1992

Proxima, A Polyhedral Interrogation Support in C++

Gang Sun

Peter Van Vleet

George Vaněček

Report Number:
92-089

PROXIMA, A POLYHEDRAL INTERROGATION SUPPORT IN C++

Gang Sun
Peter Van Vleet
George Vanecek, Jr.

CSD-TR-92-089
November 13, 1992

Proxima, A Polyhedral Distance and Classification Support in C++

Gang Sun Peter Van Vleet George Vaněček, Jr.

Computer Science Department
Purdue University
West Lafayette, IN 47907

November 24, 1992

Abstract

This paper describes Version 1.0 of *Proxima*, a C++ based library support for polyhedral objects. *Proxima* provides topological adjacency, classification and distance-based operations. The classification operations consist of the point/solid, the line/solid, the polygon/solid and solid/solid operations. At present only the point/solid distance operation is supported.

Proxima is based on a data structure that unifies the *multidimensional space partitioning* tree and a *boundary representation* known together as the *Brep index*.

With *Proxima*, application developers can deal directly with the problems at hand and not waste time implementing low level geometric modeling support. Example applications that are well suited to take advantage of *Proxima* are, physical-based simulators and animation systems needing contact analysis, mesh generators and ray tracers needing point and line/solid classifications, and virtual reality and robot guidance systems needing collision avoidance.

Contents

1 Introduction

Recent years have seen a large increase in the number of applications dealing with our physical environment. This surge is made possible by the advances in powerful workstations, computer graphics hardware, visualization techniques and geometric modeling. One common factor in these applications is the use of polyhedral objects. Consider applications in the well known areas of animation, virtual reality, ray tracing, robot guidance, physical-based simulation, and mesh generation. In all these areas, the objects are assumed to be constructed by some geometric modeling system after which the objects are used but not modified. Their use requires only a small subset of the many operations supported by a full sized geometric modeling system. Thus a small subsystem consisting of an interrogation support for objects is sufficient.

Proxima is such a small subsystem which trades the generality of a large solid modeler for the speed of few well chosen operations. Its intent is to provide an efficient low-level object-interrogation support with emphasis on the speed of the supported operations. Since the objects are static, they can be preprocessed to yield an efficient representation that unifies a *multidimensional space partitioning* (MSP) tree and a *boundary representation* (Brep) known jointly as the *Brep index*. The boundary is represented as a variant of the winged-edge representation with the vertices, edges and faces stored in separate arrays, and each entity maintaining references to adjacent entities. The MSP tree is represented as a ternary tree whose leaves index the vertices, edges and faces of the Brep. Consequently, the MSP tree serves as a spatial index into the Brep.

Currently, *Proxima* provides the following support:

- topological adjacency information between vertices, edges and faces, as well as the related geometric information,
- the point, the line segment, the polygon and the solid classification operations, and
- the point distance operation—in the next release *Proxima* will also include the line, the polygon and the solid (i.e., operations that determine the shortest distance from a line to a solid, from a polygon to a solid, and between two solids) distance operations.

Proxima is written in the C++ programming language and comes as a library, include files and several examples. This paper is the user manual for *Proxima* and describes its language interface. For details on the Brep-index data structure, please refer to the appropriate references listed at the end of the paper. In the next section we only briefly review the Brep-index. In Section 3 the external file formats are described. In the subsequent sections, the C++ classes are described in detail and several short examples are given where appropriate. In Section 5, we present two full examples, and in Section 7, the layout of *Proxima* is described along with the instructions for obtaining *Proxima* by ftp, and for compiling *Proxima* and your application programs.

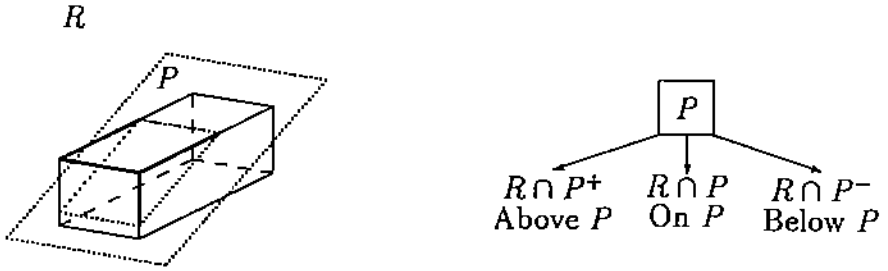

Figure 1: Region R cut by a plane P is partitioned into three subregions that compose the three subtrees of node represeting R.

2 The Brep-Index

The Brep index is a data structure [3] which is a generalization of the well known binary space partition (BSP) tree [2]. The Brep index consists of a ternary-tree that represents a recursive multidimensional partitioning of space called an MSP tree, and a Brep. Associated with each node is a convex region of space where the root node represents the entire space. Associated with each nonleaf node n of the MSP tree is a cut plane P that partitions the region represented by n into three subregions (refer to Figure 1). The three subregions are represented by the three children of n, and are labelled above, on and below P respectively. The resulting spatial partition is called multidimensional because the leaves of the tree represent zero, one, two and three-dimensional regions. If the region represented by a node is of dimension d, then the subregions above and below P are also of dimension d, but the subregion on P is of dimension $d-1$.

As an example of a Brep index, consider the boundary representation of a tetrahedral solid shown in Figure 2. The figure shows the MSP tree, and the Brep with labelled vertices, edges and faces. A cut plane P_i in the Brep index is a support plane of face F_i, and has a normal vector that points away from the solid. Since the tetrahedron is a convex solid, there is only a single region lying inside the solid, and this region is indicated in the figure by a black node.

