S W I F T

Speedy Walking via Improved Feature Testing

http://www.cs.unc.edu/∼geom/SWIFT/

Version 1.0

Application Manual

Stephen A. Ehmann

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
ehmann@cs.unc.edu
http://www.cs.unc.edu/∼ehmann/

Introduction

SWIFT is a collision detection package capable of detecting the intersection, computing the approximate distance, computing the exact distance, or determining the contacts amidst a scene composed of rigid convex-pieced polyhedral models.

It is a robust and efficient library shown to be substantially faster than currently available packages. It is a powerful package from its input interface to its query interface providing a rich feature set. There are many settings available but the basic principles are rather simple and straightforward.

This manual will attempt to cover all the features available in SWIFT as well as provide useful shortcuts and efficiency tips. The material will first be covered in general followed by added detail.

Contents

1 License Agreement

The SWIFT library is Copyright 2000 The University of North Carolina at Chapel Hill. The LICENSE file that accompanies the distribution gives in detail what this means along with the restrictions on any executable that this library is linked with. SWIFT is written in C++ and all the source code is included in the distribution. The LICENSE file also covers redistribution and code modifications.

2 Customization, Build, and Installation

Assuming that the library has been unpacked into the directory SWIFT/, follow these steps to get a compiled library:

1. Look through the configuration file in the SWIFT/include/ directory called SWIFT_config.h and set all desired options. Various compile-time features may be selected including floating point type, usage of the hierarchy, type of hierarchy to be used, debugging, etc. Refer to section 4.1 and SWIFT_config.h for full details.

2. Compile the library
   - If compiling using VC++ 6.0 use swift.dsp.
   - Else if compiling for some Unix look through SWIFT/Makefile and do the following:
     - Set CC to point to the proper compiler and CFLAGS to the desired compile flags
     - Uncomment the Qhull and QSlim flags and point them to the current system's installation of those packages if the QSlim hierarchy is desired (and the option was turned on in SWIFT_config.h).

- Type "make" in the SWIFT/ directory and a library will be built as SWIFT/lib/libSWIFT.a. There may be a few warnings which can be safely ignored.
- Else you will have to create your own compiler configuration

3 Architectural Overview

The application making use of SWIFT interfaces by including SWIFT.h (in the SWIFT/include/ directory) and creating a *SWIFT_Scene* object. More than one *SWIFT_Scene* may exist at any given time. A *SWIFT_Scene* can import object geometry, test objects for intersection, compute distance or determine contacts between them, find closest features, closest points, contact normals, and produce reports on the results. All application calls are made to methods of the *SWIFT_Scene* object. More details on these methods are provided in detail later in this document.

There are two main phases in the use of the SWIFT system. The first is the precomputation phase where information is computed for the objects in the scene in preparation for the query phase (the second one).

The steps to using the SWIFT system are typically performed in the outlined order given here for each of the two phases. This order is the intended order and is therefore the most efficient.

3.1 Precomputation Phase

1. Create a *SWIFT_Scene* object and set the desired scene configuration. Reasonable defaults are preset.

2. Add or copy all of the objects which compose the scene by calling the Add_*_Object() methods. Note that if two or more objects are composed of exactly the same geometry and input transformation, then it is better to cause the object to be copied (after the first time) so that its geometry is not replicated. Copied objects can still move independently in the scene and there is no performance loss from this geometry sharing.

   The Add_*_Object() methods are meant to precompute information that is used at query time. If they are called during the query phase, performance will suffer ***greatly***.

   There are two types of objects: fixed and moving. When an object is added, one of these types is chosen and cannot be undone. Fixed objects are usually ones that never move such as walls or other static objects. They are never tested against one another (but can be moved around if so desired).

3. Select pairs of objects to be tested by activating and/or deactivating them. By default, all object pairs are initially deactivated if both objects are fixed and activated if either of the objects is not fixed. Pairs that are activated or deactivated stay that way for subsequent queries until the activation state is changed. This step is listed in the precomputation phase because it incurs an overhead if done during the query phase. If at all possible try to activate or deactivate as many pairs as possible before the query phase. Only change the activation

status of objects during the query phase if it cannot be determined during the precomputation phase.

4. Set the first transformations of all the objects including the fixed objects using one of the Set_Object_Transformation() methods. Fixed objects need not (and should not) have their transformations ever set again since they do not move. Transformations must be only rotations and translations (within floating point error). If non-rotation matrices are given, SWIFT will malfunction.

5. Since the first query is likely to be slow for an application whose scene exhibits high coherence, a dummy query may be desirable before the query phase to "initialize" the query. The objects should obviously be in their initial positions.

3.2 Query Phase

1. Set object transformations using either one of the Set_Object_Transformation() methods or one of the Set_All_Object_Transformations() methods which set the transformations for all of the moving objects with a single call. It is highly recommended that object transformations be set at most once per object per query since an overhead is incurred for each setting. The fixed object transformations should only be set during the precomputation phase (but they can be set during the the query phase as well if so desired). Furthermore, moving object transformations should only be set if the object actually moves. Transformations must be only rotations and translations (within floating point error). If non-rotation matrices are given, SWIFT will malfunction.

2. Call a Query_*() method based on the application's needs. There are four choices, the details of which will be explored later. They are: intersection detection, approximate distance computation, exact distance computation, and contact determination.

3. Objects may be deleted during the course of the query phase by using the Delete_Object() method. Objects may be copied and added during this phase as well but the application must be careful as the overhead could be quite high for the addition. More on this later.

4 Sketch of Provided Functionality

Here a sketch of the functionality that SWIFT provides is laid out. Details are filled in later on in section 6. For now, the aim is to just show what is possible. First the compile-time configuration is discussed, then library configuration, then scene configuration, and then object configuration. The input geometry possibilities are then described, followed by a description of the available output.

4.1 Compile Configuration

SWIFT may be configured in various ways at compile-time. The configuration can be done in the following ways by editing SWIFT_config.h before compilation:

- **Floating Point Type:** the floating point type can be chosen as *float* or *double* by defining the **SWIFT_USE_FLOAT** variable or not. The type *SWIFT_Real* is the floating point type that is used universally throughout SWIFT, including the interface. This should be set to match the application. The default is *double*.

- **Maximum Vertex Valence:** set the maximum valence that any vertex given to SWIFT will have.

- **Lookup Table:** a lookup table may be used to provide initialization to speed up queries. Turning this on means that initialization of the distance minimization routine is ***always*** done through the lookup table. See the technical report for more details. The lookup table is enabled by defining the **SWIFT_ALWAYS_LOOKUP_TABLE** variable. Note that the hierarchy option cannot be on if this option is on. The default is off.

- **Lookup Table Resolutions:** a lookup table is always built around each object (even if SWIFT_ALWAYS_LOOKUP_TABLE is not defined) and is used to initialize the query in some cases. The lookup table resolution must be chosen to be one of 22.5 degrees, 11.25 degrees, or 5.625 degrees. They are chosen by defining one of **LUT_RESOLUTION_22_5**, **LUT_RESOLUTION_11_25**, or **LUT_RESOLUTION_5_625**. The storage costs are approximately 0.5 kB, 2 kB, and 8 kB respectively. There is not much of a performance difference but in general, the highest resolution provides the highest performance.

- **Convex Hierarchy:** a hierarchy of simpler convex objects may be used to speed up queries. This is enabled by defining the **SWIFT_HIERARCHY** variable. Note that the lookup table option cannot be on if this option is on. The default is off.

