V-Clip User's Manual

Brian Mirtich
mirtich@merl.com
MERL – A Mitsubishi Electric Research Laboratory

1 Introduction

The Voronoi Clip, or *V-Clip*, algorithm is a low-level collision detection algorithm for polyhedral objects. It is described in

> Brian Mirtich, "*V-Clip:* Fast and Robust Polyhedral Collision Detection," Technical Report TR97–05, Mitsubishi Electric Research Lab, Cambridge, MA, July 1997.

The *V-Clip* library is a C++ implementation of this algorithm, along with facilities for constructing and manipulating geometries. The source code is freely distributed for educational, research and non-profit purposes. Permission to use the library in commercial products may be obtained from MERL - A Mitsubishi Electric Research Lab, 201 Broadway, Cambridge, MA 02139.

The major design goals of *V-Clip* were robustness and efficiency. The implementation works well even in the presence of geometric degeneracies and contains no floating point tolerances for the user to tune. It compares favorably to other algorithms from an efficiency standpoint (see the tech report for details). In typical applications *V-Clip* computes the closest points between objects in almost constant time.

V-Clip operates on polyhedral objects which may be nonconvex or even disconnected. It returns the closest points between objects and the distances between them. If the objects penetrate, it returns a penetration depth. *V-Clip* requires the application to specify nonconvex or disconnected objects as hierarchies of convex pieces. When called on disjoint nonconvex objects, *V-Clip* returns a lower bound on the actual distance between the objects.

This manual describes the API to the V-Clip library and the file formats used by the default geometry reader. Instructions for building the library are contained in the Readme file of the top level directory.

1.1 Other collision detection alternatives

V-Clip is a low-level collision detection algorithm. The basic operation provided to the application is that of performing a collision check between two objects. The decisions of when to perform the checks between which object pairs are left to the application. In particular, the *V-Clip* library does not include facilities for higher level collision check culling, using bounding boxes or bounding spheres, for example. If this is required, consider writing one yourself, or consider using a complete collision detection package such as *I-Collide* (www.cs.unc.edu/~geom/I_COLLIDE.html) or *V-Collide* (www.cs.unc.edu/~geom/V_COLLIDE).

V-Clip is designed for objects that are bounded by closed surfaces, and not utterly nonconvex. If you need to detect collisions within polygon soup or between intricately intertwined objects, there are better alternatives: consider using *Rapid* (www.cs.unc.edu/~geom/OBB/OBBT.html).

When objects penetrate, *V-Clip* returns a penetration depth, such as the distance between a face and a vertex that lies below it. The features used in the penetration analysis are usually the expected ones, however there are no guarantees about which features will be used. For this reason, penetration depths might change discontinuously as penetrating objects move through each other. This problem is unlikely to occur if penetration is slight, or if penetration initiates backtracking to find a collision point. If a more well defined measure of penetration depth is needed, consider using the *Enhanced Gilbert-Johnson Keerthi algorithm* (www.comlab.ox.ac.uk/oucl/users/stephen.cameron/distances.html).

1.2 STL, Qhull, and Inventor

The *V-Clip* library relies extensively on the C++ Standard Template Library (STL) for container classes, such as lists, vectors, and hash tables. There are many good books and online resources on STL, as well as publicly available implementations (www.sgi.com/Technology/STL).

V-Clip should be compiled with the publicly available *Qhull* library (www.geom.umn.edu/software/qhull). This library provides a convex hull routine that is used by *V-Clip* to build polyhedra from vertex sets and to build enclosing hulls of sets of polyhedra. *V-Clip* can be built without *Qhull*, but then it will not handle nonconvex objects.

If *V-Clip* is compiled with the SGI *Inventor* library, a function is provided to translate from *V-Clip's* geometric data structures to *Inventor* models.

2 Polyhedra, PolyTrees, and default geometry reader

The fundamental data structures used by *V-Clip* are class Polyhedron and class PolyTree. A Polyhedron is a single convex solid, with topological and Voronoi region information. PolyTrees are hierarchical descriptions of convex or nonconvex geometry; they use class Polyhedron internally. The depth of a PolyTree and the branching factors at the individual nodes are arbitrary. The leaves of a PolyTree correspond to the underlying convex pieces of the solid. The application must specify the decomposition—it is not done automatically—but facilities are provided to support efficient processing of the hierarchy. For instance, the convex hulls of appropriate subsets of the complete geometry are automatically computed at all of the internal tree nodes. A trivial PolyTree comprising a single node represents a convex object, and is called an atomic PolyTree. A PolyTree that is not atomic is compound.

For convenience, the *V-Clip* library includes a default geometry reader that reads Polyhedra and PolyTrees definitions from simple ascii files. This section presents this default reader and the file format it uses. Section 4 describes the API that can be used to design new readers or to build up geometry programatically.

The default reader hides class Polyhedron from the application, which need deal only with atomic and compound PolyTrees. The interface to the application is a single function:

- `int loadPolyTreeFile(const char *fname, PolyTreeLibrary &library)`
  Read an entire geometry file specified by given filename, which may be a pathname. Add all defined atomic and compound PolyTrees to the specified PolyTreeLibrary. Return the number of PolyTrees read in.

PolyTreeLibraries are described in Section 3. They organize the geometry database and provide memory management. PolyTrees that have been added to a library can be instantiated by the application. The remainder of this section describes the format of the geometry files for the default reader. Several example files may be found in the `ptrees` directory of the distribution.