If a solid is convex, then we can always construct a Brep index of size $v+e+f$, where v, e, and f are the number of vertices, edges and faces, respectively. For example, the tetrahedon of Figure 2 has four vertices, six edges and four faces, and a tree with 14 internal nodes. For nonconvex solids, the size grows as a result of global fragmentation caused by cut planes that slice through the solid. There is no one unique tree for a solid. The MSP trees for a solid vary depending on the ordering and choice of the cut planes. Changing the order while creating the tree results in an equivalent, but different MSP tree of possibly different size. A relatively balanced tree can be attained by initially choosing cut planes that split the object roughly in half. Thus both the size of the MSP tree and its shape can be controlled by chosing a different order of cut planes when creating the MSP tree. As an example, a convex polyhedron with 128 faces may have an MSP tree

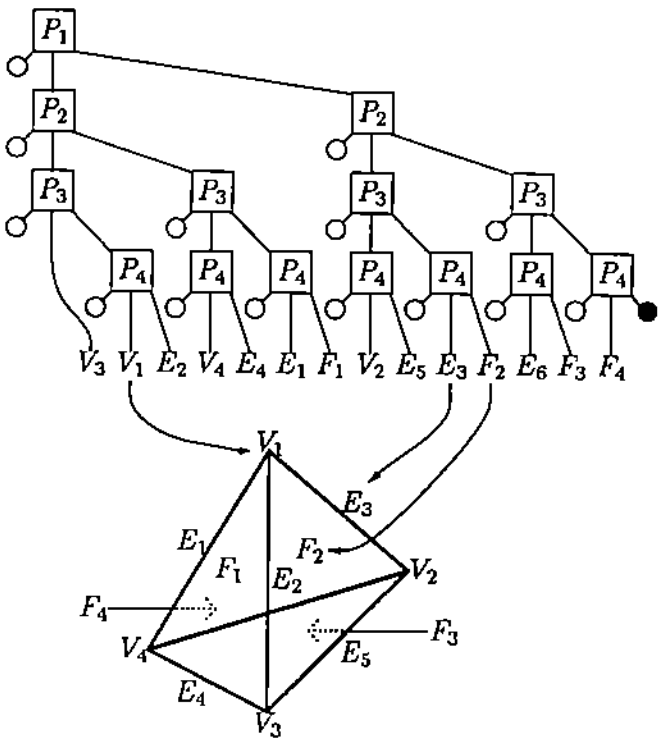

Figure 2: Example Brep index for a tetrahedral solid. The white circles indicate outside regions; the black circle indicates the single inside region.

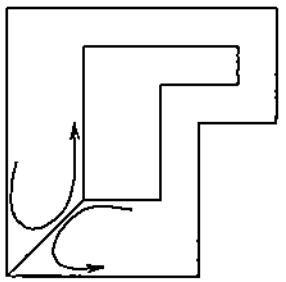

Figure 3: The edge-loops of a face with a hole connected via a bridge edge.

with maximum depth of 128 and average depth of 54; such a tree uses only cut planes that are the support planes of the faces. With the addition of well chosen initial cuts that are not support planes of the faces, the maximum depth drops to 19 and the average depth to 8.

Conceptually, the MSP tree spatially orders the vertices, edges and faces yielding a sublinear access time to each. To classify a point against the solid, the point is passed down the tree beginning at the root; the leaf node represents the region of space containing the point and along with that region, it tells us exactly either what boundary entity contains that region or whether it is inside or outside the solid. Thus the cost is proportional to the average height of the MSP tree.

3 Boundary Representation

The intent of *Proxima* is not to be a solid modeler providing all the expected operations for creating and manipulating objects such as Boolean operators [5]. The intent of *Proxima* is to provide a fast classification and distance computation support for applications dealing with already created objects. This is reflected in *Proxima*'s representation. For a full solid modeler, a highly dynamic and linked structure is required to facilitate the changes in the model. In *Proxima*, a highly compact and static representation is sufficient.

3.1 Internal Representation

The boundary of each object is stored in a classic hierarchical graph structure consisting of vertices, edges, faces and solids, and various topological adjacency information between them. There are no shells and loops, commonly found in boundary representations. Information about the faces making up a shell can be obtained in linear time by traversing and marking the boundary. Thus it is not necessary to explicitly store the face partitions making up the shells. On the other hand, multiconnected faces which contain one or more holes need to be maintained either by explicitly keeping all the edge loops

associated with the face, or by introducing bridge edges (refer to Figure 3). A bridge edge connects two edge loops of a single face (say, the outside edge loop and one of the holes) creating a single edge loop. In *Proxima* we elected to use bridge edges. There are two reasons for this choice. The first is that the internal representation is simpler, since we do not have to explicitly deal with or recognize the holes in the faces. The second is that bridge edges occur naturally in the process of decomposition during the construction of the Brep-index.

3.2 External File Formats

Proxima currently uses two external file formats. One is called the *Polyfile* format and the other the *Proxima* format.

3.2.1 Assumptions

Proxima assumes that the input polyhedron is geometrically consistent. The points of different polygons that should be coincident must be strictly equal. Also, the points of a face must be coplanar within some small epsilon value (we assume $\epsilon = 10^{-10}$). That is, given the support plane of a face as $P(x, y, z) = Ax + By + Cz + D$, then all points (x, y, z) on the face must satisfy $|P(x, y, z)| \leq \epsilon$.

It is also assumed that the set of polygons is topologically consistent with each polygon oriented in a counterclockwise direction when viewed from above the face and outside the solid, and together, the polygons enclose a single finite volume of space.

Finally, we assume that the solids are manifold. Nonmanifold objects may be introduces in the future.

If the polyhedra are generated by a solid modeling system rather than pieced together by some adhoc means, then these conditions should be satisfied.

3.2.2 PolyFiles

The Polyfile format is the most basic format one can devise for representing polyhedra. It gives the polygons bordering the polyhedron. Each polygon is given by the number of points followed by the coordinates of the points listed in a counter-clockwise order when viewed from the outside and directly above the polygon. The coordinates are read in double-precision format. For example, the tetrahedron shown in Figure 2 comes in the following Polyfile.

```
3
0 0 0
0 1 0
1 0 0
3
0 0 0
```

```
0 0 1
0 1 0
3
0 0 0
1 0 0
0 0 1
3
0 1 0
0 0 1
1 0 0
```