- **QSlim Convex Hierarchy:** a hierarchy that uses the QSlim package to create simpler levels. This is enabled by defining the **SWIFT_QSLIM_HIER** variable. We have found that QSlim hierarchies perform better. In order to use this type of hierarchy, the application must edit the Makefile to point to Qhull and QSlim distributions on the system. These packages must also be linked to by the application. If this is off but the hierarchy is on, the hierarchy will be built using a modified Dobkin-Kirkpatrick method. The default is off.

- **Debug:** debugging information may be turned on to help uncover faults in the input. This is enabled by defining the **SWIFT_DEBUG** variable. Extensive input geometry checking is done as well as some limited query checking. Performance is slower when this option is turned on. The default is off.

4.2 Library Configuration

The only library configuration that may be done at run-time is the registration of application-defined file readers. This may be useful if the application wishes to have SWIFT read certain files directly without SWIFT knowing about the format of the files. The procedure for building a custom file reader is simple and is described more in section 9.

4.3 Scene Configuration

SWIFT may be configured in various ways during *SWIFT_Scene* object creation. Different created scenes may be configured differently. A scene can be configured in the following ways:

- **Broad Phase:** a sweep and prune algorithm based on axis-aligned bounding boxes and dimension reduction can be used to cull away a large number of more expensive tests. This is achieved by constructing boxes to meet "region of interest" criteria. Note that when the hierarchy is on, the bounding boxes are created about the coarsest level. The default is on.

- **Global or Local Sorting:** the sorting step of the sweep and prune algorithm can be either local or global. If it is global, then the boxes are sorted all at one time at the beginning of a query. If it is local, then each object's box is locally sorted as soon as the object's transformation is updated. This option has no effect if the broad phase algorithm is not turned on. The default is global.

4.4 Object Configuration

Objects that are added to a scene can be configured in various ways. The only object configuration that should be done at query time is the position and orientation of an object and possibly its activation status with respect to other objects. All other possible configurations of objects are done when they are added and cannot be changed thereafter. These *static* configurations are the following:

- **Geometry:** an object's geometry can be given as arrays of vertices and faces (indices into the vertices) or as filenames that indicate files to read the geometry from. In the case of files, SWIFT will attempt to read the given files using whatever file readers have been registered. An object can be composed of one or more convex pieces. SWIFT allows for the copying of any already added piece. This can usually avoid geometry replication. More details are given later. SWIFT will automatically triangulate any convex polyhedral model composed of convex polygonal faces. In other words, the faces need not be triangles. SWIFT will also fully share the vertices that are given according to strict equality. In other words, a vertex may be duplicated as long as it has the exact same coordinates.

- **Hierarchy:** an object will have a convex hierarchy built around it if the appropriate compile-time variable was defined (see above). Alternatively, the application can provide the hierarchy to be used. SWIFT will enforce the appropriate hierarchy constraints such as registration, boundedness, and parenting. Each hierarchy can have up to 32 levels. If more are given or if more can be created, SWIFT will use the first 32 available. Application provided hierarchies are not subject to the scene configuration parameters affecting hierarchies. Those parameters only affect hierarchies built by SWIFT.

- **Fixed or Moving:** an object can either be fixed or moving. As mentioned previously, if an object is fixed, its transformation should only be set once at the beginning (but may be set later if so desired). Furthermore, fixed objects are never tested against one another. The default is moving.

- **Input Transformation:** an object can be rotated, translated, and/or scaled up or down from its input geometry when it is first added to the scene. This input transformation is performed explicitly as a precomputation step and is not incorporated into the movement transformations which can only be rotations and translations. Do ***NOT*** attempt to work a scaling factor into the movement rotation matrices. This will cause SWIFT to malfunction.

  If a copy is done then the geometry will not be replicated if the input transformation given is the identity transformation, otherwise it will[1]. The default is identity.

- **Bounding Box Type:** there are two types of bounding boxes available. The first type is a cube. It is relatively efficient to update but does not fit large aspect ratio objects very well. The second type is a dynamic bounding box. This type costs more to update with each transformation but fits the underlying geometry better, causing fewer boxes to overlap. The application can chose from five options: **DEFAULT**, **CUBE**, **DYNAMIC**, **CHOOSE**, or **COPY**. If the **DEFAULT** option is given, then the defaults as described below are used. The second two choices have already been explained. If the **CHOOSE** option is chosen, then SWIFT will calculate the maximum and minimum spread of the object and use this aspect ratio to decide what type of bounding box to use based on the object configuration parameter (cube aspect ratio) described below. The **COPY** option is only allowed if a copy is being done. The type of bounding box that is set for the piece being copied is chosen for the piece being copied to. This is the recommended setting when a copy is done except for the case where a copy is done *from* a fixed object *to* a moving object since a fixed object always has a dynamic bounding box. The default is **CUBE** if no copy is done and the object is moving. If the object is fixed and no copy is done then the box type is always set to be **DYNAMIC**. If a copy is done *from* a moving object *to* a moving object, then the default is **COPY**. If the copy is done *from* a fixed object *to* a moving object, then the default is **CUBE**. If the copy is done *to* a fixed object, then **DYNAMIC** is used since the fixed object is not assumed to move (although this is possible).

- **Bounding Box Enlargements:** a bounding box may be enlarged by an absolute and/or relative amount. This is useful if the application is interested in performing toleranced queries such as distance or contacts. In this case an absolute enlargement equal to the desired tolerance should be used. Even though enlarging bounding boxes causes more overlap, it can be used to maintain coherence for objects that are relatively close over several frames. In this case, a relative enlargement may be desirable. The default is 0.0 for both relative and absolute enlargements.

- **Minimum Triangle Count:** the number of triangles below which the hierarchy is not built. This quantity indicates when to stop building coarser levels for the hierarchy. This option has no effect if the hierarchy is not compiled on. This option only affects hierarchies that are built by SWIFT and not those defined by the application. The default is 50.

[1]If this turns out to cause an unnecessary explosion in terms of storage for some applications, bring this to our attention and something can be done about it.

- **QSlim Triangle Ratio:** the ratio of triangles between two levels of the QSlim hierarchy since this can be controlled. This option has no effect if the hierarchy is not compiled on or if the QSlim hierarchy is not compiled on. This option only affects hierarchies that are built by SWIFT and not those defined by the application. The default is 0.25.

- **Cube Aspect Ratio:** the ratio between the maximum spread and the minimum spread of an object below which a cube is used for the sweep and prune algorithm if the bounding box option of an object is **CHOOSE**. This decision is made automatically by SWIFT – no user intervention or spread values are required. This option has no effect if the broad phase algorithm is not turned on or if an object's bounding box option is not **CHOOSE**. Note that it does not make sense to set this to be less than 1.0. The default is 2.0.

4.5 Input Geometry

The input geometry has to be convex polyhedra that are closed. If the geometry is passed using arrays, then the vertex array is indexed by the elements of the face array to determine connectivity. Faces do not have to be triangular. Things are simpler if they are however. As mentioned above, an input transformation may be applied to the input geometry. The input transformation can be quite useful in assembling, say, a multiple piece object[2].

The following geometric properties must be satisfied by all input geometry:

- The object description must be convex which also implies that the topology is homeomorphic to a sphere.
- It must be closed. Every edge must belong to exactly two faces.
- There must not be any degenerate elements such as zero length edges, collinear edges, or zero area faces.
- All vertices on a face must be coplanar (to within floating point precision).

4.6 Output

The output that SWIFT provides is accessed through the query methods. The four methods provide a rich set of proximity queries. An early exit option is provided for each query to exit as soon as an intersection is found. If the early exit option is turned on, there is no reporting if an intersection is found. If there is no intersection then reporting occurs as normal. The four types of queries are:

- **Intersection:** the objects are tested for intersection and all pairs of intersecting objects are reported.