2.1 Atomic PolyTree file format

The specification of an atomic Polytree begins with the keyword `atomic` followed by a name for the PolyTree. Then the vertices are listed in any order, one per line. Each vertex line contains a name for the vertex, followed by its three spatial coordinates. The fields are delimited by whitespace. The coordinates may be specified as integers or floating point numbers, and explicit plus signs are optional for positive values. A single asterisk character (`*`) on a separate line terminates the vertex list.

Next comes the list of faces, again specified one per line. Each face line contains a name for the face, followed by a list of the vertices on the face. The vertices in a face list are specified by their names defined earlier. The fields in a face line are delimited by whitespace, and must appear on a single line. There is no theoretical limit on the number of vertices per face. *The vertices must be listed in counter-clockwise order* from the perspective of a viewer standing outside the polyhedron, looking at the face. The face list is terminated with an asterisk character on its own line.

Edges are never specified or named explicitly. They are constructed automatically, and their names derive from the names of the vertices at their endpoints. An edge between vertex *here* and vertex *there* is named *here:there*. Figure 1 shows the specification of a cube.

```
atomic cube
ne0     +1 +1 -1
nw0     -1 +1 -1
sw0     -1 -1 -1
se0     +1 -1 -1
ne1     +1 +1 +1
nw1     -1 +1 +1
sw1     -1 -1 +1
se1     +1 -1 +1
*
bottom  sw0 nw0 ne0 se0
top     sw1 se1 ne1 nw1
south   sw0 se0 se1 sw1
north   ne1 ne0 nw0 nw1
east    se0 ne0 ne1 se1
west    sw0 sw1 nw1 nw0
*
```

Figure 1: Atomic Polytree specification of an origin-centered cube, two units on a side.

2.1.1 Shortcuts

The names given to vertices and faces are helpful documentation and useful in debugging, however assigning names is burdensome when a polyhedron has many features. Default names alleviate this problem. Omiting the name on a vertex line generates a default vertex name. Unnamed vertices are automatically named *v0, v1, v2, ...*. These default names can be used in face lists just as any other vertex name. Placing a dash (-) in the name field of a face line generates a default face name, taken from the sequence *f0, f1, f2, ...*.

The vertices specified in a face line can be referred to by name, as described above, or by integers. An integer n in a face line specifies the nth vertex, counting from zero. This feature works regardless of whether default vertex names are used or not. A vertex referred to by its integer index can be found without having to compare string names. For this reason, reading large atomic PolyTrees is much faster when the vertices are referred to by integers in the face lines. Explicit vertex names should not begin with a digit as the reader interprets such strings as vertex indices.

As an additional shortcut, an atomic PolyTree can be defined soley by its vertices. If no faces are listed between the asterisk terminating the vertex list and the asterisk terminating the face list, the faces are computed from the convex hull of the vertex set. The faces are named *f0, f1, f2, ...*, and their ordering can not be controlled. If any specified vertices do not lie on the convex hull, they are automatically eliminated from the vertex list. This shortcut is only available when the *Qhull* library is included in the compilation. Computing faces automatically may produce a slightly different geometry than when they are explicitly specified, if any explicitly specified faces have more than three sides. The convex hull routine may triangulate some of these faces due to floating point error. This should not affect the operation of *V-Clip*. Figure 2 shows various ways of defining a tetrahedron.

2.2 Pose file format

Poses play an important tole in specifying compound PolyTrees. A pose is a member of the group $SE(3)$, in other words, the combination of a three-dimensional translation and rotation. The relative positions and orientations of two bodies (or two frames) in three-dimensional space can be described by a pose. See any basic text on 3D graphics for more details.

The file format for specifying poses is based on elementary transformations. An elementary transformations is either a pure translation or a pure rotation. A translation is specified by the keyword `trans` followed by three numbers specifying the displacement of the translation. For example,

```
trans -3 0 2.0
```

```
atomic tetra
origin    0 0 0
x-pt      1.5 0 0
y-pt      0 +3 0.0
z-pt      0 0 2.1
*
yz-face    origin z-pt y-pt
xy-face    origin y-pt x-pt
zx-face    origin x-pt z-pt
slant-face x-pt y-pt z-pt
*
```

```
atomic tetra
origin 0 0 0
  1.5 0 0
  0 +3 0.0
  0 0 2.1
*
-  origin v2 v1
-  origin v1 v0
-  origin v0 v2
slant-face 1 2 3
*
```

```
atomic tetra
0 0 0
1.5 0 0
0 +3 0.0
0 0 2.1
*
*
```

Figure 2: Three different specifications of the same first octant tetrahedron. *Left:* Most verbose specification. *Center:* Using default names for some of the vertices and faces. The vertices around the last face are referred to by integer index. *Right:* Tersest possible specification uses default names for all vertices, and the faces are automatically generated from the convex hull.

specifies a translation by the vector $(-3, 0, 2)^T$. A rotation is specified by the keyword `rot`, followed by an angle *specified in degrees* and an axis about which to rotate. For example,

`rot 20 1 2 3`

specifies a rotation of 20 degrees about the axis containing the vector $(1, 2, 3)^T$. The specified axis need not be unit magnitude; it is automatically normalized. A right-handed convention is followed. The inverse of the rotation shown above could be expressed as `rot 20 -1 -2 -3` or `rot -20 1 2 3`. The abbreviations `i,` `j,` and `k` may be substituted for the canonical unit vectors along the x, y, and z axes, respectively. These abbreviations may also be prefixed signs, with no intervening white space. For example, `trans -j` specifies a translation by one unit along the negative y-axis, and `rot 90 +k` specifies a rotation by 90 degrees about the positive z-axis (the plus sign is optional). Poses are built from a series of simple translations and rotations, surrounded by curly brackets. For example:

