1990

The SHILP Solid Modeling and Display Toolkit A User's Manual

Vinod Anupam

Chanderjit Bajaj

Steven Klinkner

Report Number:
90-988

THE SHILP SOLID MODELING AND DISPLAY TOOLKIT* A USER'S MANUAL

Vinod Anupam
Chanderjit Bajaj
and
Steven Klinkner

Computer Sciences Department
Purdue University
Technical Report CSD-TR-988
CAPO Report CER-90-26
June, 1990

* Supported in part by NSF grant DMS 88-16286 and ONR contract N00014-88-K-0402.

The SHILP Solid Modeling and Display Toolkit *
A User's Manual

Vinod Anupam *Chanderjit Bajaj* *Steven Klinkner*

Department of Computer Science,
Purdue University,
West Lafayettte, IN 47907

Abstract

We describe herein a number of tools designed for the creation, editing and display of algebraic surface models, with an emphasis on "how to use" the facilities. These have been written in *Common Lisp* and *C* (with a small portion in FORTRAN) on various workstations over the past two years.

MakeSolid allows creation of solids through a textual description of vertex, edge and face adjacencies. The **MakeSolid** routine transforms this "external" form into an "internal" (LISP `defstruct`) form with a more redundant adjacency structure, more suitable for manipulation routines (boolean operations, triangulation, etc.).

QMPlot permits quick wireframe plots of the curved solid boundary data structure for arbitrary algebraic surfaces. **QMPlot** effects these plots via calls to a robust FORTRAN surface-surface intersection routine using LINPACK singular value decompositions.

The **Model Creation** and **Model Editing** tools offer the ability (through an interface to **MakeSolid**) to create complete solid boundary descriptions using offset, extrude and revolve operations on a restricted domain of objects. The *offset* operation may be performed on points, a single line segment, or two consecutive line segments. *Extrusions* are permitted on arbitrary polygonal lamina (with holes). Solids of *revolution* can also be created, currently allowing planar lamina (without holes) which lie in the positive half plane of the axis of revolution (including the axis of revolution). Three dimensional solid transformation routines have also been crafted, allowing the user to arbitrarily scale, translate and rotate any portion of a model.

The **Solid Decomposition** tools include robust routines for computing convex decompositions of polyhedra, as well as determining the nesting structure between coplanar polygons.

We also describe a tool for fleshing out the surfaces in a structure using **Hermite** interpolation. Here we use the hull of a solid for its topological structure only; surfaces are crafted by applying vertex, edge and normal constraints and interpolating a surface in between.

Finally we describe the **Solid2Octree** routine used to convert these solids into **Octree** input files. **Octree** in turn decomposes algebraic surfaces into polygons for display on color workstations.

It is our hope that these efforts will be of benefit to all projects involved in the manipulation of solids with boundary descriptions.

*Supported in part by NSF grant DMS 88-16286 and ONR contract N00014-88-K-0402

Contents

1 Introduction

This user manual describes how to use a number of tools which have been developed for the purpose of interactively creating, editing and displaying solid models defined with algebraic surfaces. This manual attempts to address the increasing number of queries concerning the utilization of these tools.

1.1 How To Read This Manual

Each section of this manual describes a major feature of the Graphical Creation, Editing and Display software. Found therein will be details regarding how to load, call and interpret the various routines associated with each feature. Also found will be pointers to the (hopefully) correct place to find additional information.

Where time has permitted, I have also included examples of calling sequences which achieve some of the described results.

1.2 Required Software

Naturally, any mildly sophisticated system will involve a substantial amount of inter-related software. If the user desires to selectively access only those routines associated with a particular feature, we have specified the routines involved in each section.

However, the user may wish to perform a "load all" operation, so as to access all files required for every creation and editing action; this can be achieved (in a crude fashion) by loading a number of files, given below:

- *bhaskar:>qm>klink>loadall3.lisp* will load all of the files associated with **MakeSolid**, **QMPlot**, lamina creation, offsets, extrudes, revolves, and transforms.
- *bhaskar:>qm>anupam>model_editing>edit_loadall.lisp* will load all of the files associated with lamina editing and model editing.

There are several pieces of software which are worth loading at the beginning so you won't have to worry about them again. You will probably want to do the following:

```
(make-package 'sm :use 'scl :colon-mode ':internal)
(load "b:>qm>klink>makesolid>solidmodeldatadefs.lisp")
(load-system 'quadricmodelmathbase :silent t
     :load-patches t :version :latest :no-warn t)
(load "h:>newton>mathbase>eqndefs.bin")
(load "h:>newton>mathbase>eqnops.bin")
(load "h:>newton>mathbase>listmath.bin")
(load "h:>newton>mathbase>arrayops.bin")
```

Also, the FORTRAN system is required to run the **QMPlot** software, and the *Macsyma* system will be required to perform Hermite interpolation.

1.3 More Items Worth Knowing

- (give grammar for the equation format used)

2 MakeSolid

A desire for a more concise solid boundary representation provided the primary motivation behind the creation of **MakeSolid**. For the sake of algorithm efficiency, we would like to provide a highly redundant "internal" solid boundary representation. By specifying these adjacencies *a priori*, many linear-time searches become constant-time.

While such a structure might be ideal for the computer, we seek a more intuitive form for the user: less redundant, free from pointers, defstructs, and other such nasty beasts. However, our specification of a solid still necessitates a complete (unambiguous) boundary description. This, in turn, requires an enumeration of:

1. The x, y, z positions of all vertices
2. Each face equation used (with normals directed outwards)
3. Edges directed from $vertex_i$ to $vertex_j$, specifying the faces to the left and right of the edge, with an initial tangent in the direction of $vertex_j$ from $vertex_i$
4. A cycle of directed edges for each closed face patch (with the solid to the right of oriented edges)

The specification for this data structure can be found in
bhaskar:>qm>klink>makesolid>solidmodeldatadefs.lisp.

