user may use these relations to locate topological elements of a given type, or to determine the type of a given topological element,

vertex(V)
Find a vertex / Is V a vertex?

edge-half (Eh)
Find an edge half / Is Eh an edge half?

loop(L)
Find a loop / Is L a loop?

face(F)
Find a face / Is F a face?

shell(Sh)
Find a shell / Is Sh a shell?

solid(S)
Find a solid / Is S a solid?

4.2 Matching on Topological Adjacencies

Genesis provides access to a variety of adjacency relations between topological elements. A user may use these facilities to locate a topological element related by a given adjacency relation to a given topological element, or to determine if two given topological elements are related by a given adjacency relation.

`vertex_eh(V,Eh)`
An edge half adjacent to the given vertex / the vertex of a given edge half.

`vertex_l(V,L)`
A vertex of a given loop / The loop of a given vertex.

`vertex_sh(V,Sh)`
A vertex of a given shell / The shell of a given shell.

`vertex_solid(V,S)`
A vertex of a given solid / The solid of a given vertex.

`edgeh_v(Eh,V)`
The vertex of a given edge half / An edge half of a given vertex.

`other_v(Eh,V)`
The vertex of a given edge half / An edge half of a given vertex.

`cw_eh(Eh,Eh2)`
The next edge half clockwise.

`ccw_eh(Eh,Eh2)`
The last edge half clockwise.

`other_eh(Eh,Eh2)`
The opposite half of a given edge.

`other_cw_eh(Eh,Eh2)`
The next edge half (clockwise) of the other edge half.

`other_ccw_eh(Eh,Eh2)`
The last edge half (clockwise) of the other edge half.

`edgeh_l(Eh,L)`
An edge half of a given loop / The loop of a given edge half.

`edgeh_f(Eh,F)`
The face of a given edge half / An edge half of a given face.

otherehjf (Eh,F)
The face of the other edge half from a given edge half / An edge half of a face adjacent to a given face.

edgeh_sh(Eh,Sh)
An edge half of a given shell / The shell of a given edge half.

edgeh_solid(Eh,S)
An edge half of a given solid / The solid of a given edge half.

loop.v(L,V)
The first vertex of a given loop.

loop_eh(L,Eh)
The first edge half of a given loop.

loop_f (L,F)
The face of a given loop / A loop of a given face.

loop_sh(L,Sh)
A loop of a given shell / The shell of a given loop.

loop_solid(L,S)
A loop of a given solid / The solid of a given loop.

face_eh(F,Eh)
The first edge half of a given face.

faceJ.(F,L)
The first loop of a given face.

face-f(F,F2)
A face adjacent to the given face.

face_sh(F,Sh)
A face of a given shell / The shell of a given face.

face_solid(F,S)
A face of a given solid / The solid of a given face.

shell.vCSh.V)
A vertex of the given shell.

shell_eh(Sh,Eh)
A edge half of the given shell.

shell_l(Sh,L)
A loop of the given shell.

shell_f(Sh,F)
A face of the given shell.

shell_solid(Sh,S)
A shell of a given solid / The solid of a given shell.

solid_sh(S,Sh)
The first shell of a given solid.

4.3 Counting Topological Elements

A user may wish to know the number of elements of a given type. For example, find the number of faces that are represented, the number of vertices of a solid, or the number of edges adjacent to a vertex. The **count_element** relation provides this utility.

count_elements(Id, Type, Use, Count)
Count the number of elements of the given type and use within / associated with a given topological element, and return the number as the value of `Count`. To find the number of elements of all the existing solids, use `Id` = 0.

Chapter 5

Matching on Geometry

This chapter describes the queries on the geometric information associated with various topological elements.

5.1 Primitive Matching

`v_coord(V,[X,Y,Z])`
Find the coordinate of the given vertex / Find a vertex with the given coordinates.

`face_normal(F,[A,B,C])`
Find the normal of the given face (extracted from the face equation).

`face_equation(F,[A,B,C,D])`
Find the equation of the given face.

`element_bbox(Id,[X1,Y1,Z1],[X2,Y2,Z2])`
Find the bounding box of the given element.

5.2 Composite Conditions

`distance_v(V1,V2,D)`
The Euclidean distance between two vertices.

`distance_eh(Eh1,Eh2,Length)`
The distance between the vertices of two distinct edge-halves.

`eh_length(Eh,Length)`
The length of an edge half.

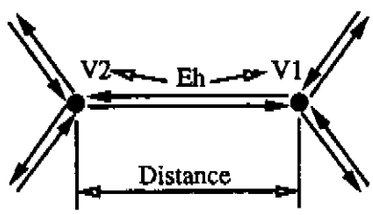

Figure 5.1: The length of an edge-half.

`eh_midpoint(Eh,Midpoint)`
The coordinates of the midpoint of an edge half.

`colinear_v([V1,V2,V3 | Rest])`
Are these vertices colinear.

`vertex_on_face(V,F)`
Does the given vertex lie on the given face.

`point_on_face(C,F)`
Does the given point lie on the given face.

`point_on_plane([X,Y,Z],[A,B,C,D])`
Does the given point lie on the given plane.

`cw_non_colinear_eh(Eh,CwNCEh)`
Finds the clockwise non-colinear edge half from the current one.

`adjacent_solids(S1, S2)`
Determine whether two solids are adjacent.

5.3 Integral Properties

`face_center(F,Center)`
The center of a face, calculated as an average of the coordinates of the vertices.

`element_area(Id,Area)`
Compute the surface area of a given element.

`solid_centroid(S,[X,Y,Z])`
Compute the center of the given solid.

`solid_volume(S,Volume)`
Compute the volume of the given solid.

`solid_moment_of_inertia_point(S,[X,Y,Z],Moment)`
Compute a moment of inertia about the given point.

`solid_moment_of_inertia_line(S,[X1,Y1,Z1],[X2,Y2,Z2],Moment)`
Compute a moment of inertia about the given line.

`solid_moment_of_inertia_plane(S,[A,B,C,D],Moment)`
Compute a moment of inertia about the given plane.

5.4 Orientations

A number of queries are provide by **Genesis** to locate faces that are oriented along the coordinate axes. The orientations are indicated by `front`, `back`, `left`, `right`, `top`, and `bottom`, with the idea that the viewer is standing within the solid and looking down the negative y-axis (and up is in the direction of the positive z-axis).

```
orientation(top,[0,0,1])
orientation(bottom,[0,0,-1])
orientation(front,[0,-1,0])
orientation(back,[0,1,0])
orientation(left,[1,0,0])
orientation(right,[-1,0,0])
```
The orthogonal vectors for determining orientations.

```
top(F)
top(F,S)
```
Determines if the face is a top (upward oriented) face.

```
bottom(F)
bottom(F,S)
```
Determines if the face is a bottom (downward oriented) face.

```
front(F)
front(F,S)
```
Determines if the face is a front face.

```
back(F)
back(F,S)
```
Determines if the face is a back face.

```
left(F)
left(F,S)
```
Determines if the face is a left face.

```
right(F)
right(F,S)
```
Determines if the face is a right face.

```
side(F)
```

`side(F,S)`
Determines if the face is a side face.

`side_not_front(F)`
`side_not_front(F,S)`
Determines if the face is a side face, but not a front face.

Chapter 6

Creating and Modifying Solids

Genesis provides a multitude of operations to create and modify solid models. This chapter describes operations that are used to create primitive solids and laminae, and modify, transform, and combine existing solids, The Euler operations are also available in **Genesis** , and are described in Chapter 7.