3.2.3 Proxima Files

A *Proxima* file consists of three parts: the extent, the boundary representation and the MSP tree. The boundary part consists of the set of vertices, edges and faces. A vertex is given by the vertex label and its coordinates. An edge is given by the edge lable and the label of the two bordering vertices. A face is given by the face label, the number of bordering edges, and the edge labels in counter-clockwise order; a negative edge number indicates the a reversed edge for the face. The MSP tree is specified in a preorder traversal of the ternary tree. Each node is given either by a descriptor C *A B C D* for a cut node with cut plane specified by its four coefficients, OUT for outside, IN for inside, V *n* for vertex *n*, E *n* for edge *n*, and F *n* for face *n*. The following *Proxima* file represents a tetrahedron which corresponds to the polyfile shown above.

```
BOX_EXTENT 0 0 0 1 1 1
/* Boundary Representation for a Tetrahedron: */
NVERTICES 4 /* Adjacent Edges: */
 1  0 0 0   /* 1 3 4 */
 2  0 1 0   /* 1 2 5 */
 3  1 0 0   /* 2 3 6 */
 4  0 0 1   /* 4 5 6 */
NEDGES 6    /* Adjacent Faces: */
 1  1 2     /* 1 2 */
 2  2 3     /* 1 4 */
 3  3 1     /* 1 3 */
 4  1 4     /* 2 3 */
 5  4 2     /* 2 4 */
 6  3 4     /* 3 4 */
NFACES 4
 1  3   1  2  3
 2  3   4  5 -1
 3  3  -3  6 -4
 4  3  -5 -6 -2
/* MSP Tree, shown here in three columns: */
```

```
C 0.58 0.58 0.58 -0.58      C 0 0 -1 0          IN
OUT                         OUT
C 0 0 -1 0                  C 0 -1 0 0
OUT                         OUT
C 0 -1 0 0                  C -1 0 0 0
OUT                         OUT
V 3                         V 1
C -1 0 0 0                  E 3
OUT                         C -1 0 0 0
V 2                         OUT
E 2                         E 1
C 0 -1 0 0                  F 1
OUT                         C 0 -1 0 0
C -1 0 0 0                  OUT
OUT                         C -1 0 0 0
V 4                         OUT
E 6                         E 4
C -1 0 0 0                  F 3
OUT                         C -1 0 0 0
E 5                         OUT
F 4                         F 2
```

Before the files are read, they are filtered throught the C preprocessor, so that the file may contain `/* comments */` and macro definitions.

4 Class Definitions

This section describes the public access layer of *Proxima*. There are many internal classes which are not intended to be accessed directly by application programs and therefore they are not described here. First, some of the basic data types are given such as Index, Counter, Boolean, Classification and Region classes. Then the geometric and topological classes are presented. This includes the Solid, Face, Edge, and Vertex topological classes, and the the Point, Vector, Plane, Extent and Matrix geometric classes. Finally, the topological access functions and looping macros for traversing the boundary representation are given at the end of the section.

4.1 Basic Types

Proxima uses several basic types, such as the Boolean, the Counter, and the Index. These help focus the intended use of integers within *Proxima*.

- `enum Boolean {FALSE=0, TRUE=1}`

- `typedef unsigned int Counter` *Used for counting*
- `typedef unsigned int Index` *Used for array indices*

4.1.1 Classifications

The classification and distance operations describe the relationship between an entity and an object. The relationship may be a single value, a set of values, or an ordered list of values. The values are called *classifications* and they are returned in structures called *classifiers*.

A classification of a classifier indicates either *inside* meaning that the entity is inside the object, *outside* meaning that the entity is outside the object, or one of the entities of the solid, namely a vertex, an edge or a face.

- `enum classification {INSIDE, OUTSIDE, VERTEX, EDGE, FACE}`

The classification of the classifier can be obtained by:

- `classification Classifier::whatKind() const`

Given that a classifier is a vertex, edge or a face, the index of the entity is obtained by:

- `Index Classifier::whichOne() const`

Thus for example, if a classifer `c` is a vertex of solid `s`, the vertex is obtained by `s.vertex(c.whichOne())`. The next classifier in a set or in an ordered list can be obtained by:

- `const Classifier *Classifier::next() const`

The classifiers returned by *Proxima* functions are internal structures to *Proxima* and are automatically reused during the next attempt to classify another entity. The application program therefore cannot modify or deleted these lists. It can only refer to them up until the time of the next function that returns a classifier list.

4.1.2 Plane Classifications

The classification of entities against a plane determines where the entity is in relation to the plane.

- `enum Where {ABOVE, ON, BELOW, CROSSAB, CROSSBA}`

An entity is either above, on, or below the plane, or it may cross. A directed line, as an example, may cross from above to below, or from below to above.

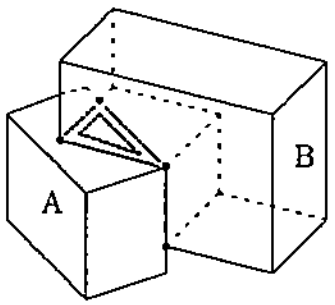

Figure 4: Objects A and B touch resulting in nine regions.

4.1.3 Regions

The classification of one solid in relation to another solid is given as a set of regions. Each region is either 0, 1 or 2-dimensional, and thus is coincident with a point, lies on a line, or lies in a plane accordingly. As such, the region's area is represented by a point, a line segment or a polygon. The number of points bordering the region `r` is `r.nPoints()`, and the ith point is `r.point(i)`. Furthermore, since a region is a result of some part of the boundary of one object lying completely inside, outside, or on the boundary of the other object, a region keeps track of the two classifications as `r.classA()` and `r.classB()`. As an example, refer to the touching solids A and B in Figure 4. Their set-theoretic intersection (as opposed to a regularized set intersection) results in nine regions; four 0-dimensional, four 1-dimensional and one 2-dimensional.

Given a region, the following are the access functions:

- `Counter Region::dimension() const` — *Either 0, 1 or 2*
- `Classification Region::classA() const`
- `Classification Region::classB() const`
- `Index Region::labelA() const`
- `Index Region::labelB() const`
- `Counter Region::nPoints() const` — *Number of points*
- `Point &Region::point(const Index i) const` — *The ith point*
- `Region *Region::nextRegion() const` — *Next region of a set*

The value of `labelA` is defined only if the classification returned by `classA` is a vertex, an edge or a face—similarily for `labelB` and `classB`. Thus for example, given that a 1D region `r` (i.e., `r.dimension()`$= 1$) is a result of a face of solids `A` touching an edge of solid `B`, the two entities are obtained by `A.face(r.labelA())`, and `B.edge(r.labelB())`. See Section 4.3.4 for a use of regions.

