Chapter 10.
Save and Restore Overview

Topic: *SAT Save and Restore

ACIS saves, or stores, model geometry information to ACIS save files (also known as SAT, part save files, or part files). ACIS also restores model information from these files. These files have an open format. Geometric entities are saved in the form of an entity list.

ACIS provides the API functions api_save_entity_list and api_restore_entity_list for saving models to and retrieving models from a disk file. Two additional API functions, api_save_entity_list_file and api_restore_entity_list_file, allow models to be saved and restored to media other than a stream file.

Applications use these functions in many ways. Single or multiple ENTITYs can be saved in a file, and applications can intermix their own data with ENTITY_LIST data. For testing or debugging purposes, it is possible to edit a file and remove all but the data of interest and load the file into a test or demonstration application, such as Scheme AIDE.

There are two types of ACIS save files: Standard ACIS Text (file extension .sat) and Standard ACIS Binary (file extension .sab). The only difference between these files is that the data is stored as ASCII text in a .sat file and in binary form in a .sab file. The organization of a .sat file and a .sab file is identical; the term SAT file is generally used to refer to both types.

Text file saves have the advantages of being human-readable and being portable between different platforms. Binary saves are not human-readable and may not be portable across platforms, but writing records in binary is roughly twice as fast as writing in text and the files are significantly smaller. Write and read binary files only by the same version running on the same type of hardware. ACIS can read and write any format of binary file on any platform. The selection of binary file format to be written is controlled by the binary_format option. The default format for writing a binary file is the native format of the machine on which it is being written.

Two sets of information may be available at any one time. There should always be information about what is going to be saved. That is stored in a FileInfo object and queried with the env:save–product–id/units Scheme extension. If one has recently retrieved a file, there should also be information about that file. It is stored in a FileInfo object and queried with the env:restored–product–id/units Scheme extension. Of course, both of these can be accessed through C++ by obtaining and querying the appropriate FileInfo object in the kernel context object.

Beginning with ACIS Release 6.3, it is **required** that the product ID and units be populated for the file header (using class FileInfo) before you can save a SAT file. Refer to the reference templates for class FileInfo and function api_set_file_info for more information.

API Functions for Save and Restore

Topic: *SAT Save and Restore

The api_save_entity_list function accepts the following arguments:

- An open output stream pointer (FILE*)
- A logical indicating the file mode of text or binary
- An ENTITY_LIST pointer

The function saves the entity to the output stream.

Top level entities (e.g., body entities) are always the first records in the save file. Thereafter, the data records are in no particular order. A call to api_save_entity_list with a list of *n* top level entities always places these entities in the first *n* records of the save file.

api_save_entity_list_file and api_restore_entity_list_file also control reading and writing of data during a save or restore. Each of these procedures take two arguments: a FileInterface pointer and an ENTITY_LIST pointer.

FileInterface is an abstract base class. By deriving classes from FileInterface, ACIS models can be saved and restored to media other than a stream file. For example, it is possible to derive FileInterface classes that allow models to be saved and restored to a block of memory or by using a pipe to transfer data between two processes.

There are classes derived from FileInterface for saving and restoring in binary or text format to stream files. These are defined in the sabfile.hxx and satfile.hxx files.

If a new type of FileInterface class is being derived, derive it from BinaryFile, which is defined in the binfile.hxx file. The BinaryFile class provides most of the virtual methods of the FileInterface class and only the four virtual methods that are specific to the derived class need to be implemented.

Applications are responsible for opening and closing the file, and for making sure that the file pointers are positioned correctly when the save or restore function is called. Each function expects that the file is open and positioned at the byte where the save or restore is to begin. When the function finishes, the file is positioned after the last byte of the body or entity saved or restored.

The process of copying an ENTITY_LIST is analogous to performing a save and restore. The copy algorithm is described with the save and restore algorithms.

Problems can occur in several areas if you use two different C runtime DLLs (e.g., one release and one debug) when using ACIS. Using the release DLL with an application debug configuration is a common source of access violations for save and restore. Refer to the “C Runtime Library DLL” topic in the Application Development Manual for more details.

