GMPOLY: A Kernel Level Polyhedral Solid Modeler

Vijay Natarajan   Rajesh Kumar
Visual Computing Group
Design and Development Center
TATA ELXSI Ltd.
Bangalore, India

June 8, 1999

Abstract

In this paper we describe GMPOLY, a kernel level polyhedral solid modeler that we have developed at the Design and Development Center of TATA ELXSI Ltd., Bangalore, India. The modeler has been designed as a class library that can be used to create, model and represent polyhedral solids using convex polyhedrons as primitives. The library can also be used to define new primitives. We describe the algorithms and procedures that have been used in the design of this modeler.

1 Introduction

A Solid Modeler helps in representing solids on the computer. It can be used to create/model solid objects, make images of them for viewing and to calculate some of their properties like volume, surface area, center of gravity etc. GMPOLY is a kernel level Polyhedral Solid Modeler that can be used to represent polyhedral solids i.e. solids whose boundaries are made up of planar surfaces. It is a kernel level modeler because it has been developed as a class library that can be built into the application programs.

GMPOLY is a hybrid modeler. The solids are stored in both the CSG and Boundary (B-rep) forms. CSG is the primary form of representation. The boundary model is constantly updated, using the CSG tree, by the boundary evaluation procedure. The user, however, has no direct access to the secondary representation (B-rep). This means that although boundary models are included in GMPOLY, the user cannot perform modeling operations like local modifications, which are specific to boundary models. Modelers like PADL-1 [5], PADL-2 [2], and GM_SOLID [1] have a similar architecture. Mantyla [3] gives a good description of the common architectures for hybrid modelers.

GMPOLY models complicated polyhedral solids by taking simple convex polyhedrons and combining them together by using the Boolean operators: Union, Intersection and Difference. These operators are analogous to the ones in set theory. The Boolean operators implemented in GMPOLY are regularized *i.e.* they do not produce solids that have dangling edges/planes (invalid solids). Mortenson has given a detailed description of Regularized Boolean operations in his book on Geometric Modeling [4].

Most of the solid modelers either do not handle self-intersecting solids or represent them as a collection of simpler solids, making the operations on the resulting solids impossible without additional preprocessing steps. GMPOLY attempts to handle various degenerate cases like self-intersecting and touching solids, by using the neighbourhood model in the boundary evaluation procedure.

In the following section we define the basic terms. Section 3 deals with the boundary evaluation procedure, which is the core module of GMPOLY. In Section 4 we describe a few implementation details. In the last section we give our conclusions and suggest modifications and further enhancements that can be incorporated.

2 Preliminaries

In this section we define and give brief descriptions of some of the terms that are used in the later sections.

2.1 Half Spaces

A plane divides the space into 2 *half-spaces*. Each of these *half-spaces* is a semi-infinite solid region. In GMPOLY a *half-space* $(Ax + By + Cz + w \leq 0)$ is represented by the equation of the corresponding plane $(Ax + By + Cz + w = 0)$. The $\leq$ sign is assumed. So, the *half-space* $Ax + By + Cz + w \geq 0$ would be represented by the plane $-Ax - By - Cz - w = 0$.

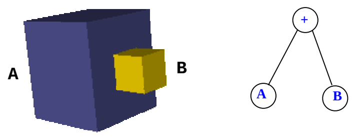

Figure 1: A Polyhedron and its CSG tree

2.2 Primitives

The *primitives* used in GMPOLY to build complicated solids are *convex polyhedrons*. Each *primitive* is represented as a set of *half-spaces*. The *primitives* form the leaves of the CSG tree. Every internal node is labeled by the Boolean operation to be performed on the two sub-trees rooted at that node. Figure 1 shows a polyhedron and the corresponding CSG tree.