2.1 Using MakeSolid

In order to run **MakeSolid**, one must first load the following files:
bhaskar:>qm>klink>makesolid>solidmodeldatadefs.lisp
bhaskar:>qm>klink>makesolid>makesolid3.lisp
bhaskar:>qm>klink>makesolid>conversion3.lisp

Having done so one envokes **MakeSolid** by calling *(makesolid external)*, where *external* is the *(vertices edges equations cycles)* list giving the "external" description of the solid as described in
bhaskar:>qm>klink>makesolid>makesolid3.doc.
See *bhaskar:>qm>klink>makesolid_ex3>*.lisp* for further examples.

2.1.1 Input To MakeSolid

The input to **MakeSolid** takes the form of lists of the required geometric and adjacency information. We refer the reader to the *reference manual* associated with this document and the file
bhaskar:>qm>klink>makesolid>makesolid3.doc for the latest description of the input possibilities. See *bhaskar:>qm>klink>makesolid_ex3>*.lisp* for further examples.

2.1.2 Output From MakeSolid

Makesolid produces a LISP `defstruct` *solid* boundary representation of the given solid, which we call the "internal" form of the solid. We refer the reader to the *reference manual* associated with this document and the file

bhaskar:>qm>klink>makesolid>solidmodeldatdefs.lisp for a complete discussion of this structure.

2.2 int.to.ext

The input to **Makesolid** provides a complete (albeit cumbersome) textual description of a solid. We desire to have this form to pass between incompatible software packages, and to provide persistence of data (since we cannot write a `defstruct` to a file).

We therefore provide the **int.to.ext** utility (so named because it converts from the "internal" to the "external" form). If we call *(int.to.ext solid)*, where *solid* is a *solid* `defstruct`, the output will be a list compatible with **Makesolid** input. A more readable form may be obtained by calling *(int.to.ext solid :output 'text)*.

2.3 Difficulties with MakeSolid

In spite of efforts to make primitive solids easier to create, **MakeSolid** remains a tedious tool. We cannot overemphasize the absolute need to provide the user with the ability to quickly build and edit complex objects by applying operations upon previously created primitive objects. We begin to address these topics in the sections on solid creation and editing.

Ideally, we would like to keep the user at as high an intuitive level as possible, allowing **Makesolid** to query the user for additional information to resolve ambiguities.

3 QMPlot

Originally intended as the wireframe plotting module for a quadric solid modeller, **QMPlot** has evolved into a somewhat more general package. **QMPlot** allows the display of edges defined by the intersection of two arbitrary algebraic surfaces which are delimited by two vertices (assumed to lie on both surfaces). **QMPlot** draws the wireframe of a solid by applying **QMPlot.edge** to each edge of the solid.

In the special case that **QMPlot.edge** encounters an edge specifically marked as type `'LINEAR`, **QMPlot.edge** renders that edge as a straight line between the two delimiting vertices. Otherwise, **QMPlot.edge** calls the `FORTRAN` numerical curve tracer with the delimiting vertices, the two surface equations, an initial tangent vector, and a maximum step size as parameters.

Subsequently, the points generated by **QMPlot** link with another interface (*Dynamic_QMPlot*) which allows rapid rotation, zooming and perspective transformations so that the user might gain a greater appreciation for the three dimensional structure of the solid.

3.1 Using QMPlot

In order to use **QMPlot**, one must first have loaded the `FORTRAN` system, as well as the following files:

bhaskar:>qm>klink>qmplot>linpack.bin
bhaskar:>qm>klink>qmplot>interface.bin
bhaskar:>qm>klink>qmplot>trace.bin
bhaskar:>qm>klink>qmplot>qmplot2.bin

One can simply call *(qmplot solid)*, where *solid* is a *solid* `defstruct`, to start up **QMPlot**. The reader should be aware that **QMPlot** has a fair number of default parameters which may be easily changed by passing key arguments. See the source code for details.

3.1.1 Input to QMPlot

The input to **QMPlot** is a solid boundary description in "internal" form, that is, a `LISP` *solid* `defstruct`. See the above section on "Output from MakeSolid" for more details.

3.1.2 Output from QMPlot

QMPlot does not directly return a meaningful value. However, in the process of traversing all edges of the solid, **QMPlot** produces a list (global variable **qmplot_record**) of all line segments plotted. **qmplot_record** has the structure $(((x_0\,y_0\,z_0)(x_1\,y_1\,z_1))((x_1\,y_1\,z_1)(x_2\,y_2\,z_2))\ldots)$. Since the segments are stored as *((x,y,z) (x,y,z))* pairs, without regard to adjacency, the storage is quite inefficient (roughly twice what is necessary when the edges are curved).

3.2 Numerical Curve Tracer

A detailed description of the algorithm can be found in the paper *"Tracing surface intersections,"* Bajaj, Hoffmann, Lynch and Hopcroft. Implementation details can be found in the reference manual associated with this document.

3.3 Dynamic_QMPlot

In order to provide a 'quick' wireframe plot facility on the *Symbolics* machines, **Dynamic_QMPlot** simply transforms and plots all points in **qmplot_record** in rapid succession. To achieve maximal plotting speed (which unfortunately *is* an issue on the *Symbolics*), **QMPlot** first conses all of the line segments plotted into one big, long list - global variable **qmplot_record**.

