S W I F T + +

Speedy Walking via Improved Feature Testing for Non-Convex Objects

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 intersection, performing tolerance verification, computing approximate and exact distance, and determining the contacts between pairs of objects in a scene composed of general rigid 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.

A preprocessor is provided with the library. It is called the decomposer and it is used to process models before they are provided to SWIFT++.

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 2001 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

To use SWIFT++, you will need the Qhull library. You will also need to build the decomposer program and the SWIFT++ library. This can be done in Unix/Linux using the provided Makefiles. It can also be done for Windows using the VC++ 6.0 workspace files. See the README file included with the distribution for more details.

3 Architectural Overview

The application makes use of SWIFT++ 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, perform tolerance verification, 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).

3.1 Precomputation Phase

1. The bulk of the precomputation for non-convex or non-closed polyhedra is done by the decomposer program. Closed convex polyhedra can be given to SWIFT++ without any precomputation. Refer to the decomposer manual for instructions on using the decomposer. SWIFT++ can process either decomposition files or hierarchy files. Hierarchy files require a relatively small amount of processing as compared with decomposition files. Hierarchy files are generally about 10 times bigger in size, however.

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

3. Add or copy all of the objects which compose the scene by calling the Add_Convex_Object(), Add_General_Object(), and Copy_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 may suffer. Using Copy_Object() during the query phase should not cause any performance problems.

   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).

4. 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.

5. 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.

6. 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 five choices, the details of which will be explored later. They are: intersection detection, tolerance verification, 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. Most of the options are turned off and must be un-commented to be enabled. The options are:

- **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*.

- **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.

- **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.

- **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.

- **Polyhedral Boundary:** if you are using polyhedra with boundary and are going to be relying on contact feature and/or contact normal computation, **SWIFT_ALLOW_BOUNDARY** should be defined. The details of this reporting in the presence of boundary are explained in more detail in section 6.5. Otherwise SWIFT++ will assume that all the polyhedra it handles are closed with respect to those reports.

- **Only Triangles:** if you are using polyhedra composed solely of triangles, then you should define the **SWIFT_ONLY_TRIS** variable. The polyhedra that are given to the decomposer must have been composed of triangles for this to be defined. Whether it is defined or not, SWIFT++ will function correctly. This setting is for optimization and additional reporting functionality. If the polyhedra are composed solely of triangles then contact features that are edges can be reported with a single id. Otherwise they would be reported by giving the ids of their endpoints. See the next item.

- **Edge Id Reporting:** if all the polyhedra being used are composed solely of triangles then contact features that are edges may be reported by giving a single id. This is done by defining the **SWIFT_REPORT_EDGE_IDS** variable. The **SWIFT_ONLY_TRIS** variable must also be defined or this option does not have any effect.

- **Piece Caching:** this is an optimization that affects intersection, tolerance verification, and distance queries. It involves storing the pair of pieces that yielded the answer in the previous query and trying them first in the next query. This option is good to use if the queries exhibit motion coherence. Turn it on by defining the **SWIFT_PIECE_CACHING** variable. This option and the next two can all be simultaneously enabled to provide faster queries.

- **Priority Directed Search:** this is an optimization that affects distance queries. It involves computing distances between currently closest nodes first to hone in to the minimum distance more quickly. This option is good even if the queries do not exhibit motion coherence. Turn it on by defining the **SWIFT_PRIORITY_DIRECTION** variable.

- **Generalized Front Tracking:** this is an optimization that affects all queries. It involves starting the search between a pair of hierarchies at a "front" that provided the answer for the last query. It avoids unnecessary work involving pairs at the top of the hierarchies. This option is good to use if the queries exhibit motion coherence. Turn it on by defining the **SWIFT_FRONT_TRACKING** variable.

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. Only files describing closed convex polyhedra may be read into SWIFT++ using a plug-in file reader. A plug-in file reader may also be used with the decomposer program to read in non-convex and non-closed models. See the decomposer manual for more details. 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 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 cannot even be changed when copying objects. They 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. The arrays method only allows closed convex polyhedra which the application is responsible for. In the case of files, SWIFT++ will attempt to read the file. It may be a file that is the output of the decomposer in which case the object may be non-convex or non-closed since it was preprocessed appropriately. Otherwise, the file can be read in as a normal model file (possibly with a registered file reader) and must be a closed convex polyhedron. SWIFT++ allows for the copying of any already added object. SWIFT++ will automatically triangulate any convex polyhedral model composed of convex polygonal faces. The decomposer will also apply triangulation to non-convex models. 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.

- **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. The vertices of the input object are transformed according to the input transformation as they are read into SWIFT++'s data structures. Do ***NOT*** attempt to work a scaling factor into the movement rotation matrices. This will cause SWIFT++ to malfunction.