4.2 Geometric Classes

The geometric classes consist of the point, the vector, the plane, the box extent and the matrix.

4.2.1 Class Point

A point represents three coordinates in 3-dimensional euclidean space.

Constructors:

- `Point::Point()`
- `Point::Point(double x, double y, double z)`
- `Point::Point(Point &p)`

Modifiers:

- `void Point::init(double x, double y, double z)`
- `Boolean Point::read(FILE *inf)` *Expects to see X Y Z*

The read returns TRUE if the point was read in and FALSE if not.

Access Functions:

- `double Point::x() const` *X coordinate of point*
- `double Point::y() const` *Y coordinate of point*
- `double Point::z() const` *Z coordinate of point*
- `Boolean Point::isZero() const`

Operators:

- `Point Point::operator = (Point &)` *Assignment*
- `Point Point::operator = (Point *)` *Assignment*
- `Point Point::operator - () const` *Negation*
- `Point Point::operator +=(Point &)` *Increment*
- `Point Point::operator -=(Point &)` *Decrement*
- `Point Point::operator + (Point &) const` *Addition*
- `Point Point::operator - (Point &) const` *Difference*
- `Boolean Point::operator ==(const Point &) const` *Equality*
- `Boolean Point::operator !=(const Point &) const` *Inequality*
- `Boolean Point::operator < (const Point &) const` *Less than*
- `Boolean Point::operator > (const Point &) const` *Greater than*

4.2.2 Class Extent

An extent is the smallest box, axis alligned, enclosing a set of entities. Extents serve the purpose of quickly determining the minimum rectangular region of space containing a set of points.

Constructors:

- `Extent::Extent()` *Initialize*
- `Extent::Extent(Point &pMin, Point &pMax)` *Minimum p, Maximum q*
- `Extent::Extent(Extent &e)` *Copy*

Access Functions:

- `const Point &Extent::Extent::pMin() const` *Minimum point*
- `const Point &Extent::Extent::pMax() const` *Maximum point*
- `Point Extent::center()` *Midpoint of Extent*
- `Boolean Extent::inside(const Point &p) const` *Is p inside?*

Operators:

- `Extent Extent::operator + (const Extent &e) const` *Union*
- `Extent Extent::operator + (const Point &p) const` *Union*
- `Extent Extent::operator += (const Extent &e)` *Enlarge*
- `Extent Extent::operator += (const Point &p)` *Enlarge*
- `Boolean Extent::operator || (const Extent &e) const` *Intersection*

4.2.3 Class Vector

The Vector class represents a vector in three-dimensional cartesian space.

Constructors:

- `Vector::Vector()`
- `Vector::Vector(Point &p)`
- `Vector::Vector(Point *p)`
- `Vector::Vector(Vector &v)`
- `Vector::Vector(double x, double y, double z)`

Functions:

- `Vector &Vector::normalize()` — *Make a unit vector*
- `double Vector::norm() const` — *Magnitude*
- `Vector cprod(const Vector &v1, const Vector &v2)` — *Cross product*
- `double dprod(const Vector &v1, const Vector &v2)` — *Dot product*

Operators:

- `Vector Vector::operator = (Point &p)` — *Assignment*
- `Vector Vector::operator = (Point *p)` — *Assignment*
- `Vector operator * (const double s, Vector &v)` — *Scalar product*
- `Vector operator / (Vector v, const double f)` — v/f

4.2.4 Class Matrix

This is a 4x4 transformation matrix.

Constructors:

- `Matrix()`
- `Matrix::Matrix(double a00, double a01, double a02, double a10, double a11, double a12, double a20, double a21, double a22)`
- `Matrix::Matrix(Matrix &m)`
- `Matrix::Matrix(Matrix *m)`
- `Matrix::Matrix(double a00, double a01, double a02, double a03, double a10, double a11, double a12, double a13, double a20, double a21, double a22, double a23, double a30, double a31, double a32, double a33)`

Operators:

- `Matrix Matrix::operator =(Matrix &m)` — *Copy*
- `Matrix Matrix::operator =(Matrix *m)` — *Copy*
- `double &Matrix::operator [](const Index i, const Index j)` — $M[i,j]$

Functions: To form a translation matrix:

- `void Matrix::translate(double x, double y, double z)`

To form a scale matrix:

- `void Matrix::scale(double x, double y, double z)`

To form a rotation matrix around either the `'x'`, `'y'`, or the `'z'` axis:

- `void Matrix::rotate(const int degrees, const char axis)`
- `void Matrix::rot(const double degrees, const char axis)`

Matrix multiplication, addition, difference are done by:

- `Matrix mulM(Matrix &m1, Matrix &m2)` $m1 * m2$
- `Matrix addM(Matrix &m1, Matrix &m2)` $m1 + m2$
- `Matrix subM(Matrix &m1, Matrix &m2)` $m1 - m2$

To tranform a vector by a transformation matrix:

- `Point Matrix::map(Point &p)` $p \cdot M$

4.2.5 Class Plane

An oriented plane is represented by a four tuple $[A, B, C, D]$. The normal vector of the plane is $[A, B, C]$, where Ax + By + Cz + D = 0 is true for all the points on the plane. We refer to the halfspace in the direction of the normal as *above* and the other as *below*. When supporting a face, because of the counterclockwise orientation of the edges around the face, above is also *outside* and below is *inside* the solid.

Constructors:

- ```
  Plane::Plane(double a = 0.0,
               double b = 1.0,
               double c = 0.0,
               double d = 0.0)
  ```
- `Plane::Plane(Plane *p)`
- `Plane::Plane(Plane &p)`
- ```
  Plane::Plane(Point &p1,
               Point &p2,
               Point &p3)
  ```

Functions: To access the four tuple $[A, B, C, D]$:

- `double Plane::a() const`
- `double Plane::b() const`
- `double Plane::c() const`
- `double Plane::d() const`

To return the normal vector:

- `Vector Plane::normal() const`

The signed distance between a given point and a plane is obtained such that if the given point is in the direction of the normal of the plane, the return value if positive. Otherwise, the return value is negative.