QMPlot then calls **Dynamic_QMPlot**, which cycles through **qmplot_record** when the user selects a viewpoint on a sphere parameterized by the x, y position of the mouse. The status window displays the current (temporary) eye position, and we allow the following operations:

- **MOUSE-L**: set the eye position at the point indicated in the status window, and redraw the screen. On the **qmplot window**, the horizontal movement of the graphics cursor is mapped to rotations about the Z axis on a sphere about the gaze point, while vertical movements are mapped to rotations above or below a plane parallel to the X-Y plane. If ***qmplot_center*** is set to *(0 0 0)*, then placing the graphics cursor in the middle of the window results in a view directly down the X axis, with the Y axis to the right, and the Z axis pointing up.
- **MOUSE-M**: abort **QMPlot**.
- **MOUSE-R**: a menu allowing the user to set the following global variables:
  - ***qmplot_center***: gaze point of the viewing eye.
  - ***qmplot_zoom***: the magnification of the object (expressed in pixels).
  - ***qmplot_radius***: the radius of the sphere (with center ***qmplot_center***) around which the eye point travels.

Finally, **Dynamic_QMPlot** provides a service as a viewer of arbitrary wireframes – one only needs to set **qmplot_record** to the list of segments to be viewed, and then call *(dynamic_qmplot)*.

3.4 Individual Space Curves

In many cases one might wish to plot, or generate points upon, an individual space curve without having to actually create a solid to do so. This capability certainly exists; however, due to the nasty `FORTRAN - LISP` interface on the *Symbolics*, the usage is not as nice as one might like.

By default, **QMPlot** plots all the points generated, and appends them to the global list **qmplot_record**. The plot and record actions are controlled by the boolean global variables **qmplot_plot_flag** and **qmplot_record_flag** respectively. To generate a fresh set of points, one can first (*setq *qmplot_record* nil*). This gives us the capability to record points without plotting, or plot without recording points.

Having set the appropriate global variables, we can initiate the curve tracing by calling *(qmplot.tsc_interface p1 p2 tangent step f1 f2)*, where

- **p1** and **p2** are *(x,y,z)* lists giving the starting and ending points on the curve, respectively. These points must be known to lie on both implicit surfaces with a high degree of precision.
- **tangent** is an *(x,y,z)* list giving an approximation to the initial tangent at **p1** in the direction of **p2**. This tangent does not have to be extremely precise.
- **step** is a floating number giving the approximate step between points generated. Since the curve tracer is curvature dependent, this number represents an upper bound on the distance between points.
- **f1** and **f2** are the implicit equations defining the space curve. These must be of the *(degree varlist termlist)* form given in *henry:>newton>mathbase>eqndefs.lisp*.

For example, we can use the following sequence to generate points upon a quarter circle in the X-Y plane, using the equations for the $z = 0$ plane and the the cylinder $x^2 + y^2 - 1 = 0$:

```
(setq *qmplot_plot_flag* nil)
(setq *qmplot_record* nil)
(qmplot.tsc_interface
    '(1 0 0) '(0 1 0)
    '(0 1 0) 0.1
    '(2 (x y z) ((1.0 (2 0 0)) (1.0 (0 2 0)) (-1.0 (0 0 0))))
    '(1 (x y z) ((-1.0 (0 0 1)))))
```

The resulting points will be found in **qmplot_record**.

4 Model Creation Tools

Here we describe several tools which have been developed for the purpose of solid boundary definitions at a high level. We allow offset, extrude, and revolve operations upon a limited class of objects. Translation, rotation and scaling are permitted upon any solid or subset of a solid (i.e. edges, vertices, faces). Here we are primarily concerned with the use of these tools; the functional details will be deferred to the reference manual.

4.1 Lamina Creation

Each of the offset, extrude and revolve routines require a basic shape with which to work. By default, these invoke a lamina creation tool **lamina.get.lamina** which allows the user to create an object suitable to the calling routine. Currently lamina for extrusion must be created in the X-Y plane, and lamina for revolution in the X-Z plane.

Unfortunately, the lamina creation routines do not currently mix interactive editing with creation; however, the ability to edit lamina objects after creation exists. The details of doing so will be found in the section "Model Editing Tools."

4.2 Using Lamina Creation

In order to use **lamina.get.lamina** to create lamina, one must first load the following files:

bhaskar:>qm>klink>model_creation>lamina_creation> lamina_defs.lisp

bhaskar:>qm>klink>model_creation>lamina_creation> lamina4.lisp

in addition to the **QMPlot** software, needed for windowing. Having done so, one may create a lamina by calling *(lamina.get.lamina type)*, where *type* is a string indicating the type of lamina to create, as discussed below.

4.2.1 Input to lamina.get.lamina

To create a lamina in the X-Y plane, call *(lamina.get.lamina "extrude")*; to create a lamina in the X-Z plane, call *(lamina.get.lamina "revolve")*.

4.2.2 Output from lamina.get.lamina

After interactively creating a lamina, **lamina.get.lamina** will return a LISP `defstruct` of type *lamina*. The complete specification of all data elements can be found in

bhaskar:>qm>klink>model_creation>lamina_creation> lamina_defs.lisp

4.2.3 lamina.get.lamina

This section describes the overall features of lamina creation. The essential idea is to place points and close edge cycles until a "no" answer to the **Another Cycle?** question causes the routine to terminate, returning a *lamina* `defstruct`.

When the user enters the lamina creation routine, if the *qmplot* graphics window does not exist, the user will first be prompted to create it using the mouse. Having done so, the mouse cursor will be confined to the graphics window with the status window displaying the following information:

- The current *direction:*
  - **CCW**: counter-clockwise, the direction for all 'outer' cycles enclosing volume
  - **CW**: clockwise, the direction for all 'inner' cycles with volume to the outside.