Integer identifiers, unique to a given history stream, can be requested for each entity. These IDs remain constant through roll, save and restore. An entity can be returned from a given ID when the entity is alive in the active state of the history stream. NULL is returned when the entity is not alive. This requires two SAT files changes: an extra integer in each entity and the number of tags assigned in the history stream.

The format of ENTITY and HISTORY_STREAM records accommodates the addition of entity IDs. An integer field in the entity record holds the entity’s ID. A value of –1 indicates that an ID has not yet been requested for the ENTITY. This is field #2 in any entity record, where the entity type is field #0. An integer field in history stream records (part of the history data section) holds the next available entity ID in that stream. This is field #3 in the history stream record, where “history_stream” is the first field.

Body Copy

Topic: *SAT Save and Restore

The process of copying a body is identical to that of save and restore, except that it omits the step of saving an intermediate representation of the body to secondary storage. The algorithms make use of the class ENTITY_LIST, which is a variable length associative array implementation used by many of the ACIS algorithms. ENTITY_LIST contains pointers to ENTITYs.

An ENTITY_LIST has four methods that pertain to copying a body:

add . . . . . Adds an ENTITY to the list if not already there and always return the index.

lookup . . . . . Searches for an ENTITY in the list and returns the index.

init . . . . . Prepares to walk the list.

next . . . . . Returns the next undeleted item.

The following is the algorithm, with pseudo code, for copying a body.

1. Set the version information so that the pointers are arranged correctly:

```
restore_major_version = get_major_version ();
restore_minor_version = get_minor_version ();
restore_version_number =
    PORTMANTEAU(restore_major_version,
        restore_minor_version);
```

2. Create an empty ENTITY_LIST to hold a pointer to each data structure in the body:

```
ENTITY_LIST list;
list.add(entities_to_be_copied);
```

   Seed the list with all entities to be copied.

3. For each entity in the list, add each of its ENTITY pointers to the list:

```
list.init();
for(int num_ents; ++num_ents) {
    ENTITY* this_ent = list.next();
    if(this_ent == NULL)
    break;
    this_ent->copy_scan(list);
}
```

4. Continue to scan until all pointers are added to the list.

   For example, the copy_scan method for BODY adds the LUMP pointer to the list:

```
list.add(lump());
```

5. Allocate an array of ENTITY pointers to hold copies of each data structure:

```
ENTITY** array = new ENTITY* [num_ents];
```

6. Copy each entity, replacing every pointer in the copy with an index:

```
for(int i = 0; i < num_ents; i++)
    array[i] = list[i]->copy_data(list);
```

7. Copy the LUMP pointer to a new body:

```
new_body->lump_ptr = (LUMP*)list.lookup(lump());
```

   The copy_data method for BODY, for example, looks up the LUMP pointer in the list and inserts the index in the LUMP pointer of the copy. The LUMP index is cast to a LUMP* to satisfy the C++ compiler.

8. For each entity in the array, replace the pointer indices with pointers corresponding to the new entities:

```
for(int i = 0; i < num_ents; i++)
    array[i]->fix_pointers(array);
```

9. The fix_pointers method for BODY finds the LUMP pointer in the list from the index:

```
set_lump((LUMP*)read_array(array, (int)lump()));
```

The copied body in array[0] is returned and the array is returned to free storage.

Save and Restore Algorithms

Topic: *SAT Save and Restore

The save algorithm works much like the first stage of copying.

1. Create an empty entity list to hold a pointer to each data structure in the body. Seed the list with the BODY pointer.

   ```
   ENTITY_LIST list;
   list.add(body_to_be_saved);
   ```

2. For each entity in the list, write out the class data. Write any pointers as integer indexes by adding them to the list. Continue to scan until all pointers have been added:

   ```
   list.init();
   for(int count = 0; ; ++count) {
       ENTITY* this_ent = list.next();
       if (this_ent == NULL)
           break;
       this_ent->save(list);
   }
   ```

The LUMP pointer in the BODY is written out as:

```
write_ptr(lump(), list);
```

This call saves the pointer to the file as an index and adds the lump to the entity list.

The example in Figure 10-1 is a body (B_0) with two lumps (L_0 and L_0'). The save begins with the scan that takes the pointers from the body and entities contained within and makes an ENTITY_LIST, assigning an index to each pointer. The body pointer is index zero (0).