6.1 Constructing Primitive Solids

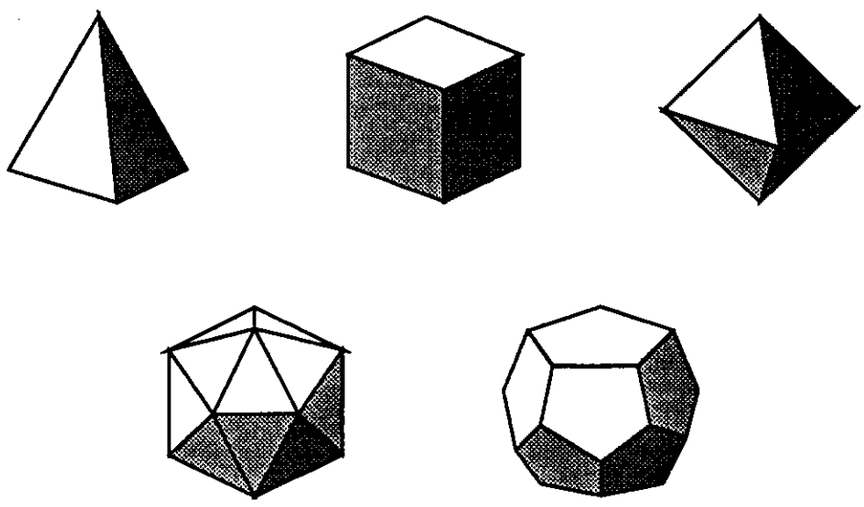

Figure 6.1: The Platonic solids.

`make_tetrahedron(Radius,NewS)`

Create a tetrahedron with a given radius, and the center of the bottom face at the origin.

`make_orthogonal_cuboid([Xlo,Ylo,Zlo],[Xhi,Yhi,Zhi],NewS)`
Create an orthogonal solid at [Xlo,Ylo,Zlo] with its distant corner at [Xhi,Yhi,Zhi].

`make_octahedron(Radius,NewS)`
Create a octahedron with a given radius, and its center located at the origin.

`make_icosahedron(Radius,NewS)`
Create a icosahedron with a given radius, and its center located at the origin.

`make_dodecahedron(Radius,NewS)`
Create a dodecahedron with a given radius, and its center located at the origin.

`make_lamina(NumV,Radius,NewS)`
Create a lamina with a given number of vertices and radius, oriented with the barycenter at the origin, and lying in the xy-plane.

`make_random_lamina(NumV,MaxRadius,MinRadius,NewS)`
Create a random-shaped lamina with the given number of vertices. The vertices will be lying in the xy-plane, oriented about the origin, and between MinRadius and MaxRadius in distance from the origin.

`make_spur_lamina(NumKnobs, Radius, Width, Solid)`
Create a lamina in the shape of a spur, with a given number of knobs and radius, lying in the xy-plane, and oriented with the barycenter at the origin.

`make_face_lamina(FaceEh,TopEh)`
Create a (solid) lamina copy of a given face, and return the new edge-half "TopEh" corresponding with the given edge-half "FaceEh". The face of the edge-half "TopEh" will have the same normal as the face of "FaceEh".

6.2 Modifying Solids

`stack_solid_height(F,Height,TopFace)`
Locates a solid on the top of the existing face. The solid is extruded in the shape of the original face, to a height of "Height" in the direction of the face's normal.

`stack_solid(StartEh,Matrix,TopEh)`

`extrude_face_height(F,Height,NewF)`

`extrude_face_eh(StartEh,Matrix,TopEh)`

Figure 6.2: The extrudeJace operator.

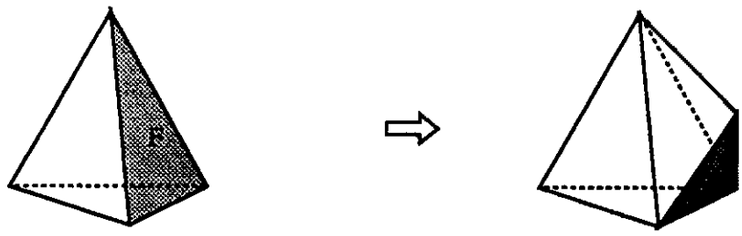

Figure 6.3: The point-face operator.

extrude-face **Jionplanar**_ch(StartEh,Matrix ,TopEh)

point-face (F, Height)

Extrudes the existing face of a solid (in the shape of the original face) in the direction of the normal and to a height of "Height".

stack-pointed_solid(F,Height,NewS)

Locates a solid on the top of the existing face. The solid has a bottom face matching the original face and extends to a point to a height of "Height" in the direction of the face's normal.

pull_face (F .Height)

Pulls the existing face of a solid (in the shape of the original face) in the direction of the normal and to a height of "Height".

pull_face.vector(F, Vector)

pull-face_matrix(F, Matrix)

split-**Solid**(F,NewS)

`split_solid(F,Height,NewS)`
Splits the existing solid into two solids, with the new solid "NewS" as an extruded version of the original face with height of "Height".

6.3 Moving Vertices

The coordinates of a vertex can be changed to locally transform the surface of a solid. This section describes various operations that modify the coordinates of a vertex.

`set_vertex(V,[X,Y,Z])`
Set the coordinates of the given vertex to `[X, Y, Z]`.

`move_vertex(V,[X,Y,Z])`
Move a vertex by the amount of the given vector.

`transform_vertex(V,Matrix)`
Transforms the coordinates of a vertex with the given matrix.

`move_vertex_random(V,MaxRadius)`
Move the given vertex a random amount in any direction a random amount less than MaxRadius.

`move_vertex_random_xy(V,MaxRadius)`
Move the given vertex in the x-y plane a random amount less than MaxRadius.

`move_vertex_random_eh(Eh)`
Move the vertex of the given edgehalf a random amount in any direction a random amount less than one third of the length of the edge.

6.4 Transforming Solids

`transform_solid(S,Matrix)`
Transform the coordinates of all the vertices of a solid with the given matrix.

`translate_solid(S,[X,Y,Z])`
Translate all the vertices of a solid.

`rotate_solid(S,[Phi,Theta,Psi])`
Rotates all the vertices of a solid.

`scale_solid(S,[Sx,Sy,Sz])`
Scales all the vertices of a solid.

`transform_vertices(Matrix)`
Transforms all vertices of all solids with the given transformation matrix.

`translate_vertices([Tx,Ty,Tz])`
Translates all vertices by the given vector.

`rotate_vertices([Phi,Theta,Psi])`
Rotates all vertices by Phi radians about the z-axis, then Theta radians about the x-axis, followed by Psi radians about the z-axis.

`scale_vertices([Sx,Sy,Sz])`
Scales all vertices by the given scaling.

6.5 Unary and Boolean Operations

`unary_union(S,NewS)`
Copies the solid, constructs the unary union of a given solid and returns the resulting solid as "NewSolid".

`unary_intersection(S,NewS)`
Copies the solid, constructs the intersection of a given solid and returns the resulting solid as "NewSolid".

`unary(N,S,NewS)`
Copies the solid, constructs the generalized unary intersection and returns the resulting solid as "NewSolid". "N" is an integer indicating the enclosing number for classifying the shells, where 1 corresponds to unary union and 2 corresponds to unary intersection.

`union(S1,S2,NewS)`
Computes the boolean union of two solids and returns the resulting solid as "NewSolid".

`intersection(S1,S2,NewS)`
Computes the boolean intersection of two solids and returns the resulting solid as "NewSolid".

`difference(S1,S2,NewS)`
Computes the boolean difference of two solids and returns the resulting solid as "NewSolid".

`copy_solid(S,NewS)`
duplicates a solid and all of its elements and geometry. "NewS" is the identifier of the copy of solid "S".

Chapter 7

Euler Operations

Euler operations are a set of operators that manipulate graph representations of the topological elements and adjacencies of the boundary of solids. They modify a boundary representation by adding and removing topological elements, while maintaining consistent topological adjacency relations. Typical Euler operations split edges (Figure 7.1), and split faces (Figure 7.2).

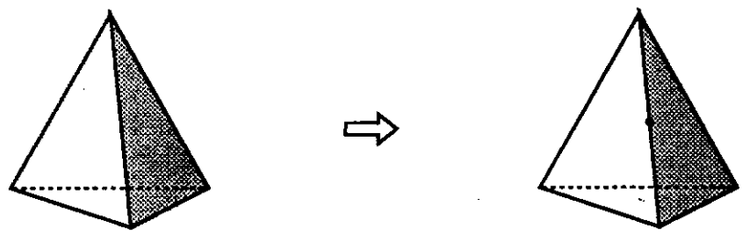

Figure 7.1: Splitting an edge.