[2] All pieces with the same geometry but different transformations within the object are currently replicated. This is in our future work section at the end of the manual.

- **Approximate Distance:** the objects are tested for intersection by traversing the hierarchies and if they are found to be disjoint, the approximate distance is given by the distance of the two levels that were found to be disjoint and the error is given as the sum of the deviations of the levels from their original objects. The exact distance lies in the range [approximate distance, approximate distance+error]. This query reports the approximate distance and associated error for pairs whose approximate distance is less than an application specified distance tolerance and whose associated error is less than and application specified error tolerance. Note that if the hierarchy is not compiled on, then the approximate distance query is the same as the exact distance query.
- **Exact Distance:** the objects are tested for intersection but when they are found to be disjoint, the test does not stop immediately but rather descends all the way to the finest levels of the two hierarchies whose distance is the exact distance. This query reports the exact distance for pairs whose exact distance is less than an application specified tolerance. This allows for the test to be stopped short of the finest levels when a sufficiently large approximate distance has been found.
- **Contact Determination:** the objects are tested the same as for exact distance. If the finest levels are reached and the distance is within the application specified tolerance, then the application can be given various reports. The reports can be any combination of: distance, nearest points, contact normals, and nearest features. From these items contact response may be computed.

5 Conventions

- The coordinate system used is a right-handed one.
- All matrices are given in row-major order.
- All transformations are applied by left-multiplication: $P_1 = RP_0$.

6 Detailed Description of the Interface

In this section a detailed explanation of each *SWIFT_Scene* method is given along with all the subtleties. Refer to the architectural overview in section 3 to see the general order in which to call the methods. SWIFT.h contains limited documentation that is similar to this.

6.1 Scene Creation (Configuration) and Deletion

Scene deletion is straightforward. Scene configuration happens when the *SWIFT_Scene* object is created and cannot be changed afterwards.

- **Constructor**
  Create a *SWIFT_Scene* object. Turn the broad phase (sweep and prune) algorithm on or off (default is on). Turn on global sorting or local sorting (by setting global_sort to **false**) (default is global_sort). See section 4.3 for more detail on the meanings of these settings.

  ```
  SWIFT_Scene( bool broad_phase = DEFAULT_BP,
               bool global_sort = DEFAULT_GS );
  ```

6.2 Object Creation and Deletion

There are various ways to create objects. Creating objects is equivalent to adding them to the scene. There are eight different ways to add objects. They differ in whether geometry is read from files or not, whether there is one piece per object or more than one piece, and whether the application provides the hierarchy or SWIFT builds it. Each hierarchy that is given or created can have at most 32 levels. Objects can also be deleted.

Each of the methods passes back an identifier for the added object which can be used on subsequent references to it when setting transformations, activating, or querying. The sequence of ids that are passed back increase by 1 starting at 0. Each method returns a boolean value indicating success of the addition. If the object addition failed, there will be a message printed to **stderr** indicating the problem.

The types *SWIFT_Orientation* and *SWIFT_Translation* are defined in SWIFT.h as constant length arrays of size 9 and 3 respectively and are used as parameter types in the following object creation functions. They represent row-major rotation matrices and translation vectors.

- **Add_Object**
  This pair of methods is for adding an object composed of a single piece. The first method is for input geometry stored in arrays while the second is for input geometry stored in files. It is possible to copy by using the first one but not the second one.

  ```
  bool Add_Object(
              const SWIFT_Real* vertices, const int* faces,
              int num_vertices, int num_faces, int& id,
              bool fixed = DEFAULT_FIXED,
              const SWIFT_Orientation& orient = DEFAULT_ORIENTATION,
              const SWIFT_Translation& trans = DEFAULT_TRANSLATION,
              SWIFT_Real scale = DEFAULT_SCALE,
              int box_setting = DEFAULT_BOX_SETTING,
              SWIFT_Real box_enl_rel = DEFAULT_BOX_ENLARGE_REL,
              SWIFT_Real box_enl_abs = DEFAULT_BOX_ENLARGE_ABS,
              const int* face_valences = DEFAULT_FACE_VALENCES,
              int copy_oid = DEFAULT_COPY,
              int copy_pid = DEFAULT_COPY,
              int min_tri_count = DEFAULT_MIN_TCOUNT,
              SWIFT_Real qslim_tri_ratio = DEFAULT_QSLIM_TRATIO,
              SWIFT_Real cube_aspect_ratio = DEFAULT_CUBE_ASPECT_RATIO );
  ```

  ```
  bool Add_Object(
  ```

```
const char* filename, int& id,
bool fixed = DEFAULT_FIXED,
const SWIFT_Orientation& orient = DEFAULT_ORIENTATION,
const SWIFT_Translation& trans = DEFAULT_TRANSLATION,
SWIFT_Real scale = DEFAULT_SCALE,
int box_setting = DEFAULT_BOX_SETTING,
SWIFT_Real box_enl_rel = DEFAULT_BOX_ENLARGE_REL,
SWIFT_Real box_enl_abs = DEFAULT_BOX_ENLARGE_ABS,
int min_tri_count = DEFAULT_MIN_TCOUNT,
SWIFT_Real qslim_tri_ratio = DEFAULT_QSLIM_TRATIO,
SWIFT_Real cube_aspect_ratio = DEFAULT_CUBE_ASPECT_RATIO );
```

The first method adds an object to the scene by reading the geometry from the *vertices* and *faces* arrays. The vertex array is a list of coordinate values (x,y,z) in 3D. The length of the vertex array is 3 times *num_vertices*. If all the faces are triangular then the length of the face array is 3 times *num_faces*. If not all the faces are triangular then the array *face_valences* should be *num_faces* long and each element indicate the number of vertices for each face. Then, the length of the face array is the sum of the number of vertices for each face over all faces. The default is that all faces are triangular signified by the *face_valences* parameter being **DEFAULT_FACE_VALENCES**. Note that the indices stored in the face array refer to the vertex index rather than the coordinate index. The vertex indices start at 0. Vertices may have duplicates (exactly equal coordinates). SWIFT will fully share the vertices internally. A contact determination query can be made to report the closest features. The identifiers of these features are related to the ordering in which the vertices and faces were given to the Add_Object() method. More details on the identifiers is given along with the description of the query. The identifier of the object is passed back in *id*.

The *fixed* parameter indicates whether the object is fixed or moving. Objects are moving by default. If the object is fixed, it is fitted with a **DYNAMIC** bounding box regardless of the type set for it since its transformations should never change (after the first one) and it might as well use the tightest box possible. Note that an object cannot be "un-fixed" once it has been declared fixed or vice-versa. All pairs of fixed objects are *never* tested so fixed objects can intersect arbitrarily.

The *orient* and *trans* arrays specify the input transformation to be applied to the object. If they are set to **DEFAULT_ORIENTATION** and **DEFAULT_TRANSLATION** respectively, then they are considered the identity transformation (this is the default). Otherwise, *orient* should be a 9 element 3 by 3 row-major rotation matrix and *trans* should be a 3 element translation vector. The *scale* parameter specifies the input scaling factor which is by default 1.0 (no scaling). All subsequent (movement) transformations that are applied to an object are applied in addition to (composed with) the input transformation. For a copy, input transformations are composed. See below.

Next, the bounding box settings are given. The default is **CUBE** along with no enlargement if the object is moving. See section 4.4 for more details about bounding box types and settings. The *face_valences* parameter is used to indicate whether there are any non-triangular faces (see above).