2.3 B-rep

The boundary representation (*B-rep*) of the object is updated whenever the CSG tree is modified. This is done by a boundary evaluation procedure. The boundary of each polyhedron is stored as a list of faces. Each face has a *parent* (surrounding) loop and a

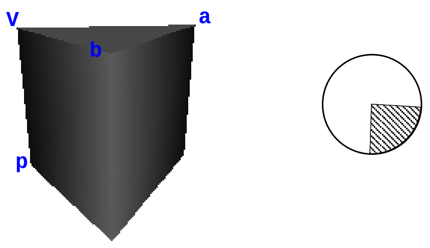

Figure 2: A pyramid and the neighbourhood of a line segment ab

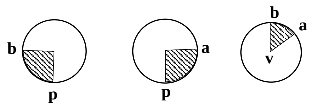

Figure 3: Neighbourhoods of the vertex v on the left, back and top faces

list of *child* loops (holes). A loop is represented as a list of edges, which is in turn represented by its end point vertices.

2.4 Neighbourhood Model

Researchers at the University of Rochester have proposed a **neighbourhood model**, consisting of points close to a line segment/vertex, to indicate whether the line segment/vertex is *in/on/out* of the solid. Mortenson [4] gives a good description of this model and how it can be used to classify line segments/vertices.

Besides classification of line segments and vertices, GMPOLY uses this model to extract face and solid information from the list of boundary edges and vertices. The neighbourhood of a line segment (*i.e.* at its mid-point) differs from that of a vertex.

The neighbourhood of a line segment is the cross section (on the plane passing through its mid-point and having the line as normal) of the volume neighbourhood at the mid-point. This neighbourhood is represented as a sequence of pairs of tangents to the bounding faces. Figure 2 shows a solid and the neighbourhood of the line segment ab alongside. A neighbourhood is constructed for a vertex with respect to each plane that it lies on. This neighbourhood is represented as a sequence of pairs of bounding line segments. Figure 3 shows the neighbourhoods of the vertex v (in Figure 2) on the three planes that it lies on.

3 Boundary Evaluation

The operations on the CSG tree, for implementing the Boolean operations on solids, are similar to ordinary binary tree operations. So, we are not describing them here. In this section we will describe the boundary evaluation procedure, which happens to be non-trivial, in detail. The input to the boundary evaluation procedure is the CSG tree and the output is the B-rep of the solid. Both of the above representations have been described in the previous section. So, we will directly move on to the description of the algorithm.

The boundary evaluation is done in three stages:

1. Generation of tentative edge and potential vertex list.
2. Classification of tentative edges and potential vertices to get the list of edges and vertices on the boundary of the solid.
3. Extraction of face and solid information using the neigbourhood model.

We now describe each of these stages.

3.1 Generation of tentative edge and potential vertex list

The boundary of the solid, represented by the given CSG tree, is made up of the set of faces on the boundaries of the primitives (leaf nodes). In this stage, we generate the list of all possible boundary edges (t-edge or tentative edge) and vertices (p-vert or potential vertex). These lists are named *lote* and *lopv*. A list (*lop*) of all the planes on the boundary is first generated. All planes in *lop* are intersected to get the lines on which the t-edges lie. These lines are now intersected with every plane in *lop* to give the list of potential vertices and tentative edges.

3.2 Classification of tentative edges and potential vertices

Each of the edge segments, obtained from lote by taking consecutive parameter values in each line entry, is classified as *in/on/out* of the polyhedron. This is done, by first classifying the edge segment against each of the primitives (leaf nodes). The edge neighbourhood is constructed with respect to the primitives at this stage. These neighbourhoods are, now, combined using the Boolean operations on the parent node in the tree. So, we finally land up with the edge neighbourhood with respect to the entire CSG tree (polyhedron). This neigbourhood can be easily interpreted to give the classification. See Mortenson [4] for a detailed description of edge classification using the neighbourhood model.

All edge segments that are classified as *on* are inserted into the list of boundary edges (*lobe*) along with their neighbourhoods. The list of boundary vertices (*lobe*) is also updated by adding the endpoints of the above edge segment. The neighbourhood of each of these boundary vertices, with respect to each of the planes that they lie on, is now computed. The neighbourhood of a vertex v on a plane p is computed, by first getting those boundary edge segments that lie on p and are incident on v. These edge segments are then sorted in the clockwise order (about the vertex). Now, the area between 2 successive edge segments is classified as *on/out* of the boundary. If there is a tangent entry in the neighbourhood of an edge corresponding to the plane area to its right, then the region swept around the vertex from this edge to the next edge is classified as *on* the boundary. All other regions are *out* of the boundary.