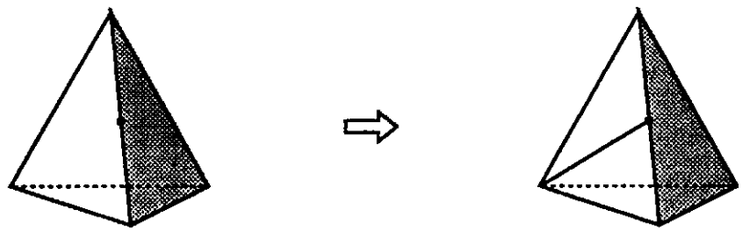

Figure 7.2: Splitting a face.

. The Euler operators in this manual follow the naming convention originally introduced by Baumgart [1]. The names describe the effect the operators have on the creation and removal, of topological elements as well as the genus of the solid. An m stands for "make" or create, and k

stands for "kill" or remove. Each of these is followed by the letters signifying the types of topological elements created or removed. `v`, `e`, `l`, `f`, `s` and `g` stand for vertex, edge, loop, face, shell, and genus. Thus, **mev** stands for "make edge, vertex", and **keml** stands for "kill edge, make loop". A few operators, such as **glue** and **esplit**, have more descriptive names.

7.1 Manifold Euler Operations

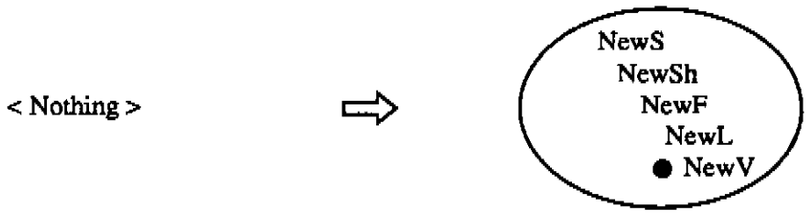

Figure 7.3: The mssflv operator

mssflv(NewS,NewSh,NewF,NewL,NewV)
"make solid, shell, face, loop, vertex" creates a new solid topology with a single shell (manifold surface) with one face (containing a single loop) and one vertex. No edges are created. This is the minimal topology needed to represent a shell, but is not sufficient to represent a polyhedron. The identifiers of the new elements are returned as the values of "NewS", "NewSh", "NewF", "NewL", and "NewV".

merge_solids(S1,S2)
merges the shells of two solids. The shells of "S2" are added to the solid "S1", and the record of "S2" is removed.

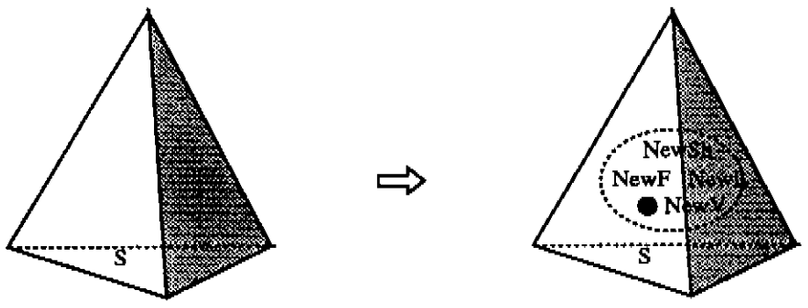

Figure 7.4: The msflv operator

msflv(S,NewSh,NewF,NewL,NewV)
"make shell, face, loop, vertex" adds a new shell (manifold surface) to an existing solid. The new shell "NewSh" is added to the given solid "S". The new shell, face, loop and vertex are

returned as the values of "NewSh", "NewF", "NewL", and "NewV". As in **msflv**, the new shell consists of a single face, loop and vertex.

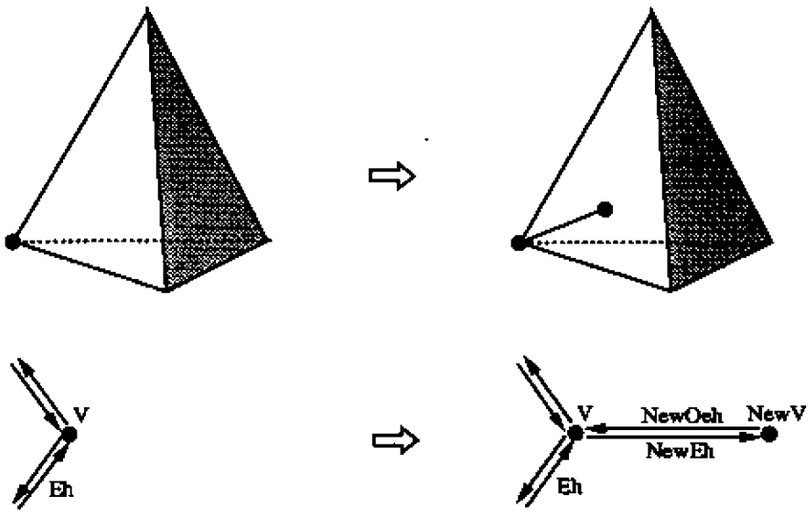

Figure 7.5: The mev operator

mev(V,Eh,NewV,NewEh)

"make edge, vertex" creates a new edge and vertex as a strut edge in a face (see Figure 7.5). For a shell with no edges, **mev** creates the first edge in the shell's single face and loop. Both sides of the new edge will be adjacent to the same face.

"V" is the existing vertex which will be at one end of the new edge. "Eh" is the edge-half that is counterclockwise from the edge-half of vertex "V". If there are no edges adjacent to vertex "V", "Eh" will be null.

When **mev** is complete:

- the new edge-half "NewEh" will be clockwise to the given edge-half "Eh";
- the new vertex "NewV" will be the vertex of the other_eh and the cw_eh of the new edge-half "NewEh".

esplit(Eh,NewEh,NewV)

"edge split" splits a given edge "Eh", creating a new edge and vertex. **esplit** is a form of **mev**.

When **esplit** is complete:

- the new edge-half "NewEh" will be clockwise to the given edge-half "Eh";
- the second new edge-half will be the other_eh of the given edge-half "Eh";

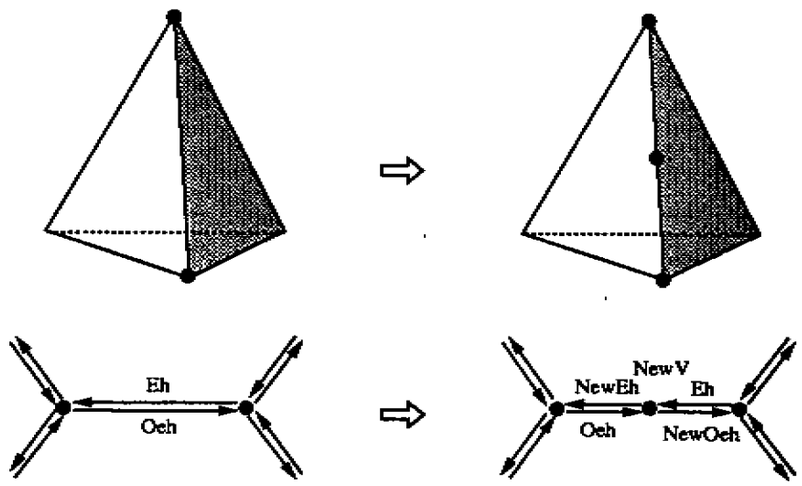

Figure 7.6: The esplit operator

- the new vertex "NewV" will belong to both the two new edge-halves.