Next, the copy parameters are given. If they are set to **DEFAULT_COPY** then no copy is done. Otherwise, *copy_oid* and *copy_pid* should be set to a previously added object id and piece id within that object respectively. Piece ids start at 0. If an object id is given but the piece id is given as **DEFAULT_COPY**, then the first piece of the object is the one copied. If a copy is done, the arrays for geometry are ignored. The fixed, input transformation, and the bounding box parameters are applied to the piece being copied. This has the effect of composing input transformations. For example, say that object 0 was added with a scaling factor of 2.0 and then object 1 is copied from object 0 with a scaling factor of 2.0. Object 1 would be 2 times as big as object 1 and 4 times as big as the original geometry used to create object 0. This applies to input orientations and translations as well. If identity transformations (**DEFAULT_ORIENTATION**, **DEFAULT_TRANSLATION**, and **DEFAULT_SCALE**) are given, then the geometry is not replicated.

The last three parameters affect the hierarchy and the bounding box creation. The *min_tri_count* parameter specifies the number of triangles to cut the hierarchy creation off at. The *qslim_tri_ratio* parameter is used to specify the triangle reduction factor when creating a QSlim hierarchy. Note that these two parameters have an effect only when the appropriate hierarchy compile configuration flags are turned on. The last parameter, *cube_aspect_ratio*, specifies the aspect ratio below which a cube is used when the box setting is set to **CHOOSE**. See section 4.4 for more details on these three parameters.

The second method is very similar to the first with the exception that it gets the input geometry from a file and no copying is allowed. If copying is desired, use the first method. Section 10 specifies the file formats natively supported by SWIFT. Additional formats may be supported through application-defined file readers that may be plugged in. More details on this are given in section 6.6 and section 9.

- **Add_Pieced_Object**
  This pair of methods is for adding an object composed of more than one piece. The first method is for input geometry stored in arrays while the second is for input geometry stored in files. It is possible to copy select pieces in either of the two methods.

  Note that it is not possible to perform "self" proximity queries on these pieced objects since the object moves rigidly as a whole. All the convex pairs that are within an object are automatically deactivated. In this version of SWIFT, there is no support for convex hierarchies utilizing convex hulls to speed up pieced object vs. pieced object testing.

```
bool Add_Pieced_Object(
            const SWIFT_Real* const* vertices, const int* const* faces,
            const int* num_vertices, const int* num_faces,
            int num_pieces, int& id,
            bool fixed = DEFAULT_FIXED,
            const SWIFT_Orientation* orient = DEFAULT_ORIENTATIONS,
            const SWIFT_Translation* trans = DEFAULT_TRANSLATIONS,
            const SWIFT_Real* scales = DEFAULT_SCALES,
            const int* box_settings = DEFAULT_BOX_SETTINGS,
            const SWIFT_Real* box_enl_rel = DEFAULT_BOX_ENLARGE_REL,
```

```
            const SWIFT_Real* box_enl_abs = DEFAULT_BOX_ENLARGE_ABSS,
            const int* const* face_valences = DEFAULT_FACE_VALENCESS,
            const int* copy_oids = DEFAULT_COPIES,
            const int* copy_pids = DEFAULT_COPIES,
            const int* min_tri_count = DEFAULT_MIN_TCOUNTS,
            const SWIFT_Real* qslim_tri_ratio = DEFAULT_QSLIM_TRATIOS,
            const SWIFT_Real* cube_aspect_ratio
                                    = DEFAULT_CUBE_ASPECT_RATIOS );

bool Add_Pieced_Object(
            const char* const* filenames,
            int num_pieces, int& id,
            bool fixed = DEFAULT_FIXED,
            const SWIFT_Orientation* orient = DEFAULT_ORIENTATIONS,
            const SWIFT_Translation* trans = DEFAULT_TRANSLATIONS,
            const SWIFT_Real* scales = DEFAULT_SCALES,
            const int* box_settings = DEFAULT_BOX_SETTINGS,
            const SWIFT_Real* box_enl_rel = DEFAULT_BOX_ENLARGE_RELS,
            const SWIFT_Real* box_enl_abs = DEFAULT_BOX_ENLARGE_ABSS,
            const int* copy_oids = DEFAULT_COPIES,
            const int* copy_pids = DEFAULT_COPIES,
            const int* min_tri_count = DEFAULT_MIN_TCOUNTS,
            const SWIFT_Real* qslim_tri_ratio = DEFAULT_QSLIM_TRATIOS,
            const SWIFT_Real* cube_aspect_ratio
                                    = DEFAULT_CUBE_ASPECT_RATIOS );
```

The first method adds an object to the scene by reading the geometry from the *vertices* and *faces* arrays. The only difference from the first Add_Object() method is that the arrays are two-dimensional because there is more than one piece. They are given in "piece-major" order. Consequently, the *num_vertices* and *num_faces* parameters are one-dimensional arrays containing the number of vertices and faces for each piece. The *face_valences* parameter behaves the same as before if some faces are non-triangular. If an entire piece is known to be triangular, then the corresponding sub-array in the *face_valences* array may be set to **DEFAULT_FACE_VALENCES**. The number of pieces composing the object is given in the *num_pieces* parameter. All of the geometry arrays should be of length equal to *num_pieces* in the first dimension (except for the *face_valences* array which may be set to **DEFAULT_FACE_VALENCESS** which has no length). The identifier of the object is passed back in *id*. If pieces ever need to be referenced, they implicitly have ids starting at 0 and increasing in the order in which they appeared in this method invocation.

The *fixed* parameter applies to the entire object since an object moves as a whole and all its pieces maintain the same relative placements at all times. The input transformations are given as before and are all identity by default. Each piece may have different box settings. By default all are of type **CUBE** and have no enlargement if the object is moving otherwise all have type **DYNAMIC** always.

Next, the copy parameters are given. If they are set to **DEFAULT_COPIES** then no copying is done. Otherwise, *copy_oids* and *copy_pids* are arrays that specify which of the pieces are copied and which are not. They are *num_pieces* in length. If an element of the *copy_oids* array is set to **DEFAULT_COPY** then that piece is not copied but rather built from the geometry arrays. Otherwise, the object given by the element of the *copy_oids* array is accessed and one of it pieces is copied to create the current piece. The piece that is copied is identified by the corresponding element in the *copy_pids* array, unless it is equal to **DEFAULT_COPY** in which case the first piece is copied. If the *copy_pids* is set equal to **DEFAULT_COPIES** then the first piece of each copied object is automatically selected. Note that not all objects must be copied or all created through geometry. A mix is allowed. The only requirement is that there be geometry present for the pieces that are not copied. For those that are copied, their positions in the geometry arrays may be empty. Input transformations act as before.

The *min_tri_count* parameter specifies the number of triangles to cut the hierarchy creation off at for each piece. If it is set to **DEFAULT_MIN_TCOUNTS** then the default is used. Similarly, the *qslim_tri_ratio* parameter is used to specify the triangle reduction factor when creating a QSlim hierarchy and the *cube_aspect_ratio* parameter is used to specify the aspect ratio below which a cube is used if the bounding setting is **CHOOSE**. These last two parameters can be set respectively to **DEFAULT_QSLIM_TRATIOS** and **DEFAULT_CUBE_ASPECT_RATIOS** to use the defaults. See section 4.4 for more details on these three parameters.

The second method is very similar to the first with the exception that it gets the input geometry from files whose names are given in the *filenames* array. This array must have length equal to *num_pieces* but may have empty filenames in the positions of copied pieces. Section 10 specifies the file formats natively supported by SWIFT. Additional formats may be supported through application defined file readers that may be plugged in. More details on this are given in section 6.6 and section 9.