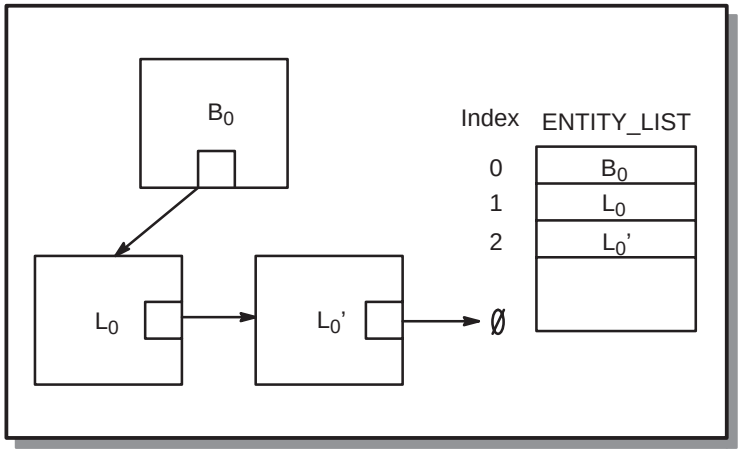

Figure 10-1. ENTITY_LIST Taken from Body

The save algorithm is analogous to the first two stages of copying. The intermediate form of the copied body is now on secondary storage instead of in the entity array.

Restore can be broken into two stages.

Restore Stage 1

Stage 1 reads the save file record and constructs the new object, leaving pointers to other restored entities in symbolic form (as indices into an array), and entering the address into the array.

The restore_entity_from_file function reads the ID string from the start of each save file record and searches its tables for the ID components, starting with the last (base) one and proceeding towards the leaf. It then calls the restore method for the leaf–most class found. That function constructs an object of the appropriate derived type, and then calls its restore_common method. That method in turn calls its parent's restore_common method, and then reads data specific to the derived class. Finally, restore_entity_from_file reads any more data in the save file record, and constructs an unknown_entity_text record to contain it, together with any unrecognized ID strings.

Restore Stage 2

After all new objects have been constructed, stage 2 visits each to convert the symbolic ENTITY pointers into genuine ones.

The fix_pointers method for each entity in the array is called with the array as argument. This calls fix_common, which calls its parent's fix_common, and then corrects any pointers in the derived class. In practice there is never anything special for fix_pointers to do, but it is retained for consistency and compatibility.

The restore process rebuilds the entity array as it reads the records from the file. Then the pointers are reconstructed as in copying. The restore process is:

1. Read the header to get the version number and set it; otherwise, the restore is not guaranteed to work correctly:

```
restore_version = read_int();
restore_major_version =
MAJOR_VERSION(restore_version_number);
restore_minor_version =
    MINOR_VERSION(restore_version_number);
```

2. Read the header to find the number of records and create an array (Figure 10-2) to hold a pointer to each restored record.

```
ENTITY** array = new ENTITY* [num_ents];
```

3. Restore each entity from the file leaving the pointers as integers.

```
for(i = 0; i < num_ents; i++)
    array[i] = restore the ENTITY
```

4. For each entity in the array, replace the pointer indexes with pointers corresponding to the new entities.

```
for(i = 0; i < num_ents; i++)
    array[i]->fix_pointers(array);
```

The first *n* entities in the array (array[0] to array[n–1], where *n* was read from the save file header) are returned and the array is returned to free storage.

While restoring the example body, the new body contains pointers to the new lumps. Lump L_1 contains a pointer to L_1',and the pointer of L_1' is NULL (indicated by *–1*). This is the data stored in the disk by the save algorithm. (Figure 10-2).

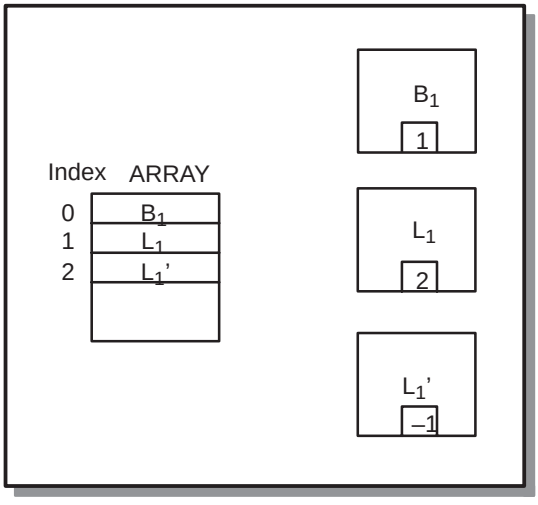

Figure 10-2. Array Created for Restore

Figure 10-3 shows the new body after fix_pointers is called for each entity in the array.

