SWIFT++
Speedy Walking via Improved Feature Testing for Non-Convex Objects
http://www.cs.unc.edu/∼geom/SWIFT++/

Decomposer 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, or 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.

In order for models to be used by the SWIFT++ library, they must first be preprocessed. The *decomposer* program, for which this document is the manual, is the program that does most of the preprocessing. It takes a basic model description and converts it into a format that SWIFT++ can use efficiently. Please refer to the application manual for information on using the SWIFT++ library in an application.

Contents

1 License Agreement

The decomposer program and the SWIFT++ library are Copyright 2001 The University of North Carolina at Chapel Hill. The decomposer program uses a slight modification of the RAPID system (included in the distribution) that is Copyright 1995 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 the library is linked with. It also covers redistribution and code modifications. All the source code is C++ and is included in the distribution.

2 Build, Installation, and Execution

Assuming that the distribution has been unpacked into the directory SWIFT++/, follow these steps to get a built decomposer:

- If compiling using VC++ 6.0 use decomp.dsw found in the SWIFT++/decomposer/ directory.
- Else if compiling for some Unix, do the following:
  - Look through SWIFT++/Makefile and set CC to point to the proper compiler and CFLAGS to the desired compile flags.
  - Make sure that the Qhull library exists somewhere on the system. Make sure that the QHULL_DIR variable in SWIFT++/Makefile points to its location (the Qhull include files as well as the compiled library should be in the same place).

  - Type “make decomposer” in the SWIFT++/ directory. This will cause a couple of libraries to be compiled as well as the source for the decomposer program. The decomposer is then created by linking. The default decomposer is the command line-only one. A graphical one may be created by typing “make decomposer_g” in the SWIFT++/ directory. The graphical decomposer must link to the OpenGL and TCL/TK libraries (modify the LIBDIRS_G and LIBS_G variables in decomposer/src/Makefile). It may also be used in command line mode (by giving the **-g** command line option). There may be a few warnings throughout the process which can be safely ignored. The decomposer program will be placed in the SWIFT++/bin/ directory and must be run from there if a graphical decomposer was created (SWIFT++/bin/ must be the current working directory).

- Else you will have to create your own compiler configuration.

To understand how to run the decomposer program give the command line option “-h” or read the rest of this manual.

3 Overview

SWIFT++ takes model descriptions as input in order to query proximity information about them. In order to make SWIFT++ perform efficiently, some preprocessing is performed on the models. This includes decomposing their surface into convex pieces followed by constructing a hierarchy on the pieces. Both of these types of preprocessing are done by the decomposer.

Models to be used by SWIFT++ must first be run through the decomposition program if they are non-convex. If there are no errors, the resulting file can be loaded into the SWIFT++ library (by passing the filename of the file resulting from the preprocessing).

The preprocessing usually works on the first try but sometimes it fails. Various reasons for this, as well as how to get around problems are discussed further in section 9. Next, a description of the preprocessing pipeline is given, followed by a discussion of the command line options.

4 Preprocessing Pipeline

There are four stages to the preprocessing. The first two are optional and are mainly used for getting around geometric problems and for optimization. The intermediate results may be saved to files after any of the last three. The input to the pipeline is a model file in the **TRI** or the **POLY** format, a decomposition (*.dcp*) file that was previously saved, a hierarchy (*.chr*) file that was previously saved, or a user defined file format that is read by means of a plug-in file reader (see section 10.2). If the input is a decomposition file, then the first three stages are not performed. If the input is a hierarchy file, then neither is the fourth. In the other cases, the fourth stage is only performed if it is enabled. The stages are:

1. **Jitter:** The coordinates of all of the vertices in the input model are jittered randomly with an amplitude (up to an amount) that is specified by the user. This is useful if there are a lot

of flat edges in the model and the decomposer is giving errors. Usually a small perturbation of the vertex positions will allow the preprocessing to go through.