  Currently the direction value is CCW for the first cycle, and CW for all subsequent cycles.
- The current *mode:*
  - **LINEAR**: line segments will connect every two points placed on the graphics window.
  - **BERNSTEIN-PARAMETRIC**: points placed will be interpreted as the control points of a Bernstein parametric curve.
  - **BERNSTEIN-IMPLICIT**: not implemented yet.
- **MOUSE-L**, **MOUSE-M**, and **MOUSE-R** values:
  - **MOUSE-L**: used to place points on the graphics window. Displayed here will be the X-Y or X-Z position of the graphics cursor. Placing a point may have several side effects. First, a point may not be placed "too close" to the immediately prior point (determined by an epsilon in the program). Second, should the point placed be "close" to the first point in the cycle, the user will be prompted to close the current cycle. A "no" answer will cause the point to be placed as specified, whereas a "yes" answer will forgo placing the point and will, instead, close the cycle and prompt the user for further cycles (holes).
  - **MOUSE-M**: used to abort the creation session, and return nil. Unfortunately this occurs without regard to the previous work accomplished or an "are you sure check" - this should be changed.
  - **MOUSE-R**: calls up a menu with several options:
    * **Extrusion Variables**: allows the user to change the *(x y z)* direction of extrusion. Currently, the lamina can only be pulled in the positive Z direction, so the X and Y components should be zero, and the Z value positive.
    * **Revolution Variables**: here the user may specify the angle of revolution (default is 360 degrees), permitting "pie slices," etc.
    * **Window Variables**: the user may change the center (in X-Y or X-Z) of the graphics window, as well as its height (Y or Z) and width (X). Last, the elevation of the graphics window above or below the plane of interest may be given, allowing for lamina which do not lie in the plane. The elevation should be zero when creating a lamina for revolution.
    * **Change Lamina Mode**: by selecting **LINEAR**, subsequent points will be interpreted as vertices on line segments. By selecting **BERNSTEIN-PARAMETRIC**, following points will be interpreted as the control points of a Bernstein curve. The **BERNSTEIN-IMPLICIT** mode is currently not available.

* **Cycle Completed:** allows the user to prematurely terminate a cycle of edges, thereby creating, for example, a non-closed polygon.
* **Refresh Display:** clears and updates the display.

4.3 Offsets of Lamina

In its current stage, we allow only the *positive* radius offsetting *("growing," not "shrinking")* of any of a point, a line segment, or two line segments with a common vertex. Thus *offset* takes one, two or three connected vertices and an offset radius to produce spheres, "capsules" and "elbows," respectively.

By default, **offset** will force the user to create a new lamina for the purposes of offsetting. One should then should create a lamina containing a non-closed cycle with one, two or three points. The user should place one, two or three points in **LINEAR** mode, and then select the Mouse-R option "Cycle Completed." The result will be a *solid* `defstruct` representing the appropriate offset object.

4.3.1 Using Offset

To perform these offsets, one must first load the file
bhaskar:>qm>klink>model_creation>offsets> offset.lisp
in addition to all the files associated with **MakeSolid** and lamina creation; the **QMPlot** files will also be needed for windowing purposes. One may then simply call *(offset)* to create an offset solid.

4.3.2 Input to Offset

Should the user wish to offset a previously created *lamina*, the following call may be used: *(offset :lamina lamina))*, where *lamina* is the previously-created lamina. Indeed, the call *(offset)* is semantically equivalent to *(offset :lamina (lamina.get.lamina "extrude"))*.

One may specify the radius of offset by passing a value for key argument *radius*, which has a default value of 1.0. For example, one might use the call *(offset :radius 10.0)* to quickly create a sphere of radius 10.

4.3.3 Output from Offset

Offset will produce, via **MakeSolid** a *solid* `defstruct` representation of the offset object. The edges present will specify all surface-surface intersections; however, in some cases additional edges may be present to provide clarity.

4.4 Extrusion of Lamina

The extrusion software performs an extrusion of a previously created lamina in the direction of **extrude_vector**. Because of the ordering imposed by lamina creation, **extrude_vector** should

have a postive z component, and currently **extrude_vector** should have x and y components of zero.

In order to create a solid of extrusion, the user may call *(extrude)* (which is semantically equivalent to *(offset :lamina (lamina.get.lamina "extrude"))*). The user may then create any planar lamina, with or without holes, and with or without curves. The result will be a *solid* `defstruct` depicting the extruded lamina.

4.4.1 Using Extrude

Extrude can be used after loading the file

bhaskar:>qm>klink>model_creation>lamina_ops>extrude4.lilisp

in addition to all the files associated with **MakeSolid** and lamina creation; the **QMPlot** files will also be needed for windowing purposes. One may then call *(extrude)* to create a solid of extrusion.

4.4.2 Input to Extrude

Extrude accepts key argument *lamina*; if this argument is not specified, the user must create a new lamina. If the user would like to extrude a previously created *lamina*, the following call may be used: *(extrude :lamina lamina)*.

4.4.3 Output from Extrude

Extrude will produce, via **MakeSolid**, a *solid* `defstruct` representation of the extruded object.

4.5 Revolution of Lamina

The revolve operation revolves a lamina about the z axis by angle **revolve_angle**, where $0 \leq$ **revolve_angle** ≤ 360. All vertices in the lamina must have nonnegative X components (for a further discussion on why, consult the reference manual).

To create a solid of revolution, the user should call *(revolve)* (which is semantically equivalent to *(revolve :lamina (lamina.get.lamina "revolve"))*). The user may then create any planar lamina, with linear or parametric edges. The result will be a *solid* `defstruct` for the revolved lamina.

4.5.1 Using Revolve

To use **Revolve**, one should first load the file

bhaskar:>qm>klink>model_creation>lamina_ops>revolve4.lisp

as well as the files associated with **MakeSolid** and lamina creation; one also needs the **QMPlot** software for windowing purposes. The call *(revolve)* is then sufficient to create a solid of revolution.