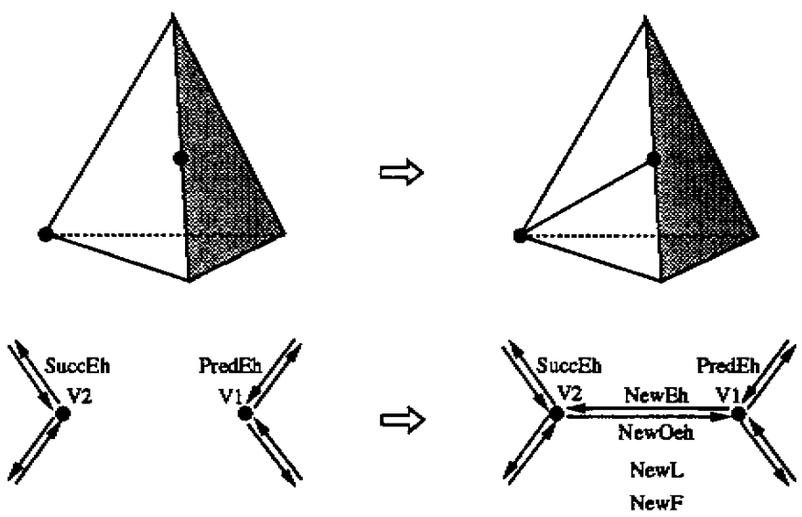

Figure 7.7: The mefl operator

`mefl(V1,PredEh,V2,SuccEh,NewEh,NewL,NewF)`

"make edge, face, loop" splits a face and loop, creating a new face, loop, and an edge separating the new and old faces.

The vertex "V1" belongs to the clockwise edge-half of "PredEh". If the vertex "V1" has

no edge-halves, "PredEh" will be null. The vertex "V2" belongs to "SuccEh". If the vertex "V2" has no edge-halves, "SuccEh" will be null.

When **mefl** is complete:

- the new edge-half "NewEh" will connect the vertices "V1" and "V2", and will be clockwise to "PredEh", and counterclockwise to "SuccEh";
- the second new edge-half will be the other_eh of "NewEh";
- the loop of the other_eh of "NewEh" will be the new loop "NewL";
- the face of the new loop "NewL" will be the new face "NewF".

If the vertices are the same ("V1" = "V2"), **mefl** creates an edge with the same vertex at each end.

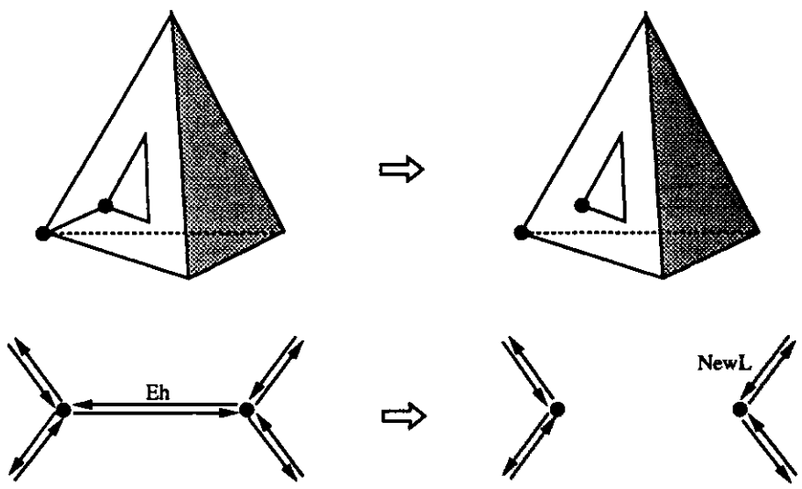

Figure 7.8: The keml operator

`keml(Eh,NewL)`

"kill edge, make loop" removes an edge on a face splitting one loop into two separate loops, creating a new loop of edges. If the edge is the only one in the loop, the result will be two loops, each containing a single vertex. If the edge is a strut edge, one of the loops will contain a single vertex.

The edge-halves "Eh" and its other_eh will be removed. The counterclockwise edge-half of "Eh" (or the vertex of "Eh") will be contained in the new loop "NewL".

`glue(F1,F2)`

"kill faces, loops" glues two faces together. The faces must have the same number of loops, and the corresponding loops must have the same number of edges and vertices.

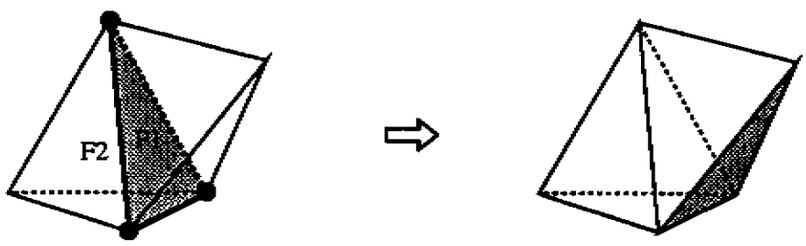

Figure 7.9: The glue operator

7.2 Inverse Manifold Euler Operations

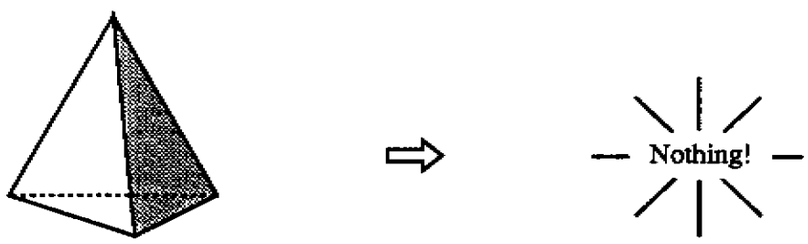

Figure 7.10: The kssflevs operator

`kssflevs(S)`
"kill solid, shells, faces, loops, edges, vertices" deletes the existing solid "S" and all of its elements.

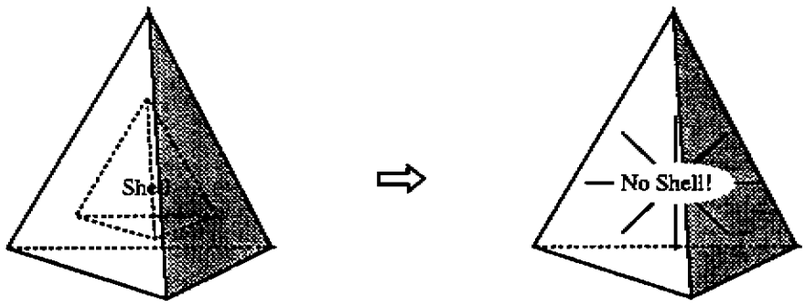

Figure 7.11: The ksflevs operator

`ksflevs(Sh)`
"kill shell, faces, loops, edges, vertices" Remove the shell "Sh" and all its elements from an existing solid.

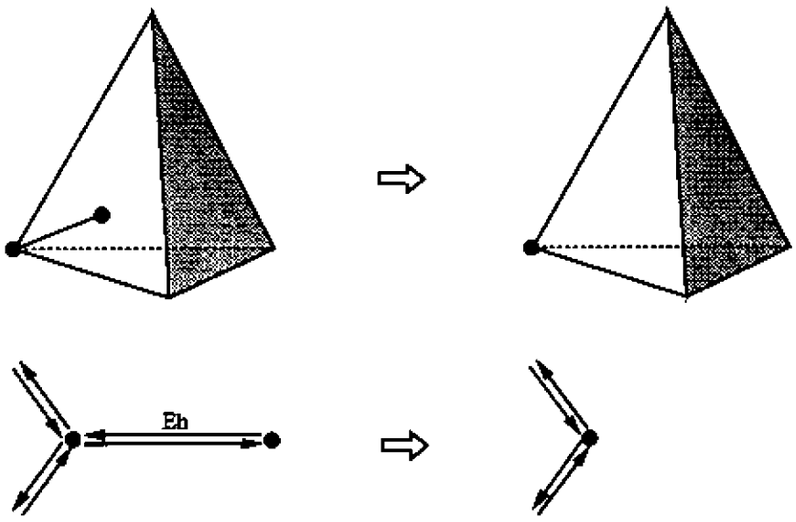

Figure 7.12: The kev operator

`kev(Eh)`
"kill edge, vertex" removes a "strut" edge from a face, and the vertex belonging to it.