- `double Plane::sDistanceToPoint(const Point &p) const`

To read in the four tuple $[A, B, C, D]$ from a file:

- `Boolean Plane::read(FILE *inf)`

To compute the intersection point of the plane with a line passing through points p and q:

```
- const Point Plane::lineIntersection(const Point &p,
                                      const Point &q) const
```

or the distance from point p to the plane along the direction v:

```
- const double Plane::lineIntersection(const Point &p,
                                       const Vector &v) const
```

To test if a given point p is above, on or below the plane,

```
- Where Plane::whichSide(const Point &p) const
- Where Plane::whichSide(const Point &q,
                         const Vector &v,
                         const double t) const
```

given that $p = q + t \cdot v$. To test if a given line segment is above, on, below or cross the plane:

```
- Where Plane::whichSide(const Point &p1,
                         const Point &p2) const
```

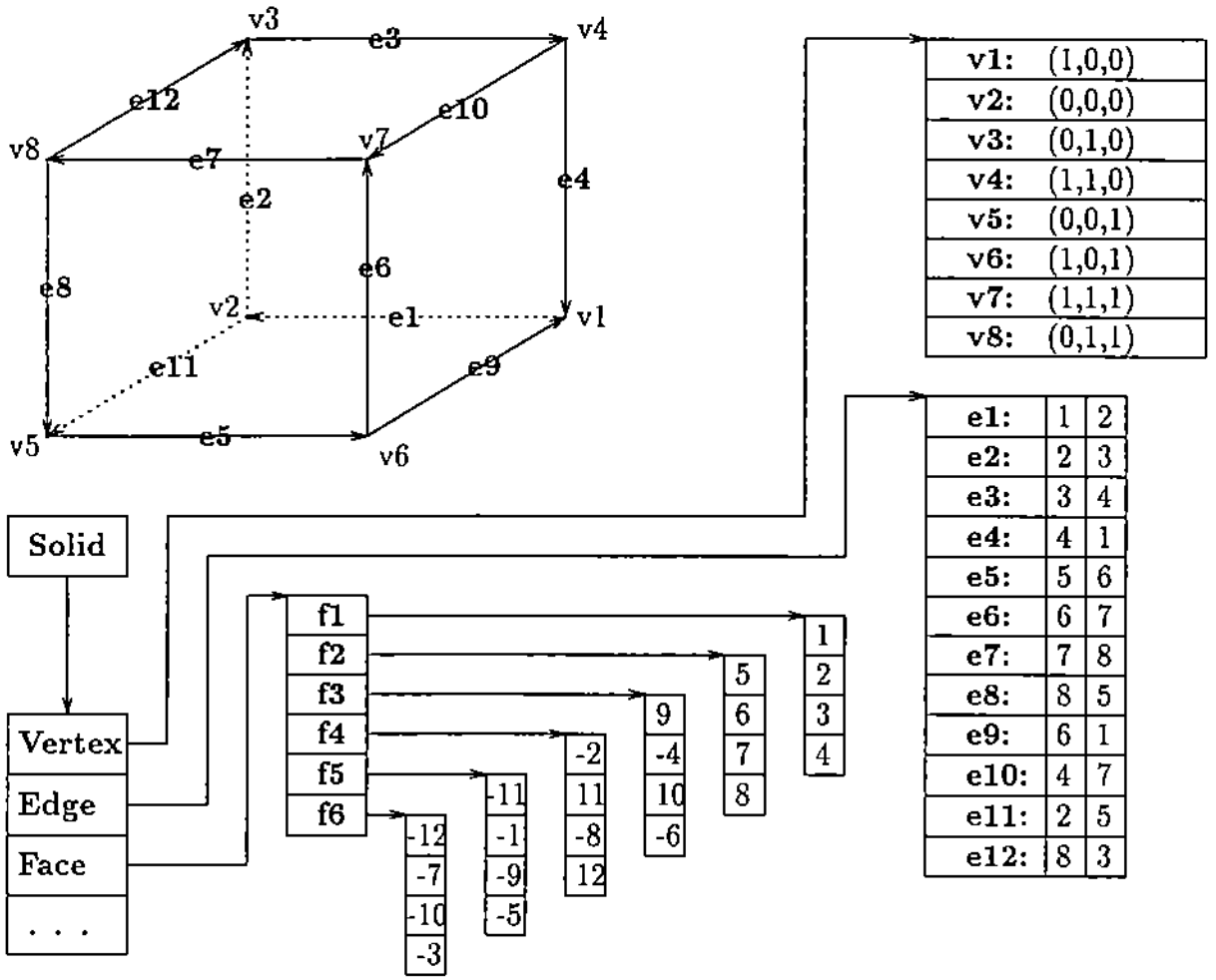

Figure 5: Boundary representation of a block object.

4.3 Topology Classes

A Solid is represented by its boundary. The boundary is made up of sets of faces, edges, vertices and other attributes. Figure 5 shows how the topological information is stored internally. A Solid keeps the vertices, edges and faces in three arrays. Given one of these entities, an adjacent entity is referred to by indexing into the appropriate array in the Solid. All the indices of adjacent entities are kept in arrays.

4.3.1 Class Vertex

A vertex is a 0-dimensional manifold entity bordering incident edges and faces. We only maintain the adjacent edges as the faces can easily be obtained from the face-adjacency information stored in the edges. The constructor and access functions for a vertex are:

- `Vertex::Vertex(Vertex &v)` — *Constructor*
- `double Vertex::x() const` — *X Coordinate*
- `double Vertex::y() const` — *Y Coordinate*
- `double Vertex::z() const` — *Z Coordinate*
- `Counter Vertex::nEdges() const` — *Number adjacent edges*
- `const Edge &Vertex::edge(const Index i) const` — *The ith adj. edge*
- `Index Vertex::edgeLabel(const Index i) const`

Here `v.edge(i)` is the ith edge adjacent to vertex `v`. The edge's index is `v.edgeLabel(i)`, so that `s.edge(v.edgeLabel(i))` is the same as `v.edge(i)`.

4.3.2 Class Edge

An edge is a 1-dimensional manifold entity embedded in a straight line. Figure 6 shows a diagram of the edge along with the access methods for the adjacent topological entities.

- `Edge::Edge(Edge &e)` — *Constructor*

An edge is bordered by two vertices called `vertexA` and `vertexB` and two faces called `faceA` and `faceB`.