- **Add_Hierarchical_Object**
  This pair of methods should not be invoked if the hierarchy is not turned on. These methods are for adding an object composed of a single piece where the application provides the hierarchy to be used by SWIFT. Application provided hierarchies are not subject to the scene configuration parameters affecting hierarchies. Those parameters only affect hierarchies built by SWIFT. The levels of the hierarchy must be convex polyhedra that are closed. SWIFT will process the given hierarchy to ensure that all the criteria are met. The only transformations that are applied during this process are translations and scalings. The first method is for input geometry stored in arrays while the second is for input geometry stored in files. No copying is allowed in these methods because the first Add_Object() method provides that functionality.

```
bool Add_Hierarchical_Object(
            const SWIFT_Real* const* vertices, const int* const* faces,
            const int* num_vertices, const int* num_faces,
            int num_levels, int& id,
            bool fixed = DEFAULT_FIXED,
            const SWIFT_Orientation* orient = DEFAULT_ORIENTATIONS,
            const SWIFT_Translation* trans = DEFAULT_TRANSLATIONS,
            const SWIFT_Real* scale = DEFAULT_SCALES,
```

```
            int box_setting = DEFAULT_BOX_SETTING,
            SWIFT_Real box_enl_rel = DEFAULT_BOX_ENLARGE_REL,
            SWIFT_Real box_enl_abs = DEFAULT_BOX_ENLARGE_ABS,
            const int* const* face_valences = DEFAULT_FACE_VALENCESS,
            SWIFT_Real cube_aspect_ratio = DEFAULT_CUBE_ASPECT_RATIO );

bool Add_Hierarchical_Object(
            const char* const* filenames,
            int num_levels, int& id,
            bool fixed = DEFAULT_FIXED,
            const SWIFT_Orientation* orient = DEFAULT_ORIENTATIONS,
            const SWIFT_Translation* trans = DEFAULT_TRANSLATIONS,
            const SWIFT_Real* scale = DEFAULT_SCALES,
            int box_setting = DEFAULT_BOX_SETTING,
            SWIFT_Real box_enl_rel = DEFAULT_BOX_ENLARGE_REL,
            SWIFT_Real box_enl_abs = DEFAULT_BOX_ENLARGE_ABS,
            SWIFT_Real cube_aspect_ratio = DEFAULT_CUBE_ASPECT_RATIO );
```

The first method adds an object to the scene by reading the geometry from the *vertices* and *faces* arrays. These arrays are two-dimensional because the levels of the hierarchy are a sequence of convex polyhedra. The *num_vertices* and *num_faces* arrays give the number of vertices and faces for each level of the hierarchy. The hierarchy is to be given in finest to coarsest order. So, the first set of vertices and faces are for the finest object and the last are for the coarsest. The parameter *num_levels* gives the number of levels in the hierarchy. The *face_valences* parameter acts the same as described for the previous two pairs of methods. The identifier of the object is passed back in *id*.

The *fixed* parameter indicates whether the object is fixed or moving. The *orient* and *trans* arrays specify the input transformation to be applied to the levels of the object. Next, the bounding box settings are given. There is a single box bounding the coarsest level of the hierarchy, hence settings for just one box. The last parameter affects the bounding box creation just like for the Add_Object() methods. See section 4.4 for more details on this.

The second method is very similar to the first with the exception that it gets the input geometry from files. Section 10 specifies the file formats natively supported by SWIFT. Additional formats may be supported through application defined file readers that may be plugged in. More details on this are given in section 6.6 and section 9.

- **Add_Hierarchical_Pieced_Object**
  This pair of methods should not be invoked if the hierarchy is not turned on. These methods are for adding an object composed of more than one piece where the application provides the hierarchies for one or more of the pieces. Entire hierarchies must be given. In other words, for a piece, either no hierarchy is given or the whole hierarchy is given. SWIFT will build the hierarchies that are not given. Application provided hierarchies are not subject to the scene configuration parameters affecting hierarchies. Those parameters only affect hierarchies built by SWIFT. The levels of the given hierarchies must be convex polyhedra that are closed. SWIFT will process the given hierarchies to ensure that all the criteria

are met. The only transformations that are applied during this process are translations and scalings. The first method is for input geometry stored in arrays while the second is for input geometry stored in files. It is possible to copy select pieces in either of the two methods.

```
bool Add_Hierarchical_Pieced_Object(
            const SWIFT_Real* const* const* vertices,
            const int* const* const* faces,
            const int* const* num_vertices, const int* const* num_faces,
            const int* num_levels, int num_pieces, int& id,
            bool fixed = DEFAULT_FIXED,
            const SWIFT_Orientation* const* orient = DEFAULT_ORIENTATIONSS,
            const SWIFT_Translation* const* trans = DEFAULT_TRANSLATIONSS,
            const SWIFT_Real* const* scales = DEFAULT_SCALESS,
            const int* box_settings = DEFAULT_BOX_SETTINGS,
            const SWIFT_Real* box_enl_rel = DEFAULT_BOX_ENLARGE_RELS,
            const SWIFT_Real* box_enl_abs = DEFAULT_BOX_ENLARGE_ABSS,
            const int* const* const* face_valences
                                        = DEFAULT_FACE_VALENCESSS,
            const int* copy_oids = DEFAULT_COPIES,
            const int* copy_pids = DEFAULT_COPIES,
            const int* min_tri_count = DEFAULT_MIN_TCOUNTS,
            const SWIFT_Real* qslim_tri_ratio = DEFAULT_QSLIM_TRATIOS,
            const SWIFT_Real* cube_aspect_ratio
                                        = DEFAULT_CUBE_ASPECT_RATIOS );

bool Add_Hierarchical_Pieced_Object(
            const char* const* const* filenames,
            const int* num_levels, int num_pieces, int& id,
            bool fixed = DEFAULT_FIXED,
            const SWIFT_Orientation* const* orient = DEFAULT_ORIENTATIONSS,
            const SWIFT_Translation* const* trans = DEFAULT_TRANSLATIONSS,
            const SWIFT_Real* const* scales = DEFAULT_SCALESS,
            const int* box_settings = DEFAULT_BOX_SETTINGS,
            const SWIFT_Real* box_enl_rel = DEFAULT_BOX_ENLARGE_RELS,
            const SWIFT_Real* box_enl_abs = DEFAULT_BOX_ENLARGE_ABSS,
            const int* copy_oids = DEFAULT_COPIES,
            const int* copy_pids = DEFAULT_COPIES,
            const int* min_tri_count = DEFAULT_MIN_TCOUNTS,
            const SWIFT_Real* qslim_tri_ratio = DEFAULT_QSLIM_TRATIOS,
            const SWIFT_Real* cube_aspect_ratio
                                        = DEFAULT_CUBE_ASPECT_RATIOS );
```

The first method adds an object to the scene by reading the geometry from the *vertices* and *faces* arrays. These arrays are three-dimensional to accomodate multiple pieces and multiple levels per piece. Consequently, the *num_vertices* and *num_faces* parameters are two-dimensional arrays containing the number of vertices and faces for each level of each piece. The *face_valences* array acts as previously documented. The arrays are given in "piece-major" order meaning that the pieces form the first dimension, the levels the second, and the vertices or faces the third. The number of pieces composing the object is given in the *num_pieces* parameter. All of the geometry arrays should be of length equal to *num_pieces* in the first dimension (except for the *face_valences* array which may be set to **DEFAULT_FACE_VALENCESSS** which has no length). The second dimension must be of length equal to the *num_levels* element for the current piece since each piece may have a variable number of levels. If the application desires SWIFT to build a hierarchy then the object geometry for that piece must be given as if there was one level. The *num_levels* entry should be set to 0 to tell SWIFT to build the hierarchy for the corresponding piece. If the *num_levels* entry were 1, then SWIFT would assume that only 1 level is desired and not build anything. The identifier of the object is passed back in *id*.