2. **Edge Flip:** Sometimes there are a lot of nearly flat edges in the model which are non-convex. Edge flipping may be used to introduce a small amount of user defined absolute error in exchange for a "more convex" model which may yield a smaller number of pieces when decomposed. By reducing the number of pieces, the memory requirements and the query time are both reduced. The results after edge flipping may be saved to a **TRI** file. Usually these files are given the extension *.tri*.

3. **Decompose:** The surface decomposition *must* be done by this program. SWIFT++ will function incorrectly if you try to give it a non-convex model that has not been decomposed. There are three decomposition variations available. The output of the decomposition may be saved to a file. The file is binary and is usually given the extension *.dcp*. Decomposition files may be read into SWIFT++.

4. **Build Hierarchy:** A hierarchy is built and may be saved to a binary file that is usually given the extension *.chr*. There are four splitting methods to choose from. Hierarchy files may be read into SWIFT++ and are the most efficient to use since the SWIFT++ system will simply have to read the file rather than compute the entire hierarchy when it starts.

Statistics and status information is printed out after each stage. Refer to section 5 for the details about how the user specifies the parameters to the above process. Some brief descriptions of the algorithms that are used in the decomposer are given in sections 7 and 8.

The features (vertices, edges, and faces) present in the input **TRI** file, **POLY** file, or in the arrays produced by a plug-in file reader, are the ones that the feature reporting mechanism in the SWIFT++ query functions is based upon. If edge flipping is done and the application wishes to have SWIFT++ report features, the user should make the decomposer save the edge flipped file so that the application can see which features SWIFT++ is referring to. In any case, the application may wish to have their own copy of the data so that features can be identified. SWIFT++ currently does not provide interaction with its internal data structures.

5 Command Line Syntax and Options

The decomposer program is run according to the following syntax:

```
decomposer [options] input_filename
```

The input filename may refer to a file of type **TRI** or **POLY**, to a decomposition file, or to a hierarchy file. The options that can be given to the program on the command line are:

-h or -help Print help information including these options and their descriptions

-g If a graphical decomposer has been built, it will run in command line mode (no graphics)

-j ampl Jitter the input at the given amplitude (stage 1)

-e err Edge flip using the given absolute error (stage 2)

-ef filename Save edge flip results to a file

-1 Decompose the model into only 1 piece. This is useful for forcing the decomposition to yield a single piece. An example where this might be useful is in decomposing a cube composed of triangles. The decomposer may create more than a single piece due to flat edges. When using this option, the user is responsible for ensuring that the model is in fact convex.

-dfs Run the plain DFS decomposition (stage3)

-bfs Run the plain BFS decomposition (stage3)

-cbfs Run the cresting BFS decomposition (default stage3)

-df filename Save decomposition to a file

-hier Build hierarchy (stage 4)

-hf filename Save the hierarchy to a file. This option builds the hierarchy if needed.

-s split Type of splitting when building the hierarchy. Can be one of “MED”, “MID”, “MEAN”, or “GAP”. See section 8 for the meaning of these.

The default behavior of the program is:

- No jittering
- No edge flipping
- Cresting BFS decomposition is used when decomposing
- Hierarchy is not built
- Split type is “MID” when building the hierarchy
- There is no saving of any results.

6 Graphical User Interface

To be able to visualize either a decomposition or a hierarchy, you must build the graphical decomposer. To get a graphical user interface, simply run the program with the file you want to visualize. It will decompose (if needed) and build the hierarchy (if specified). There are various keystrokes that perform functions in the main view window. The mouse is used to navigate.

There are two main modes: decomposition and hierarchy. A decomposition of the model is what is shown first. If the input is not a hierarchy file and the decomposer was not started with either of the **-hier** or **-hf** options, the hierarchy mode will not be available. To switch between the two modes, press the keys ‘d’ or ‘h’ or choose the mode from the menu on the right panel.