- `const Vertex &Edge::vertexA() const` — *Source Vertex*
- `const Vertex &Edge::vertexB() const` — *Destination Vertex*
- `Index Edge::vertexALabel() const`
- `Index Edge::vertexBLabel() const`

Given that this is an edge of solid `s`, that `s.vertex(e.vertexALabel())` is the same as `e.vertexA()`, and `s.vertex(e.vertexBLabel())` is the same as `e.vertexB()`.

The two faces are:

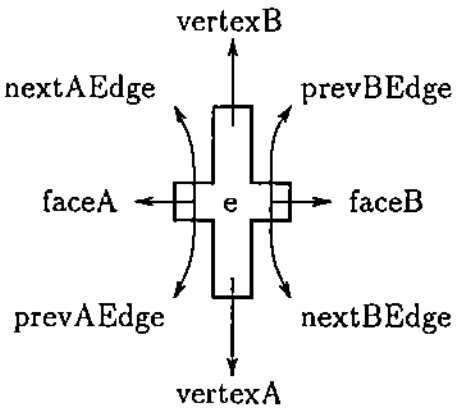

Figure 6: Diagram of a winged-edge representation used by *Proxima.*

- `const Face &Edge::faceA() const`
- `const Face &Edge::faceB() const`
- `const Index Edge::faceALabel() const`
- `const Index Edge::faceBLabel() const`

Note that `s.face(e.faceBLabel())` is the same as `e.faceB()`, and that `s.face(e.faceALabel())` is the same as `e.faceA()`.

There is a next edge and a previous edge for both adjacent faces since the edges around the faces are kept in a counterclockwise order.

- `const Edge &Edge::nextAEdge() const`
- `const Edge &Edge::prevAEdge() const`
- `const Edge &Edge::nextBEdge() const`
- `const Edge &Edge::prevBEdge() const`

The above edges can also be obtained by indexing the particular edge of `faceA` or `faceB`.

- `const Edge &Edge::nextAIndex() const`
- `const Edge &Edge::prevAIndex() const`
- `const Edge &Edge::nextBIndex() const`
- `const Edge &Edge::prevBIndex() const`

Thus for example, `e.nextAEdge()` is the same as `e.faceA.edge(e.nextAIndex())`.

Finally, for we can find out if an edge is a bridge edge by checking if faceA and faceB of the edge are the same face:

- `Boolean Edge::isABridge() const`

4.3.3 Class Face

A Face is a 2-dimensional bounded portion of a plane bordered by an alternating sequence of edges and vertices.

- `Face::Face(Face &f)` *Constructor*

Face `f` has `f.nEdges()` number of edges and the ith edge is `f.edge(i)`.

- `const Counter Face::nEdges() const` *Number of Edges*

- `const Edge &Face::edge(const Index i) const`
- `const Index Face::edgeLabel (const Index i) const`
- `const Vertex &Face::vertex(const Index i) const`
- `Index Face::vertexLabel(const Index i) const`

Note that for solid `s`, `s.edge(f.edgeLabel(i))` is the same as `f.edge(i)`.

Since the edges must be ordered counterclockwise around the face, an edge may be oriented in the wrong direction in terms of its source and destination vertices. The wrong orientation is recorded and the edge can be flipped as needed.

- `Boolean Face::flipEdge(const Index i) const`

To return the support plane of the face:

- `const Plane &Face::supportPlane()`

4.3.4 Class Solid

A solid represents a single polyhedron. Since *Proxima* cannot create nor manipulate solids, a new solid can only be read in from a file. Thus the only constructor for a solid takes a filename. The file can be either a polyfile or a proxima file (see Section 3.2).

- `Solid::Solid(char *filename)` *Load a solid*
- `void Solid::save(FILE *fp)` *Save to a Proxima File*

If a solid is loaded from a polyfile, its boundary representation is constructed but not its MSP tree. The MSP tree can be constructed on demand or automatically if needed. Saving a solid forces the construction.

- `void Solid::constructBrepIndex() const` *Make the Brep-Index*

To print out to file `fp` information about a solid and its MSP tree:

- `void Solid::describe(FILE *fp) const`

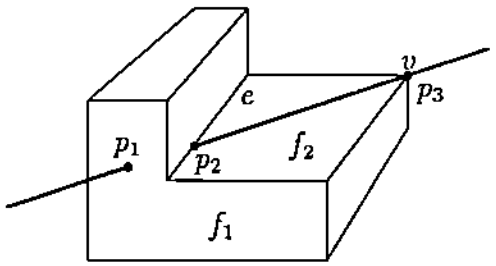

Figure 7: An example of a line/solid classification.

To return the extreem points of the box extent:

- `const Point &Solid::pMin() const` *Minimum Point*
- `const Point &Solid::pMax() const` *maximum Point*

The total number of vertices, edges or faces is:

- `Counter Solid::nVertices() const` *Number of Vertices*
- `Counter Solid::nEdges() const` *Number of Edges*
- `Counter Solid::nFaces() const` *Number of Faces*

The vertices, edges and faces are enumerated from one, not zero, and are obtained by:

- `const Vertex &Solid::vertex(const Index i) const` *Get ith vertex*
- `const Edge &Solid::edge(const Index i) const` *Get ith Edge*
- `const Face &Solid::face(const Index i) const` *Get ith face*

Point Classification: The classification of a point is the determination of whether the point is inside or outside the solid, or if its on the boundary what vertex, edge or face it lies on. To classify a point, start at the root of the tree and traverse a path from the root to a leaf. At each internal node n, the point is checked against the cut plane to determine whether the point is above, on, or below the plane. The path ends at a leaf representing a vertex, edge, or a face of the Brep, or the labels **inside** or **outside**. Accordingly, the search continues with the left, middle, or right child of n. Thus, the cost of classifying a point is proportional to the length of the path.