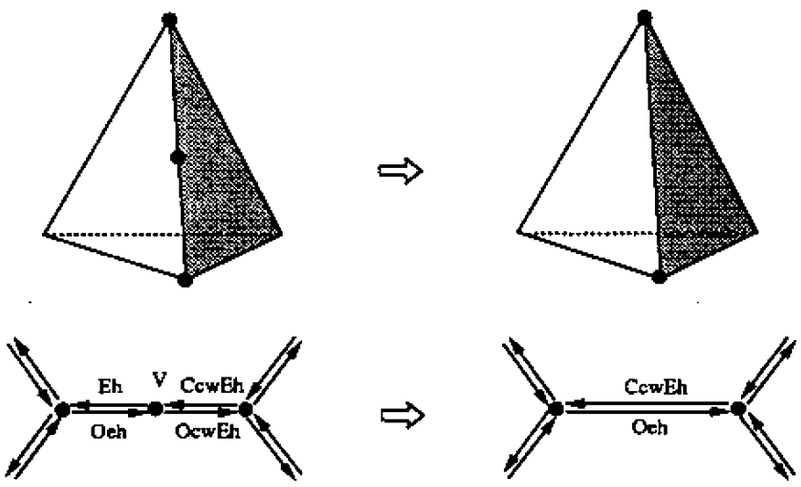

Figure 7.13: The ejoin operator

`ejoin(Eh)`
"edge split" joins an edge, creating a new edge and vertex. `ejoin` is a form of **kev**.
`ejoin` applies when the vertex of "Eh" is adjacent to only two edge-halves, "Eh" and the

cw.eh of the other .eh of "Eh". When ejoin is complete, "Eh" and the cw_eh of the other.eh of "Eh" will be removed, as well as the vertex adjacent to these two edge-halves.

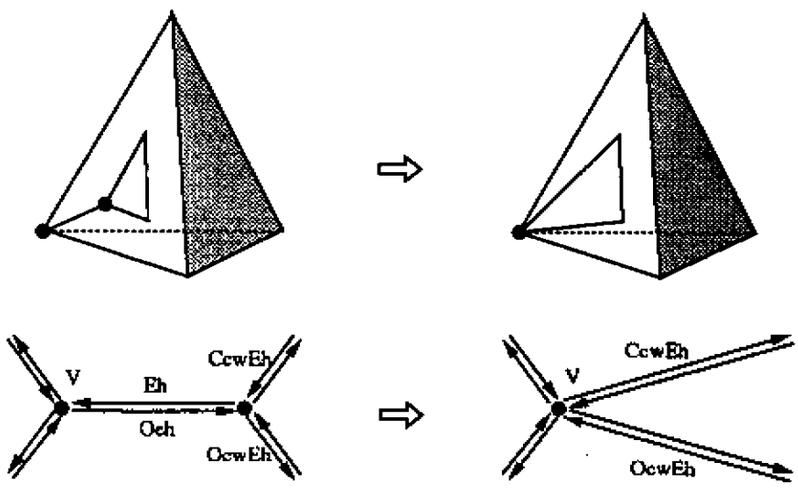

Figure 7.14: The esqueeze operator

esqueeze(Eh)

"edge squeeze" joins two vertices, removing the edge between, **esqueeze** is a form of kev.

"Eh" and its other .eh will be removed, as well as the vertex belonging to "Eh". The other edge-halves adjacent to the vertex of "Eh" will now be adjacent to the vertex that belonged to the other-eh of "Eh".

kef1(Eh)

"kill edge, face, loop" joins a face and loop, removing the edge separating the two faces, and the face and loop of that edge.

mekl(VI,PredEh,V2,SuccEh,NewEh)

"make edge, kill loop" adds an edge between two vertices, VI and V2, on different loops of a face.

The vertex "VI" belongs to the clockwise edge-half of "PredEh". If the vertex "VI" has no edge-halves, "PredEh" will be null. The vertex "V2" belongs to "SuccEh". If the vertex "V2" has no edge-halves, "SuccEh" will be null.

When mekl is complete:

- the new edge-half "NewEh" will connect the vertices "VI" and "V2", and will be clockwise to "PredEh", and counterclockwise to "SuccEh";
- the second new edge-half will be the other_eh of "NewEh";

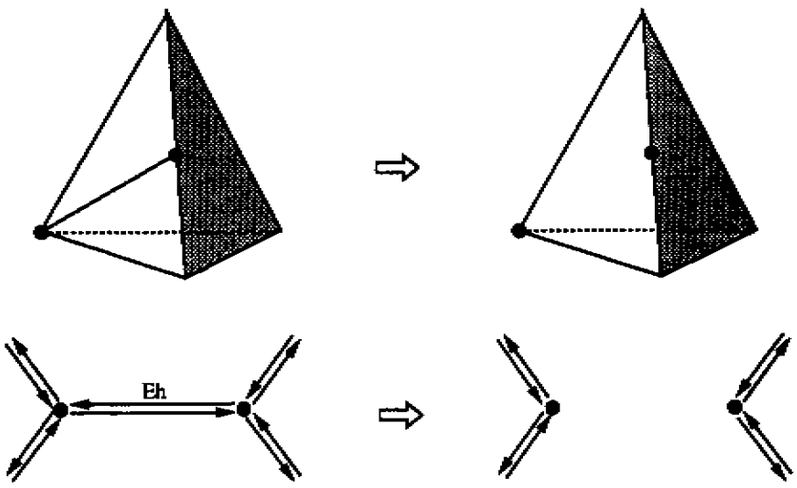

Figure 7.15: The kefl operator

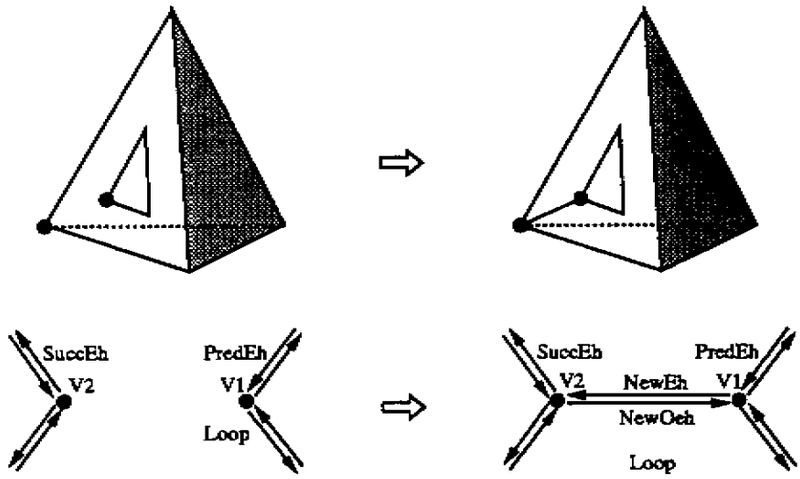

Figure 7.16: The mekl operator

`unglue(CycleOfEhs,NewF1,NewF2)`

"make faces, loops" unglues a cycle of edges, creating two new faces and loops. `CycleOfEhs` is a list of the pairs of edge-halves to be separated.