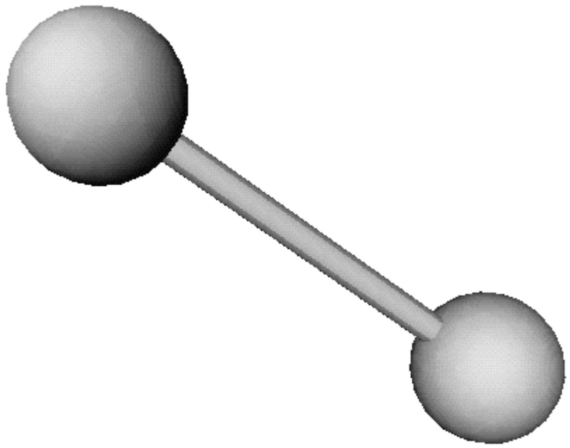
{`trans -1 -1 -1 rot 22.5 -k trans 5 0 0`}.

When building sequences of elementary transformations, it is imperative to keep in mind whether the transformations are specified relative to a body frame or a fixed frame. When the composite transformation is read left to right, the body frame is being used. Imagine an object with a set of coordinate axes attached to it, and initially placed such that this body frame exactly coincides with a set of axes fixed in space. The above transformation is interpreted as translating the body along the vector $(-1, -1, -1)$ in the body frame, then rotating by 22.5 degrees about the body frame's negative z-axis (which is no longer colinear with the fixed frame's z-axis after the translation), and finally translating five units along the body frame's x-axis (which is in a different direction than the fixed frame's x-axis, after the prior rotation).

Alternatively, the composite transformation can be interpreted in the fixed frame by reading from right to left. Beginning again from the point where the body and fixed frames coincide, the object is translated by five units along the fixed frame's z-axis, then rotated about the fixed frame's negative z-axis, and finally to translated by the vector $(-1, -1, -1)$ in the fixed frame. Unlike the body frame, the fixed frame never moves as a result of previous elementary transformations. It doesn't matter whether the body frame or the fixed frame interpretation is used. In either case, the final position of the object is the same. Use whichever interpretation is more convenient for the situation at hand, but know which interpretation you are using and order the transformations accordingly.

2.3 Compound PolyTree file format

The specification for a compound PolyTree begins with the keyword `compound` followed by a name for the PolyTree and an opening curly bracket. Following this is an alternating list of poses and names of previously defined Polytrees. Each pose specifies the position and orientation of the subsequent PolyTree relative to the



```
compound barbell {
   {trans -6 0 0}  ball
   {trans +6 0 0}  ball
   { }  shaft
}
```

Figure 3: Specification of a barbell compound PolyTree.

frame of the compound PolyTree being defined. For example, suppose two atomic PolyTrees have already been defined: `ball`, a unit radius sphere, and `shaft`, a thin cylinder centered at the origin and lying along the x-axis. Then a barbell PolyTree could be defined by the specification in Figure 3. The empty pose specification for the shaft specifies the identity transformation; no translation or rotation is necessary.

The component PolyTrees specified in a compound PolyTree definition need not be atomic.[1] For instance, after the barbell PolyTree is defined, one can use it to build up a jack geometry (Figure 4). As a final example, Figure 5 defines the complete geometry for a *sixspike*: an object obtained by attaching the square bases of six pointy pyramids to the faces of an imaginary cube.

3 Storage of geometry and PolyTreeLibraries

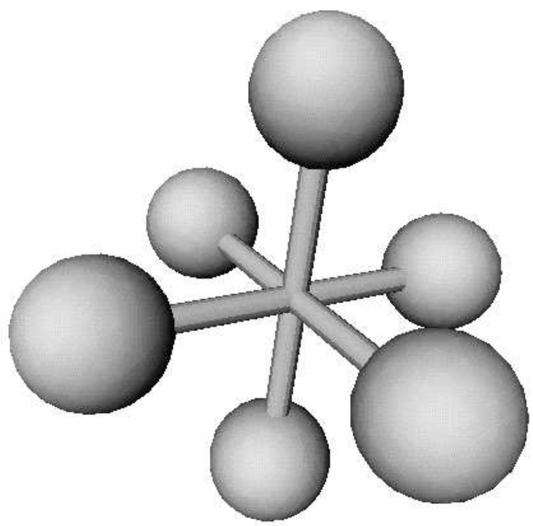
A central principle of the geometry data storage is that PolyTrees are replicated while Polyhedra are shared. If existing PolyTree A is used as a building block in PolyTree B, then all of the PolyTree nodes of A are duplicated for incorporation into B's structure. Likewise, if an application later instantiates two instances of PolyTree B, a separate copy of all of B's PolyTree nodes are created for each instance. On the other hand, the Polyhedra pointed to by A's PolyTree nodes are not duplicated when A is incorporated into B. Rather, B's PolyTree nodes point to the same Polyhedra as A's. Instances of PolyTrees A or B that are instantiated by the application also share the same Polyhedra structures. As an example, Figure 6 depicts the geometry storage after a jack geometry (see previous section) is read by the default reader.

A more memory efficient scheme would make the PolyTree nodes shared instances, just like the Polyhedra. The PolyTree hierarchy would be a digraph rather than a forest. While the replication of PolyTree nodes seems wasteful, there are good reasons for doing it.