The *fixed* parameter applies to the entire object since an object moves as a whole and all its pieces maintain the same relative placements at all times. The input transformations are given as in the Add_Pieced_Object() method and are all identity by default. If a piece is to have its hierarchy built by SWIFT, there should be only one corresponding input transformation for the piece. Each piece may have different box settings. Recall that there is only a single box setting per piece (not per level). By default all are of type **CUBE** and have no enlargement if the object is moving.

Next, the copy parameters are given. If they are set to **DEFAULT_COPIES** then no copying is done. Otherwise, *copy_oids* and *copy_pids* are arrays that specify which of the pieces are copied and which are not. They are *num_pieces* in length. If an element of the *copy_oids* array is set to **DEFAULT_COPY** then that piece is not copied but rather built from the geometry arrays. Otherwise, the object given by the element of the *copy_oids* array is accessed and one of it pieces is copied to create the current piece. The piece that is copied is identified by the corresponding element in the *copy_pids* array, unless it is equal to **DEFAULT_COPY** in which case the first piece is copied. If the *copy_pids* is set equal to **DEFAULT_COPIES** then the first piece of each copied object is automatically selected. Note that not all objects must be copied or created through geometry. A mix is allowed. The only requirement is that there be geometry present for the pieces that are not copied. For those that are copied, their positions in the geometry arrays may be empty. If a piece is to be copied, there should be only one corresponding input transformation for the piece. Otherwise, input transformations act as before.

The *min_tri_count* parameter specifies the number of triangles to cut the hierarchy creation off at for each piece only if the piece is to have its hierarchy created by SWIFT. If it is set to **DEFAULT_MIN_TCOUNTS** then the default is used. Similarly, the *qslim_tri_ratio* parameter is used to specify the triangle reduction factor when creating a QSlim hierarchy and the *cube_aspect_ratio* parameter is used to specify the aspect ratio below which a cube is used if the bounding setting is **CHOOSE**. These last two parameters can be set respectively to **DEFAULT_QSLIM_TRATIOS** and **DEFAULT_CUBE_ASPECT_RATIOS** to use the defaults. See section 4.4 for more details on these three parameters.

The second method is very similar to the first with the exception that it gets the input geometry from files whose names are given in the *filenames* array. This array must have length equal to *num_pieces* in the first dimension and length equal to the element of the *num_levels* array for the current piece in the second dimension. It may have empty sub-arrays in the positions of copied pieces. Section 10 specifies the file formats natively supported by SWIFT.

Additional formats may be supported through application defined file readers that may be plugged in. More details on this are given in section 6.6 and section 9.

- **Delete_Object**
  This method is for deleting an existing object.

  ```
  bool Delete_Object( int id );
  ```

  The id of the object to be deleted is given in the *id*. This id should not be used any longer unless a subsequent object creation returns this id to the application. Calling this function has the effect of deleting all memory independently related to the object.

6.3 Object Transformation

The only transformations that are allowed by SWIFT are rotations and translations given by 3-vectors and 3 by 3 matrices respectively. Scaling, skewing, and any other transformations are not allowed. If they are mixed into the rotation matrix, SWIFT will malfunction. For maximum query performance, avoid setting an object's transformation more than once per query. Each object's transformation must be set at least once after its creation to ensure correct querying. There are four different ways to set transformations. They differ in whether a single object's transformation is set or all the objects' transformations are set at once, and whether the transformations are given separately or together as 3 by 4 matrices.

- **Set_Object_Transformation**
  This pair of methods is for setting the transformation of a single object.

  ```
  void Set_Object_Transformation( int id, const SWIFT_Real* R,
                                          const SWIFT_Real* T );

  void Set_Object_Transformation( int id, const SWIFT_Real* RT );
  ```

  The first method is for setting the rotation and translation separately. The id of the object (gotten from an Add_*_Object() method) is given in *id*. The rotation and translation are given in row-major order in *R* and *T* as arrays of length 9 and 3 respectively.

  The second method is similar to the first but different in that the rotation and the translation are given together in a single 3 by 4 matrix. The leftmost 3 by 3 is the rotation matrix and the fourth column is the translation vector. This also is given in row-major order as an array of length 12.

- **Set_All_Object_Transformations**
  This pair of methods is for setting the transformations of all the objects at once. The only savings this has over the previous pair of methods is method invocation overhead.

  ```
  void Set_All_Object_Transformations( const SWIFT_Real* R,
                                       const SWIFT_Real* T );

  void Set_All_Object_Transformations( const SWIFT_Real* RT );
  ```

The arrays are given as a sequential stream of row-major matrices, one for each moving object. Fixed object transformations should not be in these lists. If fixed object transformations need to be set, then one of the first pair of methods should be used. The ordering of the objects to which the transformations apply is the same as the order in which they were added to the scene which is the same as the order of their ids.

The first method is for setting the rotations and the translations separately. The rotations and translations are given in row-major order in R and T as arrays of length equal to 9 times the number of moving objects and 3 times the number of moving objects respectively.

The second method is similar to the first but different in that the rotations and the translations are given together as a series of 3 by 4 matrices in the same form as for the second Set_Object_Transformation() method described previously. The array is of length equal to 12 times the number of moving objects.

6.4 Pair Activation

If a pair is active, then it is tested during a query. If it is not, then it is not tested and there will not be any results reported for it. Pairs where both objects are fixed, are never active (and can never be set active). There are six different activation methods. They differ in whether an activation or a deactivation is done, and whether two, one, or no ids are given.

- **Activate**
  These methods are for activating pairs of objects.

  ```
  void Activate( int id1, int id2 );

  void Activate( int id );

  void Activate( );
  ```

- **Deactivate**
  These methods are for deactivating pairs of objects.

  ```
  void Deactivate( int id1, int id2 );

  void Deactivate( int id );

  void Deactivate( );
  ```

  The first group of three methods are for activating pairs and the last group of three are for deactivating pairs. The first method in each group is for changing the activation of exactly a single pair given by the two object ids *id1* and *id2* if allowed (both non-fixed). The second method in each group changes the activation of all pairs containing the object identified by *id* (if allowed). The third and last method in each group is used to activate or deactivate all the pairs in the scene. They can be used to reset the activation of the scene. The result of calling the Activate() method is the same as the default activation provided by the scene. The object ids mentioned above are gotten from the Add_*_Object() methods.

6.5 Query

There are four different ways to query SWIFT. Each of the methods returns a boolean indicating whether intersection was detected anywhere in the scene (among the objects being tested). All the arrays that are used for reporting the results of the query are managed by the SWIFT system. Do ***NOT*** allocate or deallocate them.

- **Query_Intersection**
  This query is for finding the pairs of object pieces that are intersecting in a scene.

  ```
  bool Query_Intersection( bool early_exit, int& num_pairs, int** oids );
  ```

  The return value indicates if there was at least one intersecting pair in the scene. If the *early_exit* parameter is set to **true**, then the computation is stopped when the first intersection is found and **true** is returned but no pairs are reported. If it is set to **false**, then reporting occurs and all intersecting pairs are found. The *oids* arrays (object ids) will be allocated and filled in by this method if intersection is detected. There are 2 times *num_pairs* elements in the array. For example, oids[0] and oids[1] are an intersecting pair (if $num\_pairs > 0$). Note that there may be multiple pairs reported for the same pair of objects if one of them has multiple pieces. Note also that there is no specific ordering in the *oids* array. A return value of **false** as well as $num\_pairs = 0$ indicates that there are no intersecting pairs.