Using Manifold Euler Operations

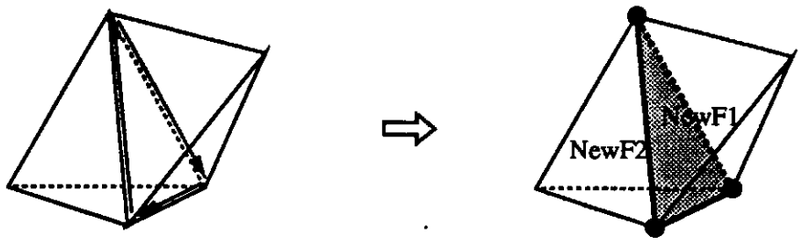

Figure 7.17: The unglue operator

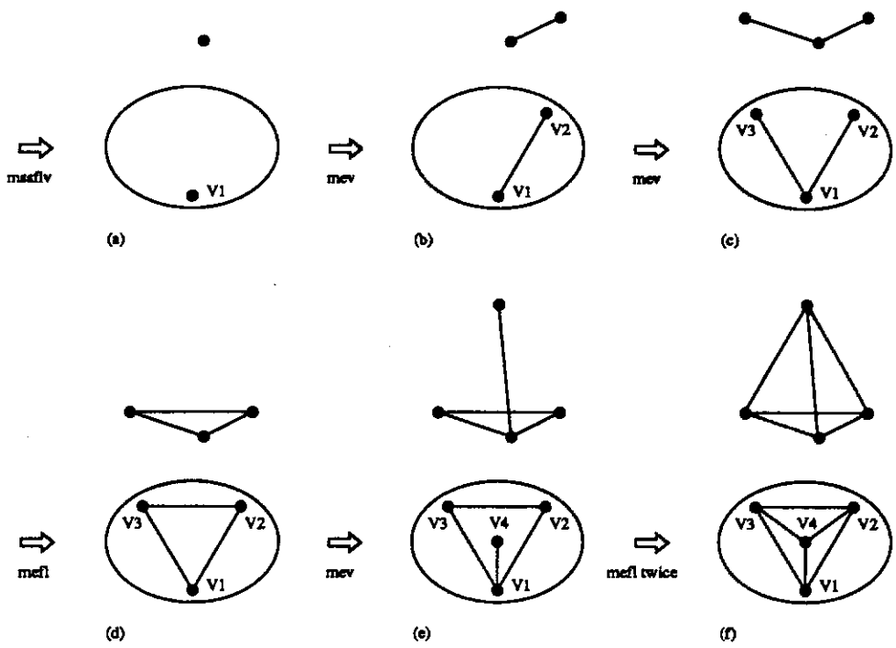

Figure 7.18: Building a tetrahedron using Euler operations

We can illustrate the use of Euler operators with a simple example, the construction of a tetrahedron. The steps of the construction are presented Figure 7.18, with both plane models and three-dimensional models.

The construction begins with a minimal topology, created with **mssflv** (a). The first and second edges are then added with two applications of **mev** (b and c). At this point, the model has a single face. By creating an edge between $V2$ and $V3$, we split the face using **mefl** (d). This gives us a triangular lamina. $V4$ and the strut edge between $V1$ and $V4$ are created with **mev** (e). Two more applications of **mefl** create the remaining two faces and two vertices, completing the topology of the tetrahedron (f). In order to complete the description of the tetrahedron, we need to assign

coordinates to the vertices.

7.3 Nonmanifold Euler Operations

Three nonmanifold Euler operations (and their inverses) construct explicit representations of nonmanifold solids using the generalized split-edge data structure. These maintain the additional nonmanifold adjacency relationships: multiple loops of faces about a vertex; multiple pairs of faces about an edge; and and multiple connected uses (components) of a shell.

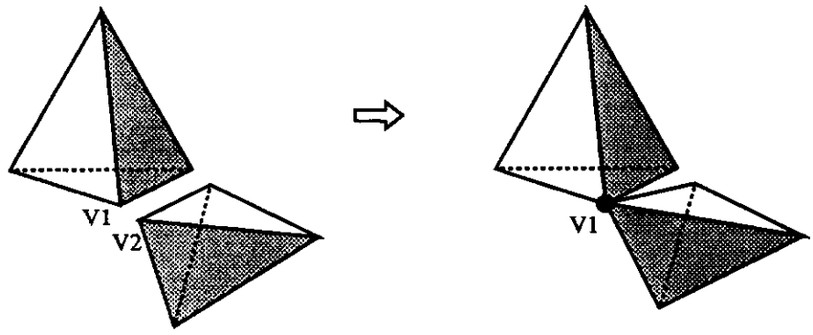

Figure 7.19: The ksv operator

ksv(V1,V2)

"kill shell, vertex" merges two vertices existing on different shells, and merges their shells.

"V1" and "V2" are the two vertices to be merged. When **ksv** is complete, "V1" and "V2" will be uses of the same nonmanifold vertex, and will be on different uses of the same shell.

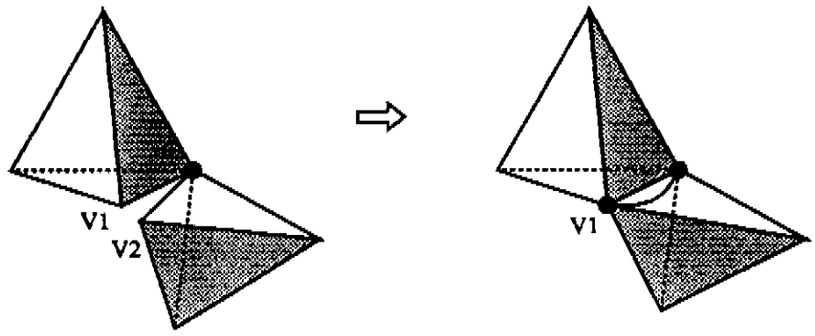

Figure 7.20: The kvmg operator

`kvmg(V1,V2)`

"kill vertex, make genus" merges two vertices existing on the same shell. This creates a (nonmanifold) handle and increases the genus by one.

"V1" and "V2" are the two vertices to be merged. When **kvmg** is complete, "V1" and "V2" will be uses of the same nonmanifold vertex, and will be on the same shell and shell use.

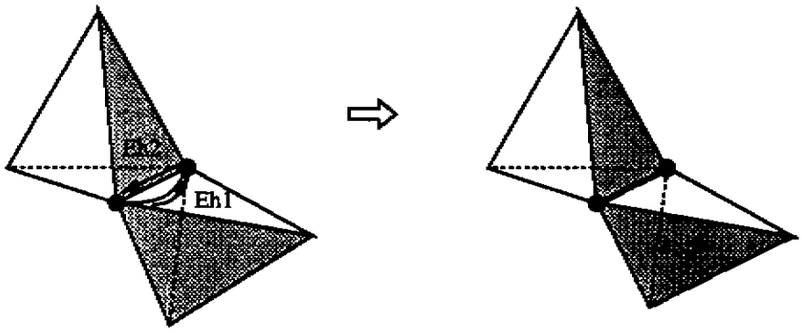

Figure 7.21: The keg operator

`keg(Eh1,Eh2)`

"kill edge, genus" merges two edges that share their two vertices. This seals a (nonmanifold) handle and decreases the genus by one, or creates a nonmanifold chamber.

"Eh1" and "Eh2" are halves of the two edges to be merged. When **keg** is complete, "Eh1" and "Eh2" will be halves of the same nonmanifold edge.

Inverse Nonmanifold Euler Operations

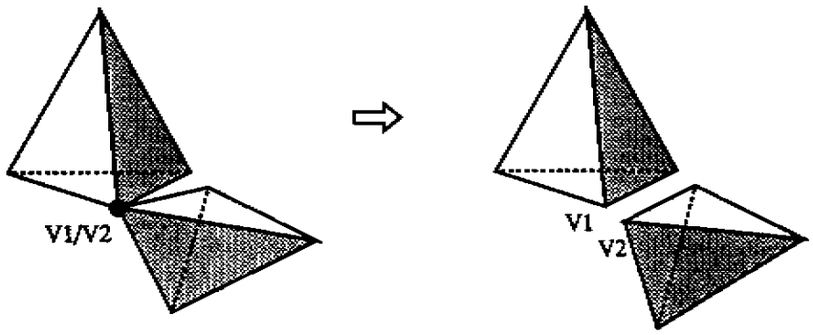

Figure 7.22: The msv operator

`msv(V1,V2)`

"make shell, vertex" splits two vertex-uses of a vertex. The two vertex-uses are on the same shell, but different shell-uses, and creates a shell.

"V1" and "V2" are different uses of the same vertex that are to be separated. "V1" and "V2" must be the only vertex uses joining their shell uses. When `msv` is complete, "V2" will be removed from the vertex of "V1" and will form a separate vertex, and will be on a separate shell.