1. Replicating PolyTree nodes allows each PolyTree node to store its pose relative to the node at the root of the tree. This would not be possible if PolyTree nodes were shared, since PolyTrees A and B might both include PolyTree C, but in different relative positions. Having this information at each node reduces the computations that must be performed in certain algorithms, like *V-Clip*, that traverse the hierarchy.

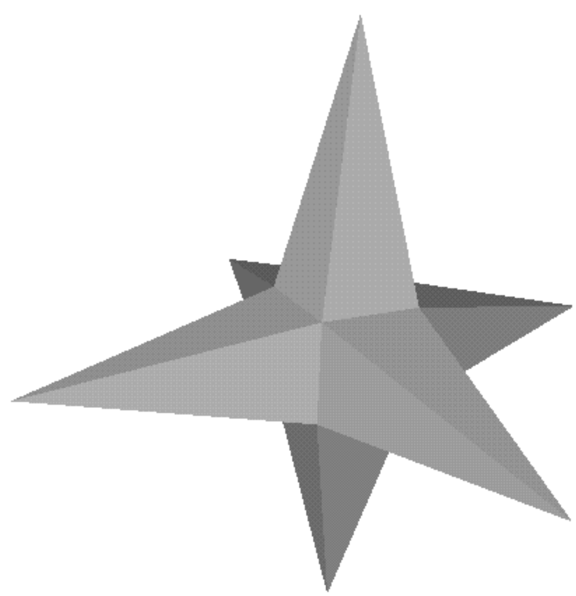
2. Replicating PolyTree nodes allows an algorithm to refer to a particular instance of the node with a single scalar—an address. *V-Clip* must cache the closest features between different objects. But these objects might have the same geometry. A portion of the geometry might even be replicated within the

[1] Actually, the component PolyTrees need not even be previously defined. The compound PolyTree reader is recursive, and supports nested descriptions of PolyTrees. See the file `ptrees/ring2.pt` for an example.



```
compound jack {
   { }  barbell
   {rot 90 k}  barbell
   {rot 90 j}  barbell
}
```

Figure 4: A jack geometry described in terms of barbell geometries.



```
atomic spike
b1  +1 +1 0
b2  -1 +1 0
b3  -1 -1 0
b4  +1 -1 0
point  0 0 5
*
base  b4 b3 b2 b1
-  b1 b2 point
-  b2 b3 point
-  b3 b4 point
-  b4 b1 point
*

compound six-spike {
 {trans +k}           spike
 {trans -k rot 180 i} spike
 {trans +i rot +90 j} spike
 {trans -i rot -90 j} spike
 {trans -j rot +90 i} spike
 {trans +j rot -90 i} spike
}
```

Figure 5: Complete specification of a sixpoint PolyTree.