- **Split Type:** this only applies to using the Add_General_Object() method in which the given file is a decomposition file for which the hierarchy must still be constructed. The split type is the split rule used when creating the hierarchy. The application can choose from four options: **MEDIAN**, **MIDPOINT**, **MEAN**, or **GAP**. For more information refer to the section in the decomposer manual entitled *Hierarchy Construction Algorithms*.

- **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 four options: **DEFAULT**, **CUBE**, **DYNAMIC**, or **CHOOSE**. If the **DEFAULT** option is given, the default is **CUBE** if the object is moving or **DYNAMIC** if the object is fixed. The second two choices have already been explained. If the **CHOOSE** option is given, 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.

- **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 tolerance verification, 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 for both relative and absolute enlargements.

- **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. The default is 2.

4.5 Input Geometry

The input geometry for SWIFT++ can come from four different places. The first two are outputs of the decomposer program and are hierarchy and decomposition files. Those two file types are the only way to import non-convex or non-closed polyhedra into SWIFT++. The input geometry that is passed through arrays or by a model description file (that may be read by a plug-in file reader) has to be closed convex polyhedra. If the geometry is passed using arrays, then the vertex array is indexed by the elements of the face array to determine connectivity. Faces given to either SWIFT++ or to the decomposer 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 following geometric properties must be satisfied by all input geometry:

- 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 five 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. Otherwise reporting occurs normally. The five types of queries are:

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

- **Tolerance Verification:** the application provides a tolerance and all pairs of objects that are closer than the tolerance are reported.
- **Approximate Distance:** the application provides a tolerance, an absolute error ϵ_a, and a relative error ϵ_r. SWIFT++ reports all pairs of objects that are closer than the tolerance, and reports the distance between them to within the given error bounds. That is, the reported approximate distance D_a that is less than the tolerance and is in the range $[D_e, max(D_e \times (1 + \epsilon_r), D_e + \epsilon_a)]$ where D_e is the exact distance between a pair of objects.
- **Exact Distance:** the application provides a tolerance and SWIFT++ reports all pairs of objects that are closer than the tolerance, and reports the distance between them.
- **Contact Determination:** the application provides a tolerance and SWIFT++ reports all pairs of objects that are closer than the tolerance along with all contacts between them if the objects are disjoint. Many times, an application is only interested in knowing when two objects come sufficiently close to warrant a response computation and which features are closest. 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 or copying them. There are two different ways to add objects and one way to copy them. Objects can also be deleted.

Each of the creation methods passes back an identifier for the created object which can be used on subsequent references to it when setting transformations, activating, or querying. Each method returns a boolean value indicating success of the creation. If the object creation 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_Convex_Object**
  This method is for adding a closed convex object with the input geometry stored in arrays.

```
bool Add_Convex_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,
            SWIFT_Real cube_aspect_ratio = DEFAULT_CUBE_ASPECT_RATIO );
```

  This 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_Convex_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).

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 this parameter.

- **Add_General_Object**
  This method is for adding an object that is described in a file. The file may be a hierarchy or decomposition file that is the output of the decomposer program or it may a model file. If it is a model file, then it must describe a closed convex polyhedron.

```
bool Add_General_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,
            SPLIT_TYPE split = DEFAULT_SPLIT_TYPE,
            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 );
```

  This method is very similar to the first method with the exception that files are used instead of array for the input geometry. Section 10 specifies the file formats 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.

  The *split* parameter specifies the type of splitting to apply when the input is a decomposition file. See section 4.4 for more details on this parameter.

- **Copy_Object**
  This method is for making a copy of an existing object.

```
bool Copy_Object( int copy_oid, int& id );
```

The id of the object to be copied is given in the *copy_oid* parameter and the id of the newly created object is given back to the application in the *id* parameter.

- **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 created in the scene (which is not necessarily 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() or Copy_Object() methods.

6.5 Query

SWIFT++ computes the center of mass of each object which can be queried.

- **Get_Center_Of_Mass**

```
void Get_Center_Of_Mass( int id,
                         SWIFT_Real& x, SWIFT_Real& y, SWIFT_Real& z );
```

  The parameter *id* specifies the object whose center of mass is being queried. The center of mass that is returned is the center of mass of the object after its input transformation was applied but not any of the other transformations that may have been set for it. Its coordinates are returned in the *x*, *y*, *z* parameters.

SWIFT++ provides five different proximity queries. 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 objects 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 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_Tolerance_Verification**
  This query is for finding the pairs of objects that are close to within some tolerance in the scene.

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

This query behaves the same as the intersection detection query with the exception that the application provides a tolerance indicating how close objects must be in order to be reported. Intersection detection is the same as tolerance verification with a tolerance of 0.