- **Query_Approximate_Distance**
  This query is for finding the error bounded approximate distance between pairs of object pieces that are close enough in a scene. An approximate distance is given along with an error such that the exact distance is in the range [approximate distance, approximate distance+error].

  ```
  bool Query_Approximate_Distance(
                  bool early_exit, SWIFT_Real distance_tolerance,
                  SWIFT_Real error_tolerance, int& num_pairs,
                  int** oids, SWIFT_Real** distances, SWIFT_Real** errors );
  ```

  The return value indicates if there was at least one intersecting pair in the scene. If the *early_exit* parameter is set to **true**, then the computation is stopped when the first intersection is found and **true** is returned but no pairs are reported. If it is set to **false**, then reporting occurs and all close or intersecting pairs are found and their approximate distances reported. The *oids* array functions the same as in the intersection query.

  The distances and errors for each pair are given in the *distances* and *errors* arrays. They are each of length equal to *num_pairs*. Approximate distance is reported if it is within the *distance_tolerance* for a pair. This ensures that any exact distance within the *distance_tolerance* is caught. An error bound equal to *error_tolerance* is guaranteed. If the *broad_phase* setting is turned on for the scene, then the approximate distance is reported only for those pairs whose bounding boxes overlap and whose approximate distance is less than or equal to the given distance tolerance. So, if the approximate distance is desired for pairs to within a tolerance, the bounding boxes of the objects should be enlarged by an absolute amount at least equal to the distance tolerance. Intersecting pairs are reported as having a distance equal to -1.0 and an error of 0.0. The distance and the error reported when there is intersection are ***NOT*** a measure of penetration depth.

- **Query_Exact_Distance**
  This query is for finding the exact distance between pairs of object pieces that are close enough in a scene.

  ```
  bool Query_Exact_Distance(
                  bool early_exit, SWIFT_Real tolerance, int& num_pairs,
                  int** oids, SWIFT_Real** distances );
  ```

  The return value indicates if there was at least one intersecting pair in the scene. If the *early_exit* parameter is set to **true**, then the computation is stopped when the first intersection is found and **true** is returned but no pairs are reported. If it is set to **false**, then reporting occurs and all close pairs are found and their exact distances reported along with any intersections. The *oids* array functions the same as in the intersection query.

  The distances for each pair are given in the *distances* array. It is of length equal to *num_pairs*. Exact distance is reported if it is within the *tolerance* for a pair. If the *broad_phase* setting is turned on for the scene, then the exact distance is reported only for those pairs whose bounding boxes overlap and whose distance is less than or equal to the given tolerance. So, if the exact distance is desired for pairs whose distance is within a tolerance, the bounding boxes of the objects should be enlarged by an absolute amount at least equal to the tolerance. Intersecting pairs are reported as having a distance equal to -1.0. The distance reported when there is intersection is ***NOT*** a measure of penetration depth.

- **Query_Contact_Determination**
  This query is for finding the exact distance between pairs of object pieces that are close enough in a scene and determining additional contact information such as nearest features, nearest points, and contact normals.

  ```
  bool Query_Contact_Determination(
              bool early_exit, SWIFT_Real tolerance, int& num_pairs,
              int** oids, SWIFT_Real** distances = NO_DISTANCES,
              SWIFT_Real** nearest_pts = NO_NEAREST_PTS,
              SWIFT_Real** normals = NO_NORMALS,
              int** pids = NO_PIDS, int** feature_types = NO_FEAT_TYPES,
              int** feature_ids = NO_FEAT_IDS );
  ```

  The return value indicates if there was at least one intersecting pair in the scene. If the *early_exit* parameter is set to **true**, then the computation is stopped when the first intersection is found and **true** is returned but no pairs are reported. If it is set to **false**, then reporting occurs and all close pairs are found reported upon. The *oids* array functions the same as in the intersection query.

The various things that may be reported for a contact are: exact distance, nearest points, contact normals, and nearest features. They are selected individually for reporting by passing an array pointer for the corresponding array to be assigned by SWIFT. To not select an item for reporting, simply pass the default parameter for that array. The default is that all items are deselected.

The nearest points are reported if the *nearest_points* pointer is not the default value. The array will be 6 times *num_pairs* elements long. For each pair, the 6 values are the x,y,z coordinates of the nearest point on the first object/piece followed by the x,y,z coordinates of the nearest point on the second object/piece. If a pair is disjoint, then the actual nearest points are reported. If there are many equally nearest points, then an arbitrary pair is reported. If a pair is intersecting, then the nearest points that are reported are those on the intersecting features (as would be reported as the nearest features). These nearest points are equivalent to the nearest points for the features as if they were floating by themselves free in space. The nearest points are given in each object's local coordinates (before the object transformation that was set is applied but after the input transformation given to an Add_*_Object() method).

The contact normals are reported if the *normals* pointer is not the default value. The array will be 3 times *num_pairs* elements long. For each pair, the values are the x,y,z coordinates of the normal pointing from object 2 towards object 1. The normal is given in global (world) coordinates. A normalized vector is given for the normal. The computation of the normals is discussed next.

If a pair is disjoint, then there are 4 cases for the pair of feature types. These are V-V, V-E, V-F, and E-E. In the V-F case, the contact normal is simply the face normal. In the E-E case, the cross product of the edge direction vectors is the contact normal. The V-V and the V-E cases are degenerate in the sense that for two close objects, they will almost never happen. However, they do sometimes so we need to handle them. In the V-E case, the contact normal is given by a line from the vertex to the nearest point on the edge. For the V-V case, the contact normal is computed as follows. First a normal is computed for each vertex as an average of the normals of the neighboring faces. Then the two vertex normals are averaged to yield the contact normal.

If a pair is intersecting then there are 2 cases for the pair of feature types. These are V-F and E-E. The contact normals are computed the in the same way as the disjoint case for each of these cases.

The nearest features are reported if none of the *pids*, *feature_types*, and *feature_ids* parameters are set to their default values. The piece ids (*pids*) array will be 2 times *num_pairs* elements long and specifies the ids of the pieces that are contacting. The feature types array will be 2 times *num_pairs* elements long. The feature ids array will be at least 2 times *num_pairs* elements long but may be longer due to features which are edges. The feature types are identified by **SWIFT_VERTEX**, **SWIFT_EDGE**, or **SWIFT_FACE** (the three types of features for a polyhedron). The feature ids are given as follows. If the feature is a vertex, a single id is given which is the number of the vertex in the order that it was given to one of the Add_*_Object() methods starting at 0. If it is an edge, two vertex ids are given (one for each end of the edge). Faces are numbered the same way as vertices starting at 0. Note that if faces with more than three edges are given to any of the Add_*_Object() methods, SWIFT triangulates them. Therefore, an edge might correspond to an edge that was not on the boundary of an input face but rather introduced by the triangulation of the face. Note also, that if the vertices that are given to any of the Add_*_Object() methods were duplicated (not fully shared) then the vertex id that is reported for one of these non-shared vertices is the id of an arbitrary instance of the duplicated vertex. This has ramifications for the edge ids that are reported. The endpoints of an edge may have ids such that there exists no edge with those endpoint ids that was given to an Add_*_Object() method.

6.6 Plug-In Registration

6.6.1 File Readers

In order to read files that are not in either of the SWIFT formats, a file reader plug-in may be provided by the application. For more information on file reader construction, see section 9 and SWIFT_fileio.h in the SWIFT/include directory.