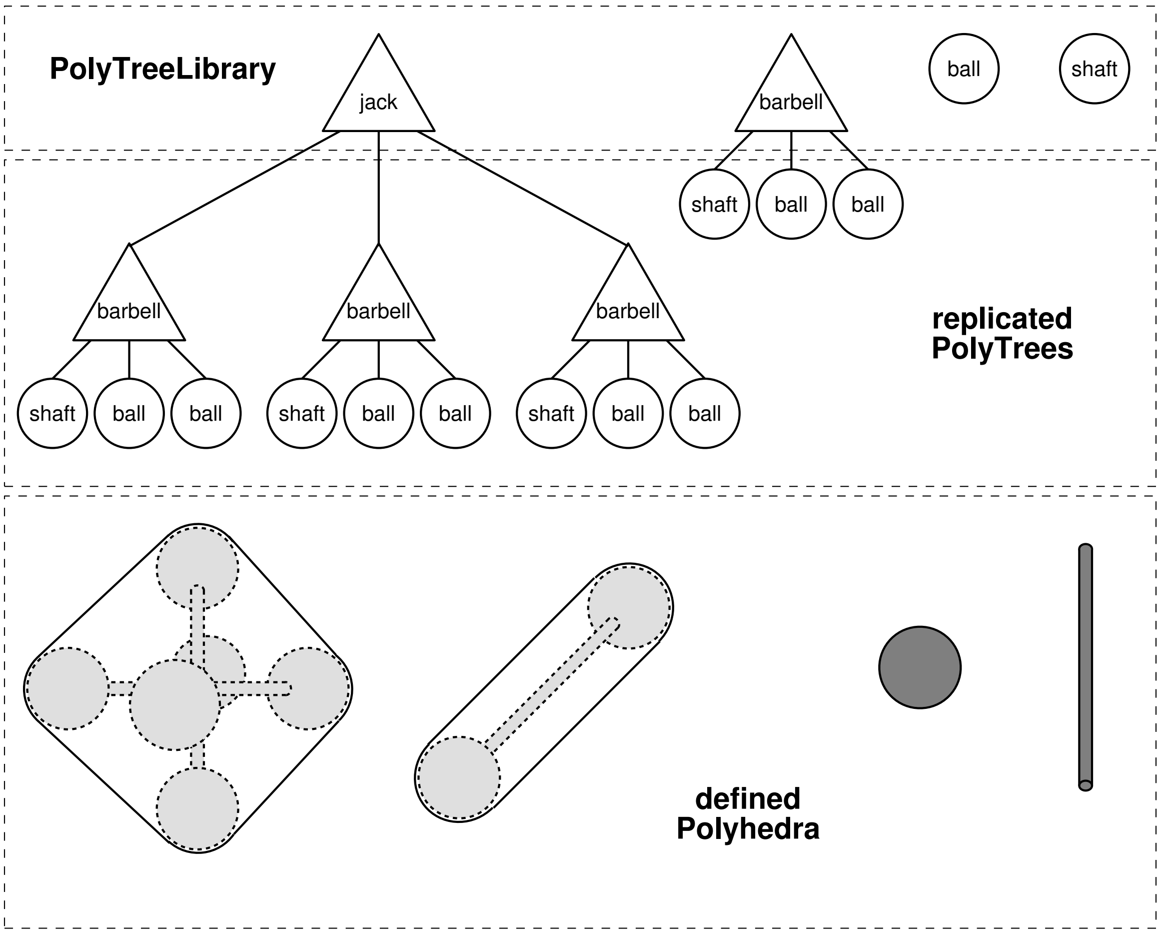


Figure 6: Geometry storage after reading a file defining the jack geometry. Circular nodes are atomic PolyTrees, and triangular ones are compound PolyTrees. Each atomic or compound PolyTree node points to one of the four Polyhedron geometries shown. The atomic PolyTree nodes point to Polyhedra explicitly defined in the file, while the compound PolyTree nodes point to Polyhedra automatically generated as convex hulls of sets of Polyhedra. Every PolyTree node shown is individually stored, while the Polyhedron structures are shared among all nodes.

same object. Replicated PolyTree nodes provide a way of referring to the *first* barbell of jack A, as opposed to the second or third barbell of jack A, or the barbells of any other jack object.[2]

3. Finally, PolyTree nodes are generally small compared to Polyhedra, which store the bulk of the geometric data. The only data stored at the PolyTree nodes are the aforementioned relative poses, a pointer to a Polyhedron structure, pointers to children nodes in the hierarchy, and some useful geometric measures such as radius and volume.[3]

Class PolyTreeLibrary is used to organize the geometry and provide basic memory management. Think of it as an ordered list of PolyTrees. Each PolyTree is the root of a (possibly trivial) PolyTree hierarchy describing some geometry. Applications can add geometries to a library, and then instantiate instances of geometries for use by *V-Clip*. The default reader adds geometries to a library as they are read and allows geometries already in the library to be used as building blocks for creating more complicated geometries. PolyTrees that are added to a PolyTreeLibrary are roots of hierarchies, and only these PolyTrees may be instantiated. Access to the internal, replicated PolyTree nodes is not provided. Class PolyTreeLibrary has the following public methods:

- `void PolyTreeLibrary::clear()`
  Remove all PolyTrees from a library, freeing memory.

- `int PolyTreeLibrary::size()`
  Return the number of PolyTrees defined in the library. For the example of Figure 6, this function returns 4.

- `void PolyTreeLibrary::add(PolyTree *pt)`
  Add a PolyTree to the library, putting it at the front of the list.

- `const PolyTree *PolyTreeLibrary::lookup(const char *name) const`
  Find a PolyTree with given name in the library. Return a pointer to it if found, otherwise return NULL. The library is searched from the front and the first match is returned. Thus PolyTrees added to the library mask PolyTrees of the same name that were added earlier, making them inaccessible by name.

- `const PolyTree *PolyTreeLibrary::lookup(int i) const`
  Return a pointer to the ith PolyTree defined in the library, counting from the front, and beginning from zero. If i is an invalid index, return NULL.

- `PolyTree *PolyTreeLibrary::create(const char *name) const`
  Like `lookup(name)` except that the returned pointer, if non-NULL, points to a newly instantiated PolyTree obtained by replicating all of the PolyTree nodes of the library version.

- `PolyTree *PolyTreeLibrary::create(int i) const`
  Like `lookup(i)` except that the returned pointer, if non-NULL, points to a newly instantiated PolyTree obtained by replicating all of the PolyTree nodes of the library version.

The `lookup()` methods are provided for informational purposes only. They are useful to check if a particular PolyTree is defined, without instantiating a new copy, or to access a PolyTree for purposes such as counting the number of vertices or printing a description. Whenever PolyTrees are to be used for *V-Clip*, they should be instantiated using one of the `create()` methods.

The library automatically allocates PolyTree nodes when `create()` methods are invoked, and frees them when `clear()` is invoked. Handles on the defined Polyhedron structures are not kept explicitly, although they appear within the PolyTree hierarchies. The Polyhedra are reference counted, so that when all PolyTree nodes that point to a particular Polyhedron are destroyed, the Polyhedron itself is destroyed. Applications are free to use multiple PolyTreeLibraries if needed.

[2]An alternative solution to this reference problem that allows tree nodes to be shared is to address a particular instance of a node with the entire path from the root of the hierarchy. This approach is often used for shared instances of graphical objects in a scene graph. The duplication approach was used in *V-Clip* because it is simpler to implement, and more efficient.

[3]These geometric measures are only provided for convenience; they are not used by *V-Clip*.

4 API for geometry construction

The API described in this section allows an application program to define new geometry readers or construct geometries programatically. The default geometry reader comprises the functions `readPolyhedron()`, `readAtomicPolyTree()`, `readCompoundPolyTree()` and `loadPolyTreeFile()`. These functions demonstrate the use of the geometry construction API described here.

4.1 Building Polyhedra

The following operators and methods are useful in constructing Polyhedra.

- `void *Polyhedron::operator new()`
  Allocates a new Polyhedron from the free store.