In either mode, navigation is possible by using the mouse. All three buttons can be used. Sometimes holding down <Shift> modifies the action of a mouse button. The functionality associated with holding each of the three buttons down while moving the mouse is:

1. Moving the mouse sideways causes the world to rotate around its center and about its Z axis. Moving the mouse up and down causes the world to tilt. With the <Shift> key depressed, moving the mouse keeps the viewing direction the same but moves the viewer’s position.

2. Moving the mouse sideways causes the viewpoint to translate sideways. Moving the mouse up causes the viewer to move closer to the world. Moving the mouse down causes the viewer to move further from the world. With the <Shift> key depressed, moving the mouse is like driving a car. If you move up, the “car” goes forward, while moving down makes it go backwards. Moving side to side is like steering.

3. Moving the mouse sideways causes the viewpoint to translate sideways. Moving the mouse up or down causes the viewpoint to translate up or down. With the <Shift> key depressed, you are in “trackball” mode and moving the mouse causes the model to rotate about its center in whatever direction the mouse is being moved in.

There are various settings in the Display section of the panel to the right. They are:

- **Backface Culling:** Draw backfacing triangles. If what you are looking at is not composed of solids you can turn this on to see more. This is on by default.

- **Wireframe:** Draw the geometry in wireframe. This is off by default.

- **Color:** Draw the wireframe in color. This is on by default.

- **Axes:** Turn on or off the axes. The world axes are the green ones and the axes at the center of the model are the blue ones. This is on by default.

- **Center World:** Centers view on the world center. This is also accomplished by keystroke ‘C’. This is useful if you get “lost” in the scene. The program starts off showing this view.

- **Set Center:** Save the current view as the user’s center. This is also accomplished by keystroke ‘s’. The saved view can be recalled later by the Center User command.

- **Center User:** Centers view on the user’s center. This is also accomplished by keystroke ‘c’.

The settings in the Visualize section that are common to both modes are:

- **Explode:** Toggle moving the shown convex polyhedra outwards from the object’s center to be able to more clearly see the structure. This is off by default.

- **Virtual Faces:** Draw the faces that were introduced by the decomposition or hierarchy building process (i.e. non original model faces). This is off by default in decomposition mode but on by default in hierarchy mode.

- **Three Color:** Draw the triangles in one of three colors depending on their classification (**ORIGINAL** in green, **CONTAINED** in yellow, and **FREE** in red). Original faces are normally assigned a random color to sort of show what piece they belong to while non-original faces are drawn in gray. This is off by default.

If in decomposition mode, there are two additional items in this section. The edge convexity toggle will turn on or off drawing of a wireframe mesh over the model showing the edge convexity. Green edges are convex edges and red edges are non-convex edges. The text field shows the pieces currently being shown. If the text field starts with the letter 'A' then all the pieces are shown (this can quickly be done by pressing 'a' without clicking on the text field). Otherwise, the text field is parsed and the specified pieces are shown. After typing the entry, press <Enter> to cause the entry to take effect. An example entry like "0,2-9,200" would cause the pieces with indices 0,2,3,4,5,6,7,8,9, and 200 to be drawn.

In hierarchy mode the only additional item in the Visualization section is the upper leaves toggle. If turned on, it will force the leaves at levels above the current level to be drawn. This may be useful if one wishes to see all the leaves when at the lowest level of the hierarchy.

Various keystrokes can change the visualization. They are:

a In decomposition mode, show all the pieces (all are shown initially).

b Toggle backface culling.

c Center the view on the user's center.

C Center the view on the world center.

d Change to decomposition mode.

h Change to hierarchy mode.

j or - If in decomposition mode, cycles through the pieces (showing one at a time) in descending order. The piece index is shown in the text field in the Visualize section. In hierarchy mode, descends the hierarchy (towards the leaves) showing entire levels at a time.

k or + Same as the previous set of keystrokes, but functioning in the opposite direction.

q Quit.

s Save the current view as the user's center.

t Toggle three color mode.

v Toggle drawing of virtual faces.

w Toggle wireframe mode.

x Toggle exploding the pieces out from the center of the object.