- **Register_File_Reader**
  This method allows for the registration of a file reader with SWIFT. It will then be used to read files that start with the given string (magic number). A file reader does not have to be registered for every scene created but only once for the entire program.

  ```
  bool Register_File_Reader( const char* magic_number,
                             SWIFT_File_Reader* file_reader ) const;
  ```

  The first parameter is the string that is matched at the beginning of the file that is read. It is referred to as the file's magic number. The second parameter is a file reader object pointer that references a reader able to read files with the given magic number. If a reader is already assigned to the given magic number than the new reader will replace it. Only one reader can handle a given type of file. A single reader can however read different types of files so this method can be invoked with multiple magic numbers and the same reader. The return value indicates success. On failure a message is written to **stderr**.

7 Ease of Use Tips

SWIFT has some convenient features which can make the application's job easier. They are:

- Use the **CHOOSE** type for bounding boxes so that SWIFT can decide what the best type of box to use is. Keep in mind however, that the application is responsible for setting the threshold aspect ratio for this decision. The provided default is not the best overall solution.

- Use the input transformations to position objects relative to each other. This is especially useful when creating pieced objects. Note that there are pitfalls with using input transformations such as replication of geometry.

- Use the copy feature whenever possible. This is possible when two pieces have the same input geometry (minus an input transformation). This is also more efficient for the precomputation.
- Instead of applying input transformations on fixed objects, apply the transformations as the moving (initial and only) transformations for them. This allows copying if multiple fixed objects have the exact same geometry. This will not work if scaling is involved however.
- Take advantage of the triangulation and vertex sharing features that SWIFT provides.
- Take advantage of the fact that SWIFT reports a large quantity of contact information from the contact determination query. Normally, this would be the application's responsibility but SWIFT includes this functionality to increase overall efficiency of applications needing the standard contact information.

8 Efficiency Tips

There are various things to keep in mind in order to use SWIFT in an optimal manner. They are:

- Only set fixed object transformations at the beginning (before the first query).
- Only set each moving object transformation at most once per query. Also, do not set an object transformation if it has not moved from the previous transformation since setting a transformation incurs an overhead.
- Try to reuse geometry by using the copy feature if at all possible. Avoid using input transformations that will replicate geometry if possible. Although the replication storage cost is not large for a few objects, it can easily become quite large for many objects. Sometimes it is possible to rephrase transformations of motion in terms of the input transformations.
- When querying, only query what is absolutely necessary. The queries are ordered from most efficient to least efficient as:

  1. Intersection
  2. Approximate Distance
  3. Exact Distance
  4. Contact Determination

  For example, when running a dynamic simulation, use the intersection test until an intersection is detected, then use the contact determination query during the bisection search. Furthermore, only ask the contact determination query for the minimal amount of information required (by setting some of the parameters to their default values).
- If not many pieces are moving relative to the total number of pieces, it might be beneficial to consider using local bounding box sorting instead of global bounding box sorting.

- Use pieced objects if an entire set of pieces is to move in the exact same manner.

9 Application Defined Plug-In's

SWIFT allows the application to provide class derivations that act as extensions. Currently, the only extension is a file reader extension where the application can create its own file reader, register it with SWIFT, and use it to access application specific file types. No other simple extensions have yet come to mind but if our users conceive of any useful ones, please let us know. In this section we describe the construction of plug-in's. For plug-in registration see section 6.6.

9.1 File Readers

The header file SWIFT_fileio.h in the SWIFT/include directory includes the abstract base class from which to derive new file readers. An example is given by the *SWIFT_Basic_File_Reader* class (see also SWIFT/src/fileio.cpp). All that must be provided is a method that does reading. The method is given as:

```
virtual bool Read( ifstream& fin, SWIFT_Real*& vs, int*& fs,
                                  int& vn, int& fn, int*& fv ) = 0;
```

The *fin* parameter is the input file stream to be read from. The *vs* parameter is a reference to the vertex coordinate array. The *fs* parameter is a reference to the face vertex index array. The parameters *vn* and *fn* are the number of vertices and faces respectively. The parameter *fv* is a reference to the face valences.

The Read() method is to read the contents of a file referenced by *fin*. The file is open and the position is set to be the beginning. This allows the reader to be able to read the magic number for itself if need be. The *vs* array is to be allocated to a length equal to 3 times the number of vertices (coordinates in 3D) and *vn* set to the number of vertices (not the number of coordinates). The *fs* array is to be allocated to a length equal to the total number of face vertices required to describe every face and *fn* set to the number of faces (not the number of vertex indices). If the faces are all triangular, *fs* will have length equal to 3 times *fn*. In this case, *fv* may be set to NULL which signifies that all faces are triangular (which is the case a lot of times). If not all the faces are triangular, *fv* should be allocated to a length equal to *fn* and each entry should reflect the number of vertices per face. The caller (SWIFT) will be responsible for deallocation of the arrays.

10 SWIFT File Formats

SWIFT provides two file formats in order to import geometry. There are other ways to import geometry such as through arrays or by reading non-SWIFT file types by providing one or more plug-in file readers (see section 9).

This section gives a description of the formats' simple syntax and semantics. The first file format is for triangular models and will be called the **TRI** file format. The other provided format is for general polyhedral models (with faces not necessarily triangular). It will be called the **POLY** file format. White space is ignored in both file formats. They are both ascii.

10.1 TRI format

The TRI format is a file format for triangular models. It supports arbitrary triangular models (some of which may not be suitable for use with SWIFT). Following are the syntax and the semantics:

```
TRI

nv<int> = number of vertices
nf<int> = number of faces

coordinates<real> = list of the vertex position coordinates as reals.
                    There are 3*nv coordinates.

face indices<int> = list of the vertex indices given in CCW orientation
                    for each face.  There are 3*nf indices.
```

First the magic number “TRI” is given. Then the number of vertices, then the number of faces, then the vertex coordinates, then the face indices which index into the vertex list by identifying the vertex position (not the coordinate position). Vertex indices start at 0. An example of a TRI file is included in the distribution in the example/ directory.

10.2 POLY format

The POLY format is a file format for arbitrary polyhedral models. Following are the syntax and the semantics:

```
POLY

nv<int> = number of vertices
nf<int> = number of faces

coordinates<real> = list of the vertex position coordinates as real.
                    There are 3*nv coordinates.

-- for each face
nfv<int> = number of vertices in the face
face indices<int> = list of the vertex indices given in CCW orientation.
-- end for
```

First the magic number “POLY” is given. Then the number of vertices, then the number of faces, then the vertex coordinates. Following are the faces. Each face consists of an integer specifying the number of vertices in the faces followed by the indices of that many vertices. The face indices are used to index the vertex list by identifying the vertex position (not the coordinate position). Vertex indices start at 0. An example of a POLY file is included in the distribution in the example/ directory.

11 Future Work

There are many ways in which SWIFT can be expanded and improved upon. There are no promises on what will be done but we would like feedback from our users if there is future work that they would like to see. Some of our improvement ideas are:

- **Less Geometry Replication:** allow input transformations to be joined to moving transformations so that same-geometry pieces within the same object do not have to be replicated.
- **Articulated Bodies:** allow a scene graph for multiple piece objects. In addition, have the option of detecting self-collision or not.
- **Penetration Depth:** provide approximate penetration depth over all directions or exact penetration depth in a single direction.
- **Convex Composition:** build a hierarchy to speed up testing of objects composed of more than one convex piece.
- **Object Deletion:** delete objects from the scene. This may be useful if an object has been destroyed in some sort of game or simulation and is not needed anymore. Currently, this same effect can be achieved by deactivating the destroyed object with respect to all the other objects in the scene. The memory will not be released however.

Let us know if any of these are of interest.