- `Vertex *Polyhedron::addVertex(const char *name, const Vect3 &coords)`
  Adds a vertex with given name and coordinates to a Polyhedron. A pointer to the vertex is returned for use with `addFace()`.

- `void Polyhedron::addFace(const char *name, vector<Vertex *> &verts, int clockwise = 0)`
  Adds a face to a Polyhedron. The face is specified by a vector of pointers to vertices previously added to the Polyhedron. The vertices must be specified in clockwise or counter-clockwise order, and the optional third argument specifies the direction, from the perspective of an observer standing outside the Polyhedron, looking at the face. The default is counter-clockwise.

The application builds a Polyhedron by first allocating one using `new`, and then making calls to `addVertex()` and `addFace()`. Edges and the Voronoi regions needed by *V-Clip*are automatically constructed within `addFace()`. Calls to `addVertex()` and `addFace()` may be interleaved, but all vertices on a particular face must be added before the face is added. All vertices and faces must be added before the Polyhedron can be used by *V-Clip*. The Polyhedra must be convex, have closed surfaces without holes, and have faces that are planar, convex polygons. There are no mechanisms that force the application to follow these rules, but *V-Clip* will certainly choke on incorrectly built Polyhedra. A helpful method is:

- `int Polyhedron::check() const`
  Runs basic tests on a Polyhedron to check for correct construction. It checks that all faces are convex polygons, and that all edges are convex. Planarity of faces is not checked. The routine verifies that the Euler formula holds, namely
  $$V - E + F - 2 = 0,$$
  where V is the number of vertices, E is the number of edges, and F is the number of faces. This holds for all Polyhedra with closed surfaces. `check()` returns zero if no problems are found, otherwise it prints out error messages and returns one. If a Polyhedron passes `check()`, there is a good chance it is correctly constructed.

One easy way to build correct Polyhedra is to have the *V-Clip* library automatically compute them as the convex hull of a vertex set. This option is only available when *V-Clip* is linked with the *Qhull* library. If this is done, the application can simply specify the vertices with `addVertex()` and then call

- `int Polyhedron::buildHull()`
  Constructs all faces, edges, and Voronoi regions of a Polyhedron from the convex hull of its vertices. If vertices are discovered that do not lie on the convex hull, these are removed from the Polyhedron's vertex list and deleted. If the *Qhull* library is not linked, calling `buildHull()` generates a runtime error.

4.2 Building PolyTrees

The following operators and methods are useful in constructing PolyTrees. All the methods that create or manipulate pointers to Polyhedra adjust the Polyhedra reference counts appropriately.

- `void *PolyTree::operator new()`
  Allocates a new PolyTree from the free store.

- `void PolyTree::setPoly(Polyhedron *p)`
  Make `p` the Polyhedron pointed to by `this` PolyTree. Every PolyTree has a single Polyhedron pointer that must be initialized. This method is usually invoked directly to set the Polyhedron pointer for atomic PolyTrees. Since Polyhedron structures are shared among PolyTree nodes, the same pointer can be passed to this method multiple times.

- `void PolyTree::addComponent(PolyTree *comp)`
  Adds `comp` as a child of `this` PolyTree. PolyTree `comp` is linked directly into the structure of `this`, rather than being copied into it, so `comp` should not be subsequently added as a component to any other PolyTree. To add `comp` to multiple PolyTrees, or multiple times within the same PolyTree, place it in a library and create multiple instantiations using `create()`.

- `int PolyTree::buildHull()`
  Compute the convex hull of all of the children of `this` PolyTree, create a new Polyhedron to represent the hull, and point `this` PolyTree's Polyhedron pointer to the new Polyhedron. This is the typical way to initialize the Polyhedron pointer for compound PolyTrees. It should only be called after all child components have been added to `this`, and after all Polyhedron pointers of these children have been initialized. The method returns 1 if an error occurred in building the hull, otherwise 0.

- `void PolyTree::xform(const Se3 &T)`
  Every PolyTree node stores the transformation from its own frame to the frame of the PolyTree at the root of the hierarchy. This method premultiplies the stored transformations at all nodes of the hierarchy rooted at `this` by `T`. It is typically used immediately before `addComponent()` to control where the child PolyTree lies relative to the frame of the parent PolyTree. For instance, if PolyTree C is to be included as a component of PolyTree P, but offset by the vector $(0, 0, 10)$ in P's frame, this can be affected by: `C.xform(Se3(Quat::ID, Vect3(0,0,10))); P.addComponent(C);`

5 Other API functions

This section describes miscellaneous API functions that are useful in manipulating and examining Polyhedra and PolyTrees.

5.1 Polyhedron methods

Three methods allow access to the individual features (vertices, edges, and faces) of a Polyhedron:

- `const list<Vertex> &Polyhedron::verts() const`
  Return a list of the vertices of a Polyhedron.

- `const list<Edge> &Polyhedron::edges() const`
  Return a list of the edges of a Polyhedron.

- `const list<Face> &Polyhedron::faces() const`
  Return a list of the faces of a Polyhedron.

These methods can be used to determine how many of each type of feature a Polyhedron has, using the `size()` method of lists. Most of the details of the feature classes are not visible to the application. Class Vertex, class Edge, and class Face are all derived from class Feature, which has a public method:

- `virtual const char *Feature::name() const`

that returns the name of the feature.[4] Also, the coordinates of a vertex can be obtained via

- `const Vect3 &Vertex::coords() const`.