- `const Classifier *Solid::classify(const Point &p) const`

Line Classification: A line segment is classified as follows. Beginning at the root, the segment is checked against the cut plane. If the segment intersects, it is partitioned with the part above being forwarded to the left subtree, the intersection point to the middle subtree, and the part below the cut plane being sent to the right subtree. In

this way, the segment is filtered through the tree as appropriate. After each part of the induced segment partition is classified, the information is passed back up to the root and recombined. Each of the three results is a sequence of point or line segment classifications. After concatenating the three results, adjacent classifications that are the same are compressed into one. For example, the results

$$\ldots, [\mathbf{out}], [\mathbf{out}], [e], \ldots$$

are concatenated to form the sequence

$$[\ldots, \mathbf{out}, \mathbf{out}, e, \ldots] \tag{1}$$

and compressed, to yield

$$[\ldots, \mathbf{out}, e, \ldots].$$

As an example, Figure 7 shows a line segment classified against a solid resulting in the classification sequence:

$$[\mathbf{out}, f_1, \mathbf{in}, e, f_2, v, \mathbf{out}].$$

The line segment between points p and q can be classified by:

- `const Classifier *Solid::classify(const Point &p, const Point &q) const`

Solid Classification: The boundary of a solid can be classified in relation to another solid, by classifying all the entities of the solid using the Brep index for the other solid. The result is a set of regions, described in Section 4.1.3. The classification routine accepts a transformation matrix that maps the solid to be classified into proper prosition.

- `const Region *Solid::classify(Solid &s, const Matrix &m) const`

The above solid classification routine can be performed incrementally for each faces, edges or vertices.

- `const Region *Solid::classify(Vertex &v, const Matrix &m) const`
- `const Region *Solid::classify(Edge &e, const Matrix &m) const`
- `const Region *Solid::classify(Face &f, const Matrix &m) const`

Unlike the list of Classifiers which is maintained internally, the application program needs to delete the list of Regions when they are no longer needed. That is, the returned list of regions are dynamically allocated and become the responcibility of the application. The proper way to delete a list of regions is to call the following function.

- `void deleteRegions(Region *r)`

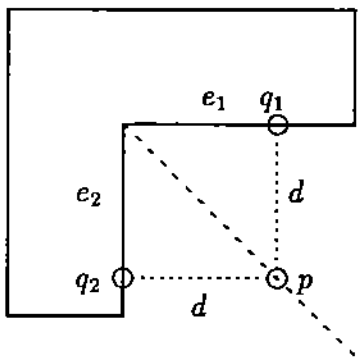

Figure 8: Edges e_1 and e_2 are the closest entities to point p at distance d with corresponding footpoints q_1 and q_2.

Distance Computation: The shortest distance d from a point p to the solid can result in one or more equidistant entities returned as a list of classifiers:

- `const Classifier *Solid::distance(const Point &p, double *d) const`

The foot point of an entity is the closest point on the entity, as illustrated in Figure 8.

- `const Point &Solid::footPoint(const Point &p, const Classifier *c) const`

4.4 Properties

Proxima provides a simple mechanism to assign properties to the vertice, edge, face and solid classes. It does this by setting aside a single pointer field in each entity. This field can be allocated, set, accessed and deleted at user control.

- `void` *entity*`::setProp(void *property)`
- `void*` *entity*`::getProp() const`

Since the property field does not know the type, you need to type cast the `getProp` result. For example, consider placing a label on each face indicating what shell it belongs to.

```
#define visited(i) (Boolean)(*(s.face(i).getProp()))
#define mark(F) (Boolean)(*F.getProp()) = TRUE

void markFacesOfShell(Solid &s)
```

```
{
  Index shell = 0;
  ForEachFaceOfS(fj,s)
    if(!s.face(fj).getProp())
      labelFace(s,fj,++shell);
}

void labelFace(Solid &s, Index fi, Index shellLabel)
{
  Index *sid = new(shellLabel);
  Face  &f   = s.face(fi);
  f.setProp(sid);
  ForEachEdgeOfFace(ei,f) {
    Edge &e   = f.edge(ei);
    Index afi = e.faceALabel() == fi
              ? e.faceBLabel()
              : e.faceALabel();
    if(!s.face(afi).getProp())
      labelFace(s,afi,shellLabel);
  }
}
```

5 Traversing the Boundary of a Solid

Each macro declares a local Index variable, say i, and then iterates variable i set to the index of each appropriate entity:

- `ForEachVertexOfS(i,s) {`
  `Vertex &v = s.vertex(i);`
  `...}` *Vertices of Solid*

In the loop iterating over all the vertices of a solid, variable i is set to values from 1 to `s.nVertices()`.

- `ForEachEdgeOfS(i,s) {`
  `Edge &e = s.edge(i);`
  `...}` *Edges of Solid*

- `ForEachFaceOfS(i,s) {`
  `Face &f = s.face(i);`
  `...}` *Faces of Solid*

- `ForEachEdgeOfV(i,v) {`
  `Edge &e = v.edge(i);`
  `...}` *Edges of Vertex*

- `ForEachEdgeOfF(i,f) {`
  `Edge &e = f.edge(i);`
  `...}` *Edges of Face*

As an example, the following code segment prints the vertex loops for each face of a solid.

```
ForEachFaceOfS(fi,s) {
   const Face &f = s.face(fi);
   ForEachEdgeOfF(ei,f) {
      const Edge &e = f.edge(ei);
      printf("%d ",
            f.flipEdge(ei) ?  e.vertexBLabel() :  e.vertexALabel())
   }
}
```

6 Examples

6.1 Converting Polyfiles to Proxima Files

In this first example, a Polyfile is converted to a Proxima file.

```
#include <stdio.h>
#include "proxima.h"
//
// usage: poly2prox [ infile.poly [ outfile.prox ] ]
//
int main(int argc, char **argv)
{
  Solid s(argc >= 2 ? argv[1] : (char *)0);
  s.constructBRepIndex();
  s.save(argc == 3 ? fopen(argv[2], "w") : stdout);
}
```