4.5.2 Input to Revolve

Like **Extrude**, **Revolve** accepts key argument *lamina*; if this argument is not specified, the user must create a new lamina. If the user would like to revolve a previously created *lamina*, the following call may be used: *(revolve :lamina lamina)*.

4.5.3 Output from Revolve

Revolve will return, via **MakeSolid**, a *solid* `defstruct` representation of the solid of revolution.

5 Model Editing Tools

The goals behind the model editing tools are:

- Provide the user with a quick, simple interface for correcting mistakes made during solid creation
- Allow the user to create complicated objects by altering existing, primitive objects without requiring an intimate knowledge of the underlying data structure.

Currently our efforts include tools for editing lamina, editing a small class of solids, and performing linear transformations upon objects.

5.1 Lamina Editing

The lamina editing facility allows the user to alter lamina previously created using the lamina creation utilities. For instance, one might wish to add or delete topological features, or alter geometric information. **Lamina.Edit** accepts as input a *lamina* `defstruct`. To edit a lamina, for example, we could call *(lamina.edit (lamina.get.lamina "extrude"))*. The result will also be a *lamina* `defstruct`.

Lamina.Edit also supports several nice features such as interactive "rubberbanding" of lines and multiple orthographic and perspective views of three-dimensional objects.

5.1.1 Using Lamina.Edit

Lamina.Edit requires the **QMPlot** software; all the other files needed can be loaded from the file
bhaskar:>anupam>model_editing>edit_loadall.lisp
A call to **Lamina.Edit** with a previous lamina or null as an argument will start up the lamina editing routine.

5.1.2 Input to Lamina.Edit

An argument of nil to **Lamina.Edit** will cause **Lamina.Edit** to first call *(lamina.get.lamina "extrude")* to produce a lamina for editing. Otherwise **Lamina.Edit** will perform editing operations upon the lamina passed as input.

5.1.3 Output from Lamina.Edit

Lamina.Edit will return the lamina passed as input (or created internally), with its structures accordingly changed.

lamina.edit has two editing modes and several viewing modes, determined by the **MOUSE-R** menu:

- **Edit Modes:**

- **Vertex Mode**: the vertex closest to the graphics cursor will be highlighted or "selected." The **MOUSE-L** menu then allows *delete* and *translate* operations on the selected vertex.
- **Edge Mode**: the edge closest to the graphics cursor will be highlighted or "selected." The **MOUSE-L** menu then permits *insert* and *translate* operations. The *insert* function places a new vertex at the midpoint of the highlighted edge. The *translate* operation allows the endpoints of the edge to be moved, subject to the constraint that the edge remains parallel to its original position.

• **Viewing Modes:**

- **Toggle Screen Mode**: this selection toggles the viewer between a single, large window and four smaller windows – X-Y, Z-Y, X-Z, and Perspective, as described below:
- **X-Y, Z-Y,** and **X-Z**: these views give the obvious projections of the lamina onto the viewing screen.
- **Perspective**: allows a perspective view of the lamina. While in this mode, the user may **Change Perspective** (which gives a **QMPlot** view of the lamina) or **Delete** (which deletes a vertex).

5.2 Solid Editing

The solid editing facility allows the user to alter solids created previously using the solid creation facilities by changing topological features, or altering geometric information. **Solid.Edit** accepts as input a *solid* `defstruct`, or *nil*. If it is invoked with nil as the argument, a new solid is created by calling *(lamina.get.lamina "extrude")*. It passes back a *solid* `defstruct`.

5.2.1 Solid.Edit

Solid.Edit has four editing modes and several viewing modes, determined by the **MOUSE-R** menu:

1. **Edit Modes:**

   (a) *Vertex Mode*: the solid vertex closest to the graphics cursor will be highlighted or "selected". The **MOUSE-L** menu then allows a user to delete the vertex, or translate it under different constraints. A vertex may be translated unconstrained, or constrained along an edge, or constrained to a plane. **Solid.Edit** takes care of the resulting topological alterations. It also checks for simplicity violations in the editing process.

   (b) **Edge Mode**: The edge closest to the graphics cursor will be highlighted or "selected". The **MOUSE-L** menu then permits insert and translate operations. The insert operation places a new vertex at the midpoint of the highlighted edge. The translate operation allows the endpoints of the edge to be moved simultaneously, subject to the constraint that the edge remains parallel to the original position. Translation can also be constrained to occur along a specified edge or in a specified plane.

(c) **Face Mode:** the face identifier closest to the graphics cursor indicates the face to be highlighted or "selected". The **MOUSE-L** menu then allows the face to be translated unconstrained or along an edge, subject to the requirement that the face stay parallel to its original position.

(d) **Global Mode:** This is the default mode of editing. Global editing operations are allowable even when the focus of attention is different. The **MOUSE-R** menu allows the user to translate the solid in space (subject to constraints like translation along an edge or translation in a plane). The user may also rotate and scale the solid.

2. **Viewing Modes:**

(a) **Toggle Screen Mode:** This selection toggles the viewer between a single, large window and four smaller windows - X-Y,Z-Y,X-Z and Perspective.

(b) **X-Y, Z-Y, X-Z:** these views give the corresponding projections of the solid onto the viewing screen.

(c) **Perspective:** allows a perspective view of the lamina. In this mode, the user may also Change Perspective ,which gives a **QMPlot** view of the solid, and allows the user to see the perspective plot from any desired eye-point.

5.3 Solid Transformations

A subclass of the more extensive model editing features, these allow linear transformations to be applied to an entire model. These include translation, rotation and scaling. Note that these operations are actually applied to the entire object and all of its equations (via symbolic subsitution), rather than simply being an artifact of display rendering (which also involve the same linear transformations).