Finally, there is a method to print out a text description of a Polyhedron:

- `ostream& Polyhedron::print(ostream &os) const`
  Print a verbose description of a Polyhedron to an output stream. This description includes lists of all the features, connectivity information, and Voronoi structure information. The stream operator `<<` is overloaded, so that, for example, if `p` is a pointer to a Polyhedron, then `cout << p` invokes `p->print(cout)`.

5.2 PolyTree methods

- `const Polyhedron *PolyTree::poly() const`
  Return the Polyhedron that `this` PolyTree points to.

- `int PolyTree::numNodes() const`
  Return the total number of nodes in the PolyTree hierarchy rooted at `this`.

- `int PolyTree::numLeaves() const`
  Return the total number of leaves in the PolyTree hierarchy rooted at `this`.

- `const Se3 &PolyTree::Tpr() const`
  Return the transformation from the frame of `this` PolyTree to the frame of the PolyTree at the root of the hierarchy. This field can be adjusted using the `xform()` method described above.

- `ostream& PolyTree::print(ostream &os) const`
  Print a description of a PolyTree hierarchy to an output stream. For atomic PolyTrees, the method `Polyhedron::print()` is invoked on the Polyhedron pointed to by the PolyTree. The output for compound PolyTrees is a short representation that refers to individual PolyTrees by name, describes their pose relative to the root of the hierarchy, and uses indentation to show the shape of the hierarchy. The stream operator `<<` is overloaded, so that, for example, if `pt` is a pointer to a PolyTree, then `cout << pt` invokes `pt->print(cout)`.

- `SoNode *PolyTree::buildInvModel() const`
  If the *V-Clip* library is compiled with the SGI *Inventor* library, this methods builds and returns an *Inventor* model of the PolyTree hierarchy rooted at `this`.

One can compute additional data for PolyTrees that is not used by *V-Clip* but that might be useful in other applications such as dynamic simulation. These include the volume of the PolyTree, the first moments of volume (which are useful for computing the center of mass of uniform density objects), and the second moments and products of volume (which are useful for computing the inertia tensor of uniform density objects).[5] The "radius" of the PolyTree may also be computed. To compute any of these quantities, one invokes:

- `void PolyTree::compVolInts()`
  Compute volume integrals and radius of a PolyTree. This method assumes that it has already been invoked on all of the children of `this` PolyTree.

---

[4] Names of edges are not explicitly stored, but are computed in a small buffer. If the application invokes `name()` on successive edges, the result from the second call will overwrite the result from the first call. Copy edge names out of this buffer if they need to be saved.

[5] See `www.cs.berkeley.edu/~mirtich/massProps.html` for information on the algorithms for computing and using these data. When components of a PolyTree overlap, computing the volume integrals is much harder because the shapes of the intersections must be determined. The *V-Clip* library blithely ignores this possibility, assuming the components are disjoint. If they are not, `vol()`, `mov1()`, `mov2()`, and `pov1()` will return incorrect results, with error proportional to the volume of the intersections.

After this has been called, the following methods return meaningful values:

- `const Real &PolyTree::vol() const`
  Return the volume of the PolyTree.

- `const Vect3 &PolyTree::mov1() const`
  Return the first moments of volume of the PolyTree. This returns the vector $(\int xdV, \int ydV, \int zdV)^T$.

- `const Vect3 &PolyTree::mov2() const`
  Return the second moments of volume of the PolyTree. This returns the vector $(\int x^2dV, \int y^2dV, \int z^2dV)^T$.

- `const Vect3 &PolyTree::pov() const`
  Return the products of volume of the PolyTree. This returns the vector $(\int yzdV, \int zxdV, \int xydV)^T$.

- `const Vect3 &PolyTree::rad() const`
  Return the radius of the PolyTree, defined as the maximum distance of any point on the PolyTree from the center of volume.

6 Invoking V-Clip

Compared to the many details of constructing and examining Polyhedra and PolyTrees, invoking *V-Clip* is quite simple. The single interface to the application is through the static PolyTree method

- `static Real vclip(const PolyTree *const ptree1, const PolyTree *const ptree2, const VclipPose &Xr1r2, ClosestFeaturesHT &ht, Vect3 &cp1, Vect3 &cp2)`

Pointers to two distinct PolyTrees are passed in. `Xr1r2` is the transformation from the (root) frame of the first PolyTree to the (root) frame of the second PolyTree. `VclipPose` is just a typedef, usually defined as type `MatX`, but see Section 8 for an alternative. The parameter `ht` is a hash table that *V-Clip* uses to cache closest features between pairs of PolyTrees. The application should declare a variable of type `ClosestFeaturesHT` and pass this to *V-Clip*. A reasonable declaration is

```
ClosestFeatursHT myVclipHashTable(3000);
```

The argument to the constructor specifies a minimum size for the hash table (the STL insures the actual table size is a prime number). A table size of 3000 should be more than adequate for most applications, but with a large number of objects, larger table sizes might improve performance. The number of associations stored in the table is bounded above by the square of the number of individual PolyTree nodes. The function `vclip()` will compute the closest features between the two PolyTrees, and from these compute the closest points between the PolyTrees. The closest points are returned to the application via `cp1` and `cp2`. Their coordinates are specified in the respective root frames of the two PolyTrees. *V-Clip* also returns the distance between the closest points as a signed real number. Negative values correspond to penetration depth; positive values correspond to separation distance.