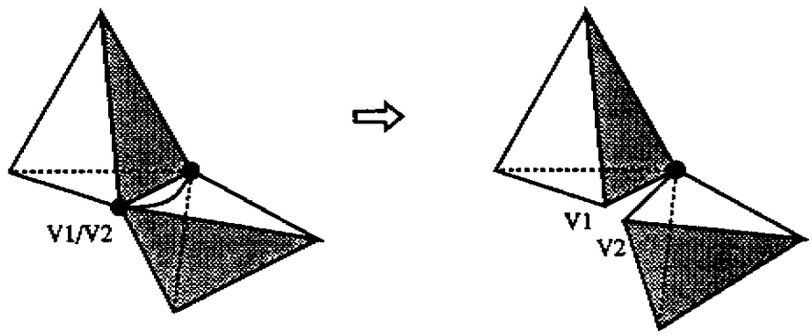

Figure 7.23: The mvkg operator

`mvkg(V1,V2)`

"make vertex, kill genus" splits two vertex-uses of a vertex that exist on the same shell and shell-use. This removes a (nonmanifold) handle and decreases the genus by one.

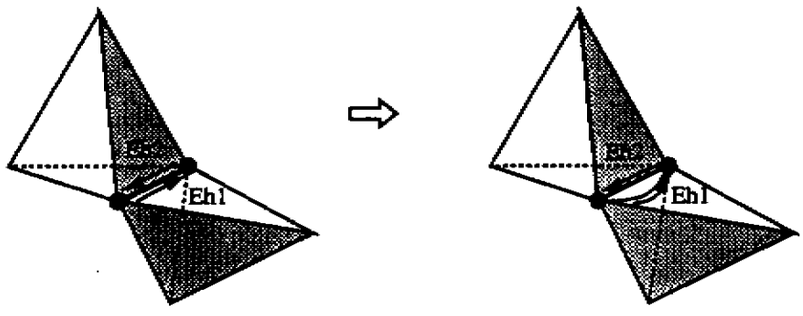

Figure 7.24: The meg operator

`meg(Eh1,Eh2)`

"make edge, genus" splits two edge-uses of an edge. This creates a (nonmanifold) handle and increases the genus by one, or removes a nonmanifold chamber.

Using Nonmanifold Euler Operations

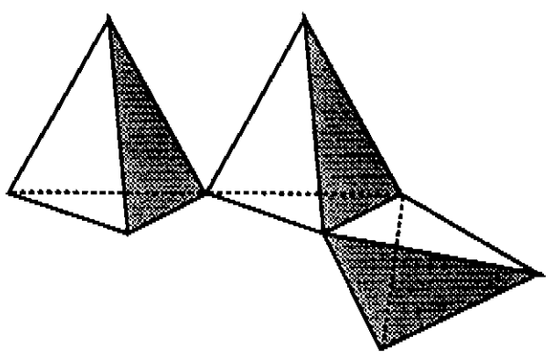

Figure 7.25: A nonmanifold solid.

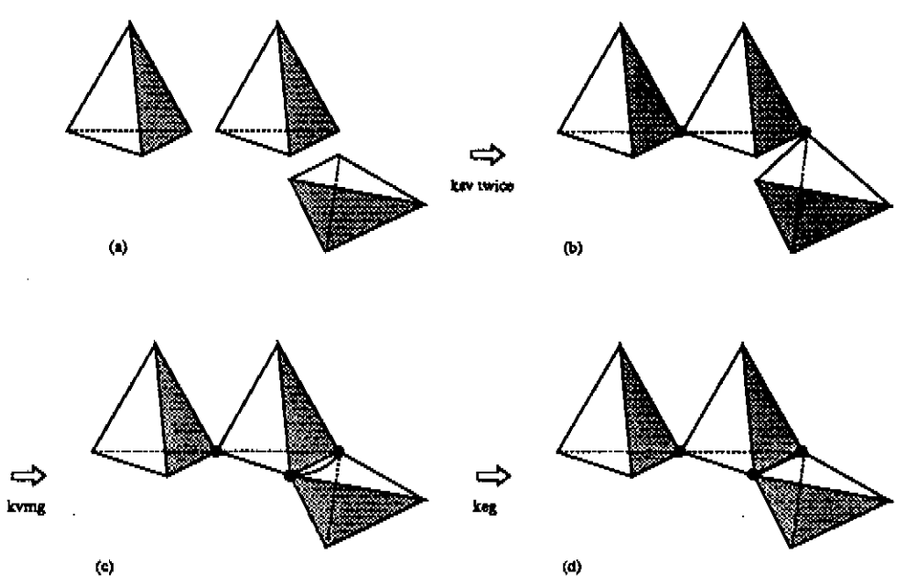

Figure 7.26: Building a solid using nonmanifold Euler operations

We can illustrate the use of nonmanifold Euler operators with another example, constructing the solid in Figure 7.25. The steps of the construction are presented Figure 7.26.

The construction begins with three tetrahedra (a). The three are connected with two applications of **ksv** (b). Two vertices are then joined with one application of **kvmg**, making a nonmanifold handle (c). One application of **keg** joins the two edges and removes the nonmanifold handle (d), and completes the solid.

Chapter 8

Labels and States

Labeling provides a mechanism for associating non-geometric information to topological elements. This allows material properties and functions to be associated to solids, or attaching markers that permit or restrict rule application to an element.

The current state of a design may be indicated with the **state** mechanism. The state controls the application of rules, and determines when a derivation of a grammar is complete.

8.1 Querying and Modifying Labels

Labels may be queried, created, and removed using the following facilities.

`label(Id,Attribute,Value)`
: Finds a label in the label database.

`make_label(Id,Attribute,Value)`
: Add a label to the label database.

`kill_label(Id,Attribute,Value)`
: Remove a label from the label database.

8.2 Querying and Modifying the State

The current state is queried with the **state** relation, and modified using the **set_state** operation.

`state(X)`
: Find the current state / Is this the current state.

`set_state(State)`
Set the current state to `State`.

Chapter 9

Graphics

A graphical user interface allows the user to move the position and direction of the camera and viewpoint, modify the brightness, color, and position of directional lights, modify the brightness and color of the ambient light, and modify the amount of specular reflection and color of specular highlights. The user interface also controls the culling of back-facing polygons, depth-cueing, the display of objects with wireframe or shaded surface rendering, the default color of objects, and the background color. The graphical user interface is constructed with widgets in the X window system.

The graphics routines are written in C. These routines access and maintain additional structures for polygon display. Hidden surface removal is accomplished using a hardware z-buffer, and HP Starbase graphics routines.

9.1 Graphical Display

`display_list(List)`
: Update the display, with the elements in `List` highlighted, and pass control to the graphics.

`display`
: Update the display and pass control to the graphics.

`redisplay`
: Display and pass control to the graphics without updating the graphics.

`display_and_continue`
: Update the display and continue, without passing control to the graphics.

9.2 Display of Solids

The user may control the color and transparency of the elements of solids represented in **Genesis** . These operations are listed below.

`set_color(Id,[Red,Green,Blue])`
: Set the color of the given element to the RGB values (from 0 to 1).

`set_transparent(Id,Transparency)`
: Set the transparency of the given element to the given value (from 0 to 1, where 0 is completely transparent and 1 is completely solid).

9.3 Highlighting

Highlighting is used to show the user where a rule may be applied during rule application and searching. The default highlight color is red, but the user may change the highlight color at any time. This is useful when there is little contrast between the color of the displayed solids and the current highlight color.

`set_highlight_color([Red,Green,Blue])`
: This predicate sets the highlight color to the given RGB value.

Chapter 10

Input / Output

Genesis provides facilities for textual display of the topological elements, adjacencies, and associated geometry of its internal representation, as well as operations for storing and retrieving these description to / from files. This chapter describes these operations.