To use the solid transformation routines, you will need to load the following files:
henry:>newton>mathbase>eqndefs.bin
henry:>newton>mathbase>eqnops.bin
henry:>newton>mathbase>listops.bin
henry:>newton>mathbase>arrayops.bin
bhaskar:>qm>klink>model_creation>transforms>matrix.lisp

As of yet, no sophisticated interface exists for these transformation routines; they must be effected via explicit procedure calls. The essential idea involves creating a transformation matrix representing the series of alterations desired. The operations will be applied to the object to be transformed in the order in which they were applied to the matrix. To this effect, we allow the following operations:

- *(identity.matrix 4)* - provides a 4x4 identity matrix; i.e., a "null" transformation matrix.
- *(translate.matrix.by.vector matrix vector)* - causes *matrix* to translate by *vector*, where *matrix* is a previously created matrix and *vector* is a 4x1 array giving the x, y, and z offsets required (the fourth element will be ignored). Returns a *new* matrix.

- *(translate.matrix.by.point matrix point)* - similar to *translate.matrix.by.vector*, except *point* is a *(x y z)* list giving the required offset. Returns a *new* matrix.
- *(rotate.matrix.by.angle matrix axis angle)* - rotate *matrix* by *angle* (in radians) about *axis*, where *axis* is one of 'x, 'y or 'z. Returns a *new* matrix.
- *(matrix.from.segment segment)* - given *segment* = $((x_1 y_1 z_1)(x_2 y_2 z_2))$, constructs a transform matrix which will translate and rotate an object so that it is centered about the segment ((000)(001)). Note that there will be an additional degree of freedom (a rotation about *segment*), so this operation only makes sense for objects which are symmetric about *segment*. Also, no scaling will be performed on the object.
- *(transform.point point matrix)* - transform *point* by *matrix*; *point* should be a *(x y z)* list. Returns a *new* point.
- *(transform.tangent tangent matrix)* - transform *tangent* by *matrix*; *tangent* should be a *(x y z)* list. This routine is like *transform.point*, but note that we desire tangents to be affected only by rotations (not translations). Returns a *new* tangent.
- *(transform.eqn eqn matrix)* - transform *eqn* by *matrix*. *eqn* should be of the *(degree varlist termlist)* form discussed in the introduction to this manual. Note that, in translating an equation, one would like to apply the *inverse* of the matrix used to transform points lying on the equation. Therefore, the higher-level transform routines will correctly handle this. However, if you are calling this routine directly, you may want to use the inverse of the matrix. This routine is non-destructive to the input.
- *(transform.vertex matrix vertex)* - given a *vertex* `defstruct`, returns the same vertex with its point transformed. Therefore, this is a destructive operation.
- *(transform.edge edge matrix)* - given an *edge* `defstruct`, return the same edge with its tangents and equations transformed. This is a destructive operation.
- *(transform.face face matrix)* - given a *face* `defstruct`, return the same face with its equations transformed. This is a destructive operation.
- *(transform.solid solid matrix)* - given a *solid* `defstruct`, return the solid with all of its subcomponents (vertices, edges and faces) transformed. This is a destructive operation.

6 Hermite Interpolation Interface

6.1 Using Concat-Hermite

Loading the file *oscar:>cho>hermite>concat>loadall.lisp* and all the software associated with **QM-Plot**, and lamina creation gives access to the Hermite interpolation interface. You may then call *(concat-hermite)* to begin the interactive session. Flow of control proceeds as follows:

1. A pop-up menu prompts the user as to whether he/she desires a new *qmplot window*, and as to the type of solid to be created from a lamina. Since **QMPlot** automatically creates a window if one does not exist, this operation is somewhat redundant.
2. Enter **lamina.get.lamina** to create a solid of extrusion or revolution, as specified in step 1 above.
3. Using **QMPlot**, provide a perspective view of the solid created in the previous step.
4. From this point onward, we will be within the **solid.select** routine, which has the following options available:
   - **MOUSE-R: Global Ops Menu** - gives the following choices:
     - **another view**: re-enters **QMPlot** to select another viewing position
     - **vertex mode**: makes vertices active for selection
     - **edge mode**: makes edges active for selection
   - **MOUSE-M: Finished?** - prompts to see whether the user has finished with **solid.select**; if so, we proceed to the next step, otherwise we re-enter the routine.
   - **MOUSE-L: Local Ops Menu** *(vertex mode)* - gives the following choices, some of which are leftovers from Anupam's **solid.edit** routine, and should probably be removed.
     - **Select Vertex**: select a vertex for use in the Hermite interpolation. This vertex will provide a constraint for the interpolation in the form of a point through which the surface must pass. Having selected a vertex, the user will be asked whether a normal will be associated with this point as an additional constraint. If so, the normal will be chosen from one of the face equations at that vertex. The user enters an interactive process through which each face can be highlighted; when the desired face is highlighted, the user can select that face's normal. The user may also specify a face by its numerical index using the **other** option.
     - **Translate on Edge**: identical to the **Solid.Edit** routine of the same name.
     - **Translate in Plane**: identical to the **Solid.Edit** routine of the same name.
     - **Translate in Space**: identical to the **Solid.Edit** routine of the same name.
     - **Relocate Vertex**: identical to the **Solid.Edit** routine of the same name.
   - **MOUSE-L: Local Ops Menu** *(edge mode)* - gives the following choices:
     - **Select Edge**: selects an edge
     - **Insert Vertex**: inserts a vertex into the midpoint of an edge

5. Display a final menu requesting three items of information:

   - **Interpolate?**: answer "no" to skip the interpolation
   - **Degree?**: the degree of the interpolating function
   - **Solid Description**: answer "yes" if you would like to perform the **int.to.ext** function on the final solid, thus printing the "external" form of the solid onto the terminal.