6.2 Point Classification

In this example, an interactive program repeatadly askes for the coordinates of a point and returns the classification of the point. Specifically, given a solid, it prints whether the point is inside or outside, or if the point is on the boundary, it returns what boundary entity the point is on.

```
#include <stdio.h>
#include "proxima.h"
// usage: testp <file.prox> OR testp <file.poly>
//
main(int argc, char **argv)
{
  Solid s(argc == 2 ? argv[1] : (char *)0);
  Point p;
  printf("Enter points as 'x y z', or ^D to quit.\nProxima> ");
  while(p.read(stdin)) {
    Classifier const *c = s.classify(p);
    printf("Point (%g,%g,%g) is ", p.x(), p.y(), p.z());
    switch(c->whatKind()) {
    case VERTEX :
      printf("Vertex %d.\n", c->whichOne());
      break;
    case EDGE :
      { const Edge &e = s.edge(c->whichOne());
        printf("on Edge %d, with vertices [%d,%d], and faces [%d,%d].\n",
               c->whichOne(), e.vertexALabel(), e.vertexBLabel(),
               e.faceALabel(), e.faceBLabel());
      }
      break;
    case FACE :
      { const Index fi = c->whichOne();
        const Face &f  = s.face(fi);
        printf("on Face %d, with %d edges [", fi, f.nEdges());
        forEachEdgeOfFace(i,f) {
          if(i % 10 == 0)
            printf("\n\t\t\t");
          printf("%d ", f.edgeLabel(i));
        }
        printf("].\n");
      }
      break;
    case INSIDE :
      printf("inside.\n");
      break;
    case OUTSIDE :
      printf("outside.\n");
      break;
    }
    printf("proxima> ");
  }
```

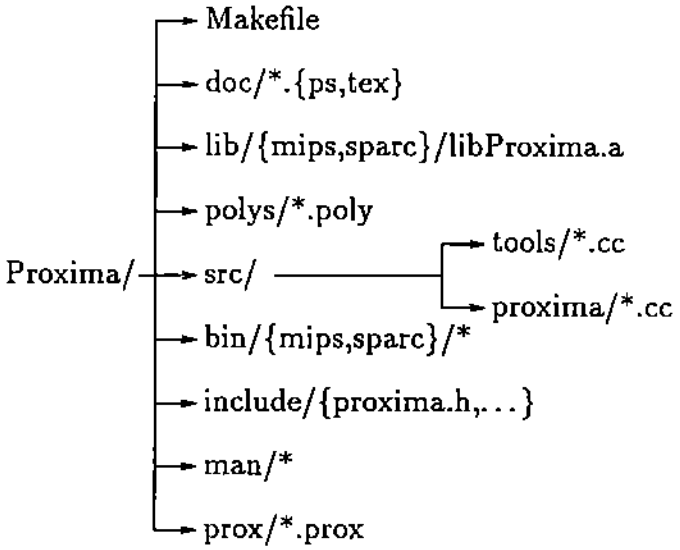

Figure 9: Proxima's directory structure.

```
  printf("\n");
}
```

7 System Overview

Proxima is set up in a directory structure shown in Figure 9. It contains the following:

- a Makefile for the entire system—this make file will recompile the source files for the *Proxima* library and all the tools;
- a documentation directory containing poscript and LaTeX file for this and other documents;
- a library directory;
- a directory of some sample polyhedron files;
- the source directory consisting of the *Proxima* source subdirectory and the sample tool subdirectory;
- an include-file directory containing all the C++ header files;
- a manual page directory; and

- a *Proxima* file directory containing the *Proxima* files for the polyfiles located in the other directory.

Since *Proxima* can be compiled for different architectures, the lib and the bin directories contain subdirectories for each architecture. We provide the Sparc and the SGI's Mips support, although others can be created by recompiling the entire system.

7.1 Obtaining Proxima by FTP

Proxima can be obtained from Purdue's Computer Science department's server by anonymous ftp. The internet address is

```
ftp.cs.purdue.edu [128.10.2.1]
```

After entering the ftp server, do the following:

```
cd pub/pse
binary
dir
get proxima-n.tar.Z
close
quit
```

where n is the latest version number you find in the directory. Once you copy the compressed tar file, do the following:

```
zcat proxima-* | tar xvf -
rm proxima-*.Z
```

7.2 Compiling Proxima

Proxima can be recompiled for a specific architecture by using the Makefile found in *Proxima*'s home directory. First the appropriate library in `lib/`, and all the object files in the source directories will be deleted, and then recompiled. Before you recompile *Proxima*, edit the Makefile and change the home directory line for

```
PROXIMA = /u/pse/proxima
```

to the path of the directory you copied proxima into, and set the proper architecture in the `CPUTYPE` field.

7.3 Compiling Applications

The following is an example of a `Makefile` for compiling the `poly2prox` tool given that the directory containing the makefile also contains the file `poly2prox.cc`:

```
.SUFFIXES: .cc .h
CPLUS   = g++
SWITCHES= -O
PROXIMA = /u/pse/proxima
CPUTYPE = mips
LIBS    = -L$(PROXIMA)/lib/$(CPUTYPE) -lProxima -lm
INCS    = -I$(PROXIMA)/include

poly2prox:  poly2prox.o
        $(CPLUS) $(SWITCHES) -s $@ $@.o $(LIBS)
depend:
        makedepend $(INCS) $(SRCS)
```

Proxima was developed using the Gnu C++ compiler. However, it will compile using the AT&T C++ compiler. Other compilers have not been tested.

Problems with *Proxima* can be forwarded to George Vaněček, Jr. at email

`vanecek@cs.purdue.edu`.

Also, if you use *Proxima* and let us know, we will inform you by email when new versions become available.

References

[1] W. Bouma and G. Vaněček, Jr. "Collision detection and Analysis in a Physical Based Simulation," *Proceedings of the Eurographics Workshop on Animation and Simulation*, 191–203, September 1991.

[2] B. Naylor. "Binary Space Partitioning Trees as an Alternative Representation of Polytopes," *Computer–Aided Design*, 250–252, 1990.

[3] G. Vaněček, Jr., "Brep-Index: A Multi-dimensional Space Partitioning Tree," *First ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAD Applications*, Austin Texas, 35–44, June 1991.

[4] G. Vaněček, Jr., *ProtoSolid: An inside look*, Purdue University, Department of Computer Science, CER–89–26, November 1989.

[5] G. Vaněček Jr., "Set Operations on Polyhedra using Decomposition Methods." PhD Thesis, University of Maryland, College Park, Maryland, June 1989.

[6] G. Vaněček, Jr., "A Data Structure for Analyzing Collisions of Moving Objects," *IEEE Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, Kailua-Kona Hawaii, Vol I:671–680, January 1991.

Index