7 Decomposition Algorithms

The decomposer program implements three variations of the same basic algorithm. A graph search (either DFS or BFS) is done over the dual graph of the model. Faces are included in a convex patch if they meet certain criteria. Basically, a convex hull is maintained which is never allowed to intersect the original model's surface in any place other than the faces that belong to the patch. When no more faces can be included during a search, the patch is complete. The convex hull that is maintained is called the convex piece. It will become a leaf in the convex hull hierarchy. The graph search starts off with a face that is not yet included in any other patch (faces can only belong to a single patch). Choosing the first face is where the cresting BFS method differs from the other variations. It attempts to choose a starting face that is distant from non-convex edges to give the patch as much opportunity as possible to become large. For more details on this and other possible decomposition algorithms, refer to the publications that go with SWIFT++ (available at the website).

8 Hierarchy Construction Algorithms

A hierarchy is built from the set of convex polyhedra which are the convex pieces produced by the decomposition. They become the leaves of the hierarchy.

First, all the leaves have their centers of mass (COMs) computed. The hierarchy is built recursively starting at the root. The convex hull of all of the leaf COMs is computed and its direction of maximal spread is computed. This direction is the axis along which the splitting of the leaves into two groups occurs. The COMs are projected onto this axis and sorted along it. The four available splitting methods come in at this point. The “MED” method will split the leaves into two equal-sized groups such that one group is entirely to one side of the other along the axis. The “MID” method will compute the midpoint of the spread of values along the axis and use that as the split point. The “MEAN” method computes the average of the values along the axis and uses that as the split point. The “GAP” method is a little more involved and basically tries to look for a large gap in the values which is then used to split them.

The convex hull of each group of leaves is computed. These convex hulls become the children of the root. The next level is built from the subsets of leaves. This continues recursively until the groups have only one leaf.

9 Decomposer Failure and “What to do about it”

Sometimes the geometric algorithms in the decomposer will fail. This is a problem inherent in using floating point computation to process geometry. There are several errors that are checked for by the decomposer. It will report them when they are found. When you get an error, it is not the end of the world. You can retry the process and it will likely work. Here are some tips:

- Try giving a small jitter value to the decomposer. This will cause the vertices to move slightly which may perturb the input away from a bad condition. This will change the model so it does introduce some error.

- Try edge flipping by giving a small absolute error to the decomposer. The program will report how many edges were flipped.
- Try using a different decomposition variation.
- If the decomposer is failing in the hierarchy stage, try using a different splitting method.

Sometimes the input model is corrupt in terms of its geometry or connectivity. There is only so much the decomposer program can do with these models.

10 Decomposer Model Input File Formats

As stated in section 4, if the application desires feature reporting, the features that SWIFT++ reports are based on the original **TRI** files, **POLY** files, or plug-in reader arrays that were given to the decomposer as input. First, the SWIFT++ file formats are described, followed by the description of a mechanism whereby the user can read in files in any format.

10.1 SWIFT++ file formats

The decomposer provides two file formats in order to import ordinary geometry. It can also handle decomposition and hierarchy files, but the user is not expected to create these on their own. This section gives a description of the formats' simple syntax and semantics. The first file format is for triangular models and is called the **TRI** file format. The other provided format is for general polyhedral models (with faces not necessarily triangular). It is called the **POLY** file format. White space is ignored in both file formats. They are both ascii.

10.1.1 TRI format

The **TRI** format is a file format for triangular models. It supports arbitrary triangular models. 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, and finally 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.

10.1.2 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.

10.2 User-Defined File Formats

The SWIFT++ library provides 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. The decomposer can make use of this.

The header file SWIFT++/include/SWIFT_fileio.h 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). A method that does reading is all the code that must be provided. In order to read files not in a SWIFT++ format, edit main.cpp in the SWIFT++/decomposer/src/ directory and put the new code there.

The second thing to do is to register your file reader with SWIFT++. Look for the following comment in main.cpp:

```
// Insert plug-in file reader registration here
```

and insert the instantiation and registration of your file reader. The decomposer will have to be built after this.