6.2 Input to Concat-Hermite

Concat-Hermite currently does not accept any inputs. Allowing a previously created solid to be passed would be a valuable option.

6.3 Output from Concat-Hermite

The output from **Concat-Hermite** is a linear system in *Macsyma* format, compatible with Ihm's Hermite interpolation program. The input to the interpolation program will eventually take the form of points, point-normal pairs, and algebraic curve equations, and the final output will be an interpolating algebraic surface.

7 Polyhedral Decomposition

Here we describe routines for decomposing solids. The **Conv** program takes a *solid* and produces a convex decomposition of the *solid*. The **Polnest** routine computes the nesting structure of a set of planar polygons. Robustness issues have received particular attention in the implementation of these routines.

7.1 Conv

In order to run **Conv**, you should have loaded the **QMPlot** and lamina creation and editing software. In addition, loading the file

george:>dey>decomp>loadall.lisp

will load all of the decomposition software.

7.1.1 Input to Conv

Conv accepts as input a *solid* `defstruct`. The *solid* in this case must be a polyhedron (for example, the *extrude* of a polygon).

7.2 Output from Conv

Conv will produce as output a list of solids representing the convex decomposition of the input *solid*. Each output solid will be a convex polyhedron.

One may display the decomposition of a solid via *(qmplot.model (conv (extrude)))*. To plot an exploded view of the decomposition, try *(qmplot.model (explode (conv (extrude))))*. **Explode** is a short routine which will explode each output solid outward; **QMPlot.Model** applies **QMPlot** to a list of solids.

7.3 Polnest

As with **Conv**, you should have loaded the **QMPlot** and lamina creation and editing software. In addition, you should load the file

george:>dey>pnest>pnestload.lisp

7.3.1 Input to Polnest

Polnest takes as input a lamina produced from **lamina.edit** or **lamina.get.lamina**; the lamina cycles represent the polygons. One may call, for example, *(polnest (lamina.edit nil))* to envoke **Polnest**.

7.3.2 Output from Polnest

Polnest produces as output a list of cycles representing the nesting structure of the polygons. Each of these cycles cotains a list in its *marked* field. The first element in this list is a pointer tot

he parent cycle; the rest pointer to the children.

Polnest also displays the polygons (lamina cycles) on the *qmplot* window, and displays for each polygon a string of indices separated by "-" on each polygon, representing its nesting.

8 Color Rendering of Solids

Here we describe several utilities for decomposing solid models into polygons for rendering on graphics workstations. First we describe **Octree**, an octree decomposition and polygonalization program, and then **Solid2Octree**, which converts the "external" form of a solid (see **int.to.ext** under the section on **MakeSolid**) into a series of **Octree** input files.

8.1 Octree

Octree is a program for polygonalizing surfaces by spatial decomposition. The input is an implicit function, with bounding surfaces and parameters for the spatial decomposition and polygonalization.

8.1.1 Using Octree

The current working version of **Octree** resides on *medusa*, in the directory */u13/cutchin/bin*. To run **Octree**, you must first create an **Octree** input file, as described below. Having done so, you may run **Octree** using either *oct input_file > output_file* or *oct < input_file > output_file*.

8.1.2 Input to Octree

The input data file contains the following information:

1. equations section – the implicit equations of the surfaces to be polygonalized
2. optional clipping surfaces section
3. octree section – parameters of the octree decomposition

Equations occupy the first section of the input file, where variables are also declared. Equations should be entered in Macsyma format, with each equation given a name. Equations may not contain the division sign, and exponents must be integers. A function *diff(equation, variable)* may be used to symbollically compute the partial derivative of an *equation* with respect to a *variable*. **Octree** assumes that only three variables (i.e. x, y, z) will be declared and used in the equations section.

The second (optional) clipping surface section provides octree with capability of clipping the primary surface with other surfaces. This permits clipping of the primary surface using equations other than those of the bounding box. The clipping surface equations should be specified in the same manner as those in the equations sections (and should use the same variables declared there).

The third section of the input file is the octree section. This section contains information for the octree decomposition and polygonalization, such as the initial cube and the resolution. See the section on keywords below.

Notes on Parsing:

1. Keywords are recognized only in lower case.

2. A maximum of 256 equations may be defined.
3. An exclamation point in the datafile indicates that the remainder of the input line is a comment.
4. Identifiers may be a maximum of 256 characters long.
5. A maximum of 256 clipping surfaces may be defined.

The data file format is as follows (keywords are shown here in capital letters, and user-supplied values are enclosed in <>):

```
BEGIN [EQUATIONS]
        VARIABLES: <var_list>;
        EQUATIONS: <eqn_list>
END [EQUATIONS]


BEGIN [PLANES]
        PLANES: <eqn_list>
END [PLANES]


BEGIN [OCTREE]
        OBJECT: <eqn_name>;
        CUBE_ORIGIN: (<float>, <float>, <float>);
        CUBE_EDGE: <float>;
        MIN_CUBE_EDGE: <float>;
        MIN_SUBDIV_LEVEL: <integer>;
        MAX_SUBDIV_LEVEL: <integer>;
        OUTPUT_NORMALS;
END [OCTREE]


<var_list> : <var_list>, var
           | var


<eqn_list> : <eqn_list>; eqn_name : eqn;
           | eqn_name : eqn;


<eqn_name_list> : <eqn_name_list>, eqn_name
                | eqn_name
```

The following is an example input file:

```
begin [equations]
        variables: x, y, z;
        equations:
                sphere: x^2 + y^2 + z^2 - 1;
end [equations]

begin [planes]
planes:
cutter: x - 0.5;
end [planes]

begin [octree]
        object: sphere;
                cube_origin: (-1.1, -1.1, -1.1);
                cube_edge: 2.1;
                min_cube_edge: 0.27;
                output_normals:;
end [octree]
```