V-Clip only traverses PolyTree hierarchies as far as necessary to verify disjointness. If the convex hulls of two PolyTrees are disjoint, then so must the PolyTrees be, and *V-Clip* simply returns the closest points between the hulls and the distance between them. Note that these points may not lie on the actual solid part of the PolyTrees, although any positive distance returned is always a lower bound on the true distance between the PolyTrees. When hulls are not disjoint, *V-Clip* unwraps one of them, and recursively calls itself with the components of the unwrapped hull and the still wrapped hull. This process repeats until disjointness can be guaranteed or two atomic (leaf) PolyTrees are found that penetrate.

The order that PolyTrees are passed to `vclip()` matters. If the function is called with `ptree1 =` A and `ptree2 =` B, then it should always be called in this order with these two PolyTrees, never the reverse order. If this rule is violated, nothing will break, but the performance will be slower since the pair (A, B) is not associated with the pair (B, A).

If the application needs the closest features between the two PolyTrees, they can be easily had by first calling `vclip()` and then calling

- `static void vclipFeatures(const PolyTree *const ptree1, const PolyTree *const ptree2, ClosestFeaturesHT &ht, const Feature *&feat1, const Feature *&feat2)`

This function simply does a hash table lookup, retrieving the last pair of closest features stored for the particular PolyTree pair. If `vclip()` has not been called for the specified pair, `vclipFeatures()` returns NULL.

7 Example programs

There are some example programs in the directories `examples` and `iv_examples`. The programs in `examples` do not require the SGI *Inventor* library; the ones in `iv_examples` do.

1. `examples/print.` This program prints out descriptions of all of the PolyTrees defined in a PolyTree file specified on the command line. This program does not use *V-Clip* at all, only the PolyTree facilities.
2. `examples/allPairs.` This program reads in a body specification file given on the command line. This file lists PolyTree file(s) to read in, followed by a list of bodies. Each body comprises a defined PolyTree at a particular pose in space. The program then runs *V-Clip* between all pairs of bodies, and prints out the results.
3. `iv_examples/inventorize.` This program generates *Inventor* models of PolyTrees. It reads the PolyTree file given on the command line, builds an *Inventor* model for the last PolyTree defined in this file, and writes the model (in ascii format) to standard output. The ouput can be piped to a file and then viewed with a viewer such as `ivview`. This example does not use *V-Clip*.
4. `iv_examples/vcdemo.` This program reads in a demo file that lists PolyTree file(s) to read in, followed by instantiations of the defined PolyTrees as bodies. It opens an *Inventor* viewer that can examine the bodies. The closest points between all pairs of bodies are automatically tracked, and the colors of each body indicate how many other bodies are penetrating it. The user can click and drag the bodies to new positions, and all of the bodies can be simultaneously moved in a continuous manner, as the closest points are tracked.

8 Vector, matrices, and quaternions

The *V-Clip* library uses a home grown collection of classes for objects like vectors, matrices, and quaternions. These are defined in `include/mv.h` and `src/mv.C`. The classes are specialized for three dimensions and may not be suitable for more general purposes. The file `mv.h` is well commented and describes the methods and operations available to the application. However, the vanilla interface should contain few surprises. Most of the functionality is not used by *V-Clip* and provided only for completeness. The specific classes defined are:

- `class Vect3`
  Three-dimensional vectors.
- `class Mat3`
  Three by three matrices, with several methods specialized for rotation matrices.
- `class Quat`
  Quaternions: four parameters specifying a three-dimensional rotation.
- `class MatX`
  A full three-dimensional transformation comprising a translation and a rotation. The translation is implemented as a `Vect3`, and the rotation is implemented as a `Mat3` (rotation matrix).
- `class Se3`
  A full three-dimensional transformation comprising a translation and a rotation. The translation is implemented as a `Vect3`, and the rotation is implemented as a `Quat`.

Each of the two representations for three-dimensional transformations have advantages. The quaternion-based representation (`Se3`) is more compact, and generally easier to work with when rotations are manipulated. The matrix representation (`MatX`) allows points and vectors to be transformed with marginally fewer floating point operations. The *V-Clip* algorithm itself can use either representation for transformations (see the compiler flag `VCLIP_MATRIX_POSE` and surrounding comments in `include/vclip.h`). The recommendation for most applications is to use the matrix representation for maximum speed.[6] Applications can use either representation for their own work. Translators between the two representations are provided.

9 Where V-Clip breaks

V-Clip was designed for numerical robustness in the presence of degenerate configurations. In particular, it was designed to avoid the "cycling" behavior that occurs with other surface-based closest feature algorithms. In robustness testing, *V-Clip* outperformed other algorithms on degenerate problems, and has never exhibited cycling (see the tech report). Degeneracy problems are quite insidious, however, and *V-Clip* is not a panacea. Extensive use has revealed that degenerately *penetrating* geometries can fool *V-Clip*. Examples include two identical objects placed with the same pose so that they exactly coincide, or two objects that have exactly intersecting edges but no edges that pierce faces. In situations like these, *V-Clip* might incorrectly classify the objects as disjoint. An epsilon perturbation of one of the objects will remedy the situation. The good news is that even these contrived examples do not cause numerical exceptions nor prevent *V-Clip* from returning. And when objects are truly disjoint, *V-Clip* has not exhibited any problems, even for degenerate configurations. Scattered evidence suggests that using `Se3` instead of `MatX` for pose representation (see previous section) improves robustness for degenerately penetrating configurations.

[6] With modern CPUs, which representation allows faster transformations is not obvious: `MatX` requires fewer floating point operations, but `Se3` may require fewer memory accesses.