10.1 Display and Debugging

print-vertex (V)

Print the entry for the (unique) vertex, V, in Genesis format:

vertex V FirstLoop FirstEh NextVertexUse X Y Z

print.vertices

Print all vertices.

print.eh (Eh)

Print the edge half, Eh, in the following format:

edge Eh Loop CwEh CcwEh OtherEh Vertex

print.ehs

Print all edge halves in Genesis format.

print-loop (L)

Print the entry for the (unique) loop, L, in Genesis format:

loop L Face FirstEh FirstVertex

print-loops

Print all loops in Genesis format.

`print_face(F)`
Print the entry for the (unique) face, F, in **Genesis** format:

```
face F Shell FirstLoop
```

`print_faces`
Print all faces in **Genesis** format.

`print_shell(Sh)`
Print the entry for the (unique) shell, Sh, in **Genesis** format:

```
shell Sh Solid OutsideSh ConnectedSh InsideSh FirstFace
```

`print_shells`
Print all shells in **Genesis** format.

`print_solid(S)`
Print the entry for the (unique) solid, S, in **Genesis** format:

```
solid S FirstShell
```

`print_solids`
Print all solids in **Genesis** format.

`print_solid_group(G)`
Print the entry for the (unique) group, G, in **Genesis** format:

```
solid_group G Parent
```

`print_group_group(G)`
Print the entry for the (unique) group, G, in **Genesis** format:

```
group_group G Parent
```

`print_groups`
Print all groups in **Genesis** format.

`print_label(Id,Value)`
Prints only one (possibly of many) label for the (unique) element id, Id, given.

```
label Id Value
```

`print_label(Id,Attribute,Value)`
Prints only one (possibly of many) label for the (unique) element id, Id, given.

```
label Id Attribute Value
```

print_labels
Print all labels in **Genesis** format.

print_state
Prints the current state in **Genesis** format:

```
state CurrentState
```

print_elements
Print all the elements in **Genesis** format.

10.2 File Input/Output

read_solids_from_file(File,NewS)
Read in a file of solids in **Genesis** format. The solid returned is the oldest solid created from the file.

write_solids_to_file(File)
This is the top level predicate for writing all the elements of all the solids to a file in **Genesis** format.

write_solids_to_noodles_file(File)
This is the top level predicate for writing all the elements of all the solids to a file in Noodles binary format.

10.3 Saving and Restoring Bitmaps

bitmap_to_file(File)
Save the image in the frame buffer to the given file.

file_to_bitmap(File)
Load the stored image into the frame buffer.

inquire_file(File)
Inquire about the information stored in the bitmap file.

set_colormap(Boolean)
Set the color map switch to 1 or 0 in when saving or restoring a bitmap.

Chapter 11

Mathematical Predicates

The IBM CLP(7£) Interpreter provides a number of mathematical relations and functions, described in the IBM CLP(7£) Reference Manual [2]. Genesis provides additional relations, primarily for vector and matrix operations. This chapter describes this additional functionality.

11.1 Basic Functions

sin(X,SinX)
: Given X, computes sin(X).

cos(X,CosX)
: Given X, computes cos(X).

tan(X,TanX)
: Given X, computes tan(X).

atan2(X,Y,ATanX)
: Given X and Y, computes arctan(^).

nearl(X,Y)

: Determines if two scalar values are equal within the tolerance of the machine.

trunc(X,Y)
: Given a scalar, X, truncates X and returns as Y.

11.2 Matrix and Vector Operations

All vectors are 3x1 (1x3 for row vectors), and all matrices are 4x4.

`scalar(Fact,[X1,X2,X3],[R1,R2,R3])`
The scalar product of a scalar and a vector.

`dot([X1,X2,X3],[Y1,Y2,Y3],Result)`
The dot product of two vectors.

`cross([X1,X2,X3],[Y1,Y2,Y3],[R1,R2,R3])`
The cross product of two vectors.

`vecplus([X1,X2,X3],[Y1,Y2,Y3],[R1,R2,R3])`
The sum of two vectors.

`vecminus([X1,X2,X3],[Y1,Y2,Y3],[R1,R2,R3])`
The difference of two vectors.

`matvecmult([A1,A2,A3,[0,0,0,1]],[X1,X2,X3],[R1,R2,R3])`
The vector product of a matrix and a (column) vector.

`vecmatmult(X,MatrixA,[R1,R2,R3,R4]`
The vector product of a (row) vector and a matrix.

`matidentity(IdentityMatrix)`
The identity matrix.

`mattranslate([Tx,Ty,Tz],TranslationMatrix)`
The translation matrix from a given translation vector.

`matrotate([Phi,Theta,Psi],M)`
The rotation matrix from a given Euler rotation vector.

`matrotate_x(Theta,M)`
The rotation matrix of Theta radians about the X axis.

`matrotate_z(Theta,M)`
The rotation matrix of Theta radians about the Z axis.

`matscale([Sx,Sy,Sz],ScaleMatrix)`
The scaling matrix for a vector giving x, y, and z scaling.

`matmult([A1,A2,A3,A4],B,[M1,M2,M3,M4])`
The product of two matrices.

`matmult([A,B | Rest],R)`
The product of a list of matrices.

`matinverse(A,Ainv)`
The inverse of a matrix.

`distance(C1,C2,D)`
The Euclidean distance between two points.

`midpoint([X1,Y1,Z1], [X2,Y2,Z2], Mid)`
The midpoint of two points.

`midpoint(CoordinateList, Mid)`
The midpoint of a set of points.

`direction([X1,Y1,Z1], [X2,Y2,Z2], D)`
The normalized direction from the first to the second point.

`normalize([X,Y,Z], [Nx,Ny,Nz])`
Normalize a vector.

`near1(X,Y)`
Are the two given numbers equal, within some error measure.

`near([X1,Y1,Z1], [X2,Y2,Z2])`
`near([C1,C2 | Rest])`
These points are coincident within some error measure.

`colinear([[X1,Y1,Z1],[X2,Y2,Z2],[X,Y,Z] | Rest])`
Are these points colinear?

`ordered(ListOfCoordinates)`
Are these points ordered?

`bbox_intersect([X1,Y1,Z1],[X2,Y2,Z2],[X3,Y3,Z3],[X4,Y4,Z4])`
Do two given bounding boxes intersect?

Chapter 12

Supplemental Predicates

This chapter describes a small number of relations that supplement the relations provided by the IBM **CLP($\mathcal{R}$)** Interpreter. The **CLP($\mathcal{R}$)** provided relations are described in the IBM **CLP($\mathcal{R}$)** Reference Manual [2]. Additional information on **CLP($\mathcal{R}$)** predicates is available in The **CLP($\mathcal{R}$)** Programmer's Manual from Monash University [3].

12.1 Miscellaneous System Predicates

`system(SystemCall)`
: Presents the given command to the operating system.

`abort`
: Breaks the evaluation and returns to the **CLP($\mathcal{R}$)** interpreter.

12.2 Standard Predicates

`append(L1,L2,L3)`
: Create a new list by concatenating two other lists.

`member(M,List)`
: Determine if the given element is in the list.

`min(X,Y,Min)`
: The minimum of two elements.

`min(List, Min)`
: The minimum element of a list.

`max(X,Y,Max)`
: The minimum of two elements.

`max(List, Max)`
: The maximum element of a list.

Bibliography

[1] B. G. Baumgart. A polyhedron representation for computer vision. In *AFIPS Conf. Proc.*, volume 44, pages 589–596, 1975.

[2] CLP($\Re$) version 1.0 reference manual. Technical report, IBM T. J. Watson Research Center, December 18 1989.

[3] Nevin Heintze, Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Rolland Yap. The CLP($\Re$) programmer's manual. Technical report, Department of Computer Science, Monash University, June 1987.

[4] Jeff A. Heisserman. *Generative Geometric Design and Boundary Solid Grammars.* PhD thesis, Department of Architecture, Carnegie Mellon University, May 1991.

[5] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Rolland H. C. Yap. The CLP($\Re$) language and system. Technical Report CMU-CS-90-181, Department of Computer Science, Carnegie Mellon University, October 1990.

Index