Keywords in Octree Section

- **object** This describes which equation from the equations section should be used as the primary surface for decomposition.
- **cube_origin** The "origin" of the initial octree decomposition cube. The "origin" is the lower left front vertex of the cube (i.e., minimum point in x, y, and z).
- **cube_edge** The edge length of the initial cube.
- **min_cube_edge** The minimum length of a cube to be used during decomposition. This defines the resolution of the output. Generally, a minimum cube edge 1/16 of the inital cube edge is sufficient.
- **output_normals** If this keyword is given on the command line, surface normals for every vertex will be printed to the output file. This is necessary for accurate rendering of the polygons using Gouroud or Phong shading. Using the actual surface normal results in better shading than using the polygon normal.
- **min_subdiv_level** If this keyword is specified, the spatial decomposition algorithm will first check the level of subdivision before evaluating the function at the vertices of the cube. If the minimum subdivision level has not been reached, the cube will be decomposed, even though the value of the surface at the vertices may not indicate subdivision. If the minimum subdivision level is N, then at least 8^N leaves will be in the octree. This parameter is helpful for locating the surface when the location or size of the initial cube cannot accurately be

determined, since spatial decomposition will occur beyond the first level. A reasonable value is 3.

- **max_subdiv_level** This integer is similar to the minimum subdivision level. If the maximum subdivision level is specified, the spatial decomposition algorithm will first check the level of subdivision before evaluating the function at the vertices of the cube. If the maximum subdivision level has been reached, the cube will not be decomposed, even though the value of the surface at the vertices may indicate subdivision.

  Specifying a maximum subdivision level of 4 is equivalent to giving a initial cube edge length of 16 and a minimum cube edge length of 1.

Hints

If the decomposition results in gaps or missing polygons, it may be helpful to shift the initial cube location, or slightly change the cube edge length.

Notes

The program attempts to determine how far the current surface location is from a clipping surface and will increase the number of polygons generated as it approaches the surface. This at the present time causes gaps in the primary surface.

8.1.3 Output from Octree

The output consists of a list of polygons. For each polygon, the number of vertices preceeds the list of x, y, z positions, one point per line. If the OUTPUT_NORMALS keyword had been given in the input file, the normals at each vertex will also be given in the output. The x, y, z values of the normal will follow the vertex x, y, and z positions on the same line.

A sample output file for three faces of a cube (without normals) is:

```
4
0 0 0
1 0 0
1 0 1
0 0 1
4
0 1 0
1 1 0
1 1 1
0 1 1
4
0 0 0
0 0 1
0 1 1
0 1 0
```

The above input file with normals is:

```
4
0 0 0    0 -1 0
1 0 0    0 -1 0
1 0 1    0 -1 0
0 0 1    0 -1 0
4
0 1 0    0 1 0
1 1 0    0 1 0
1 1 1    0 1 0
0 1 1    0 1 0
4
0 0 0    -1 0 0
0 0 1    -1 0 0
0 1 1    -1 0 0
0 1 0    -1 0 0
```

The components of the point are in the same order as the variables declared in the input file (i.e., the order could be y x z). Since the output file does not have a header, output files may be concatenated together if desired.

8.2 Solid2Octree

Solid2Octree takes the "external" form of a solid and creates output files in a form compatible with **Octree** input. This tool has been crafted primarily for the purpose of providing an automatic method for rendering previously created solids. **Solid2Octree** will use heuristics to create as many **Octree** input files as it deems necessary to render the solid.

8.2.1 Using Solid2Octree

To use **Solid2Octree**, you should first load the file *george:>cutchin>solid2octree.bin*. You should also have loaded the FORTRAN system as well as the **QMPlot** software, as **Solid2Octree** calls the numerical curve tracer.

Having loaded these files, one may simply call *(solid2octree solid.ext "fileprefix")* to invoke **Solid2Octree**, where *solid.ext* is the "external" description of a *solid* `defstruct`, and *"fileprefix"* is the prefix for the files where you would like to send the octree data files. The **Octree** input files will be written to the directory of the specified file prefix.

8.2.2 Input to Solid2Octree

Solid2Octree accepts as input the "external" form of a solid (typically obtained by calling **int.to.ext** on a *solid* `defstruct`), and a file prefix string, e.g. *"medusa:/usr/joe/octree/sphere"*.

Solid2Octree will then produce a series of files *sphere0.oct, sphere.1.oct, ...* in the given directory.

8.2.3 Output from Solid2Octree

Solid2Octree will return the number of **Octree** input files it created. The **Octree** files will be written to the specified directory using the given file prefix; these files should be used as input to **Octree**.

8.3 Putting Octree and Solid2Octree Together (Color Rendering)

Here is an example of how you can create a solid, create **Octree** input files, run **Octree**, and finally render the solid on a graphics workstation.

Perform these steps in order:

1. Create a solid "sample." For example, call *(setq sample (revolve))*, and create a solid of revolution.
2. Obtain the external form of the solid by calling *(setq sample.ext (int.to.ext sample))*.
3. Call *(solid2octree sample.ext "filepref")* to create the **Octree** files, where "filepref" is an appropriate file prefix name, for example "medusa:/usr/joe/octree/sample".
4. Log into *medusa*, and call */u13/cutchin/bin/oct < filepref N.oct > N.out* for each *filepref N.oct* that **Solid2Octree** produced.
5. Find a program which can render the polygonal output of **Octree**. By plotting all of the *N.out* files together, you should (hopefully) have a reasonable rendition of the solid.

9 Future Work

The rapidity of progress in this area depends on the tools previously crafted more than anything else. Features which can be implemented from existing software proceed orders of magnitude faster than features which must be started from scratch.

Here give an outline of capabilities yet to come.