3.3 Extraction of face and solid information

Once the edges and vertices on the boundary of the object are known, the next step would be to extract the topological relationships between them *i.e.* the face information. The neighbourhood model is used to do this, as follows:

First, for each plane that contains boundary vertices, we get the list of loops on the plane. Then, each of these loops is classified as a surrounding loop (*parent*) or a hole (*child*). Once this is done, the faces (consisting of one *parent* loop and zero, one

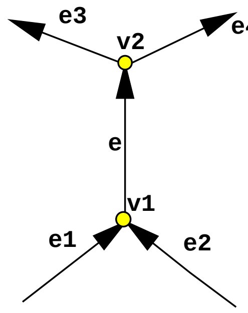

Figure 4: Two entries in the neighbourhood of $v2$ contain e

or more *child loops*) on each plane are recognized. These faces are now grouped together, using shared edges and their neighbourhoods, to give the list of faces for every polyhedron. We now describe the above procedures in detail.

3.3.1 Generation of list of loops

Consider the generation of the loops on a particular plane (p). The current vertex (along with its neighbourhood in p) and the current edge are used to get the next vertex and next edge in the loop. This process terminates when the next edge is the same as the first edge. The neighbourhood of the current vertex gives the pair of the incoming edge (current edge). This is the next edge. There can be one or two entries corresponding to the current edge in the neighbourhood. We explain how the conflict is resolved (when there are two entries) using an example (Figure 4).

Let e be the current edge. An edge status is maintained and used to determine the correct pair. The initial value of edge status is FREE_CHOICE. If the edge status is IN_FIRST (*i.e.* *e2* is the previous edge) then $e4$ is chosen as the next edge. If edge status is IN_SECOND then $e3$ is chosen. In this case *e1* would have been the previous edge. If the edge status were FREE_CHOICE then one of the two edges is chosen arbitrarily. The edge status is updated after the next edge is determined as described below. If there are two pairs for the next edge, in the neighbourhood of the next vertex ($v2$) on p, the edge status is computed by getting the orientation of e with respect to the other pair. If e were the only pair of the next edge then edge status remains unchanged.

3.3.2 Classification of loops

A loop can be classified as *parent/child* by taking an edge of the loop oriented such that the interior lies to the right of the edge. If the edge neighbourhood has a tangent entry corresponding to the plane to the right of the edge (*i.e.* the area interior to the loop and near the edge lies on the boundary of the solid) then the loop is a parent loop. Else, it is classified as a child loop.

3.3.3 Grouping loops into faces

Given the list of loops on a plane along with their classification the faces on the plane can be obtained as follows:

There is a face corresponding to every *parent* loop. The *parent* loop (and hence the corresponding face) containing a given *child* loop can be found out as the smallest *parent* loop that encloses the *child* loop.

3.3.4 Grouping faces into solids

Once the list of faces (*lof*) is obtained the next and final step is to group the faces of a polyhedron together and hence get the list

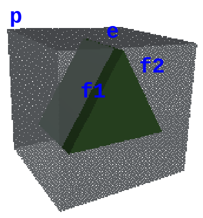

Figure 5: Neighbourhood of e has two entries corresponding to plane p

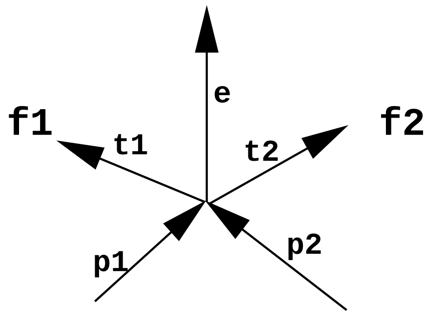

Figure 6: Two faces $f1$ and $f2$ on the same plane sharing the edge e

of polyhedral solids represented by the CSG tree. This grouping is done as follows:

The first face can be chosen arbitrarily from one of the faces in lof that have not been considered previously. Recursively, for every edge (e) in this face f add the face that is paired with f in the edge neighbourhood (provided that the face has not been considered previously). Getting the face that is paired with f from the edge neighbourhood is done in two steps:

Let l be the loop of f that contains e. Since the edge neighbourhood contains plane information, the plane pair is found out initially. There could be 2 tangent entries, in the edge neighbourhood corresponding to the current plane (the one that contains the current face). For example, in Figure 5, plane p corresponding to face $f1$ has two pairs ($f2$ and $f3$) in the edge neighbourhood of e. The correct pair can be determined by generating a point that is on p, near the edge on the side given by the tangent direction in the neighbourhood, and checking if it lies in the interior of l. If the point lies in the interior of l then the current tangent entry is the required entry and its pair is returned.

Once the plane pair is determined, all the faces lying on that plane are scanned to get the face pair. The face that contains e is the correct face. Once again, there could be two (but, not more) such faces ($f1$ and $f2$). In order to resolve the conflict we use the previous edge in the loops $l1$ (from $f1$) and $l2$ (from $f2$) that contain e. The angular position of the tangent (that is returned from the previous step) with respect to the previous edges and e gives the required loop and hence the required face. For example, if the faces are as in Figure 6 and $t1$ ($t2$) is the tangent then $f1$ (respectively, $f2$) is the required face.

4 Implementation Details

In this section, we describe some of the implementation issues involved in the development of GMPOLY. The user is provided with a class library, which could be used to create both primitives and complex polyhedrons. The user need not know the details of the solid representation and manipulation schemes. A simple interface helps in necessary operations for modeling a polyhedron.

GMPOLY is written in the C++ language. A C++ compiler along with the standard template library (STL) is the only support that is required to use this class library. This makes it platform independent.

There are 2 classes that can be used. The first one, called *ConvexPolyhedron*, corresponds to the primitive. It contains the *half-space* representation of the primitive. Methods to load a primitive from a file, create a primitive given the half-spaces and store a primitive object into a file have been implemented.

The second class, called *Polyhedron*, corresponds to the modeled object. It contains the CSG as well boundary representations of the Polyhedron. The two representations are made consistent after every operation. These representations are not visible to the user though. The $+$ $(+=)$, $*$ $(*=)$ and $-$ $(-=)$ operators are overloaded to implement the Union, Intersection and Difference operations. Load and Store functionalities are also provided.

5 Conclusions

GMPOLY has a very simple design philosophy. Any person who has a basic knowledge of programming in C++ can understand and start using GMPOLY within a day. Since it is a kernel level modeler, it can be built into application programs. The user is provided with simple tools (that are analogous to the Boolean operations) to model complex polyhedral solids. The design is generic in the sense that the basic skeleton could be used to design a solid modeler that handles non-polyhedral solids also.

Many modelers do not support self-intersecting solids or represent them as a collection of simpler solids. This makes further operations on them impossible without some pre-processing. GMPOLY attempts to handle most of the degenerate cases (like self-intersecting solids) efficiently, by using the neighbourhood model for edge/vertex classification.

Further development can be done by optimizing the boundary evaluation procedure and incorporating additional features like storage of mass proprties. All search routines could be optimized by storing the edges and vertices in a geometric data structures instead of linear lists that are currently used.

References

[1] J.W. Boyse and J.E. Gilchrist. GM-SOLID: Interactive Modeling for Design and Analysis of Solids. *IEEE Computer Graphics and Applications*, 2(2):86–97, March 1982.

[2] C.M Brown. PADL-2 - A Technical Summary. *IEEE Computer Graphics and Applications*, 2(2):69–84, March 1982.

[3] Martti Mantyla. *An Introduction to Solid Modeling.* Computer Science Press, Rockville, Maryland, 1988.

[4] M. Michael Mortenson. *Geometric Modeling.* John Wiley and Sons Inc., 1985.

[5] H.B Voelcker. The PADL-1.0/2 System for Defining and Displaying Solid Objects. *Computer Graphics*, 12(3):257, July 1978.