If the *broad_phase* setting is turned on for the scene, then only pairs whose bounding boxes overlap and whose distance is less than or equal to the given tolerance are reported. Thus, if a true tolerance verification is to be done, the bounding boxes of the objects should be enlarged by an absolute amount at least equal to the tolerance. This is done when an object is created.

- **Query_Approximate_Distance**
  This query is for finding the approximate distance between pairs of objects that are close enough in a scene.

```
bool Query_Approximate_Distance(
               bool early_exit, SWIFT_Real tolerance,
               SWIFT_Real abs_error, SWIFT_Real rel_error,
               int& num_pairs, int** oids, SWIFT_Real** distances );
```

The application provides a tolerance, an absolute error ϵ_a, and a relative error ϵ_r. No approximate distances below the tolerance are reported. The reported approximate distances lie in the range $[D_e, max(D_e \times (1 + \epsilon_r), D_e + \epsilon_a)]$ where D_e is the exact distance between a pair of objects. The approximate distances are reported in the *distances* array. It is of length equal to *num_pairs*.

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 distance is less than or equal to the given tolerance. Thus, if the approximate 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_Exact_Distance**
  This query is for finding the exact distance between pairs of objects 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 );
```

This query behaves the same as the approximate distance query with the exception that the application does not provide any error bounds since the exact distance is being reported. The exact distances are reported in the *distances* array. It is of length equal to *num_pairs*.

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. Thus, 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 contacts between objects in the scene. Contact information such as nearest features, nearest points, and contact normals can be reported.

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

  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 *num_pairs* parameter refers to the number of object pairs being reported and not the number of contacts being reported. There are 2 times *num_pairs* elements in the *oids* array. There are *num_pairs* elements in the *num_contacts* array which is used to specify the number of contacts for each pair. If a pair is intersecting, the *num_contacts* entry will be -1. This is considered a single contact between the objects so the non-NULL arrays will have to be advanced appropriately. In the intersecting case, the element of the distances array will be -1 and the contents of the other arrays will not be meaningful. If the objects are not intersecting then the *num_contacts* entry will be greater or equal to 1.

  The nearest points are reported if the *nearest_points* pointer is not the default value. The array will be 6 times the total number of contacts long. For each contact, the 6 values are the x,y,z coordinates of the nearest point on the first object followed by the x,y,z coordinates of the nearest point on the second object. 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. 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 the total number of contacts 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.

The nearest features are reported if none of the *feature_types* and *feature_ids* parameters are set to their default values. The feature types array will be 2 times the total number of contacts long. The feature ids array will be at least 2 times the total number of contacts long but may be longer in some cases 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 that are reported are based upon the input to the decomposer or to SWIFT++, whichever read the model file. They are reported as follows:

- **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 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.
- **Edge:** for an edge there are two cases. The first is that SWIFT++ is configured to accept only triangles (**SWIFT_ONLY_TRIS** defined) and configured to enable edge id reporting (**SWIFT_REPORT_EDGE_IDS** defined) in which case the edge ids that are returned are equal to $3\times$ the face id of the face the edge belongs to $+$ the id of the edge within the face. The id of an edge within a face is either 0, 1, or 2. A face is input as three vertex indices. If the edge spans index 0 and index 1 then it has an id of 0, the edge from index 1 to index 2 has id 1 and the third edge has id 2. If the polyhedron the edge belongs to is not closed, an edge may be found which is not part of the original mesh. In this case, it is reported with an id of -1. This should only happen if the **SWIFT_ALLOW_BOUNDARY** variable is defined.

  Otherwise, the edge is reported as two ids that are the vertex ids of its endpoints. 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. 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. If the polyhedron the edge belongs to is not closed, an edge may be found which is not part of the original mesh. In this case, the two vertices belonging to the edge are still reported but this pair of vertices may not make any sense to the application.

- **Face:** a single id in the same manner as the vertices, starting at 0. If the polyhedron the face belongs to is not closed, a face may be found which is not part of the original mesh. In this case, it is reported with an id of -1.

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 copy feature whenever possible. This is possible when two pieces have the same input geometry and the same input transformation.

- 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. The replication storage cost can quickly become quite large so try to avoid replication if possible.
- When querying, only query what is absolutely necessary. The queries are ordered from most efficient to least efficient as:

    1. Intersection
    2. Tolerance Verification
    3. Approximate Distance
    4. Exact Distance
    5. 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.

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 four file formats in order to import geometry. The hierarchy and decomposition file formats are not required to be understood by a SWIFT++ user so they are not discussed further here. The other two formats are discussed here. They are used to import geometry into the decomposer program as well. If they are used to import geometry into SWIFT++, the geometry must be closed convex polyhedra. 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 scaled objects to share geometry.
- **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.

Let us know if any of these are of interest.