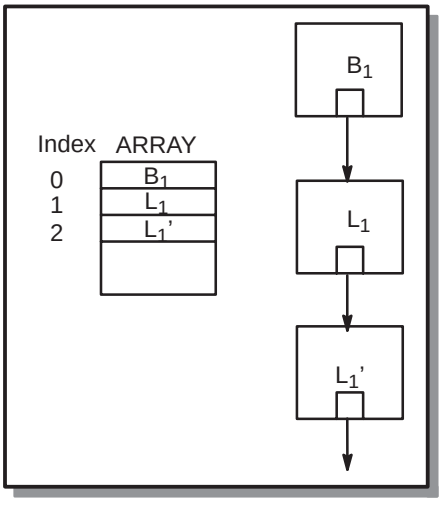

Figure 10-3. Body Recovered by Restore Algorithm

Each class derived from ENTITY has a save method, and the object is saved by invoking the save method for the class.

Restoring is more difficult. ACIS uses a function that is not a class method to restore each class derived from ENTITY. When the ACIS modeler begins execution, each class derived from ENTITY notifies the restore code of its identifier and restore function. During the restore process, the restore code deciphers the identifier and calls the appropriate function.

Part Restore Progress Meter

Topic: *SAT Save and Restore

A restore progress callback mechanism has been added to the SAT/SAB restore functionality. This callback provides the restore progress information needed by an application to implement a progress widget. An example of how the data can be used to implement a Windows progress bar is included in the scm/main/windows/wstdio.cxx source file.

The progress mechanism uses either the entity count or the size of the restore file to measure progress. When the entity count is available it is used by default. Otherwise an attempt is made to acquire the restore file size, and if successful is used to measure progress. The customer can change the progress measurement by setting a custom file size during the very first call to the callback function. This is useful when the file size was not obtainable or the customer has a better understanding of the ACIS data block within the restore file that possibly contains customer data or multiple ACIS data blocks.

The callback frequency can be managed by the customer and is by default set to cause the callback to get called roughly every percent. This can be set to cause the callback to get called more or less often as desired.

The callback mechanism is enabled by installing a custom callback function with the set_restore_progress_callback function. The set_restore_progress_callback function accepts one argument, which is the custom callback function.

An example call to install the callback looks like this:

```
set_restore_progress_callback( my_restore_progress_callback);
```

The installed callback function will be called during every restore operation unless it is either disabled (discussed later) or uninstalled by calling the set_restore_progress_callback with a NULL argument.

An example call to uninstall the callback looks like this:

set_restore_progress_callback(NULL);

The custom callback prototype is typedefed to proc_restore_progress_callback in the savres.hxx header. The function receives a pointer to a restore_progress_data object, which provides all of the progress information, as an argument and returns an integer as a success indicator.

A simple example of a custom callback implementation looks like this:

```
#include ”kernel/acis.hxx”
#include ”kernel/kerndata/savres/savres.hxx”

int my_restore_progress_callback( restore_progress_data * pd)
{ ¿ }
```

The restore_progress_data object has 4 methods available to the application that can be used to modify the default behavior of the callback and to retrieve the progress data. They are public exported methods of the restore_progress_data class and have the following signatures:

```
int count() const;
int index() const;
int size( int input_size = –1);
int interval( int input_interval = –1);
```

The count method returns a positive integer count that is either the number of entities to restore or the size of the restore file when the entity count is not available, or a zero when none of this information is available or obtainable.

An example call looks like this:

```
if( pd–>count() > 0 ) { ¿ }
```

The index method returns a positive integer of the achieved progress, which can be either the number of entities restored or the number of bytes read from the file, or a zero on the very first callback call. The customer can modify the progress measurement during the very first call as mentioned above.

An example call looks like this:

```
if( pd–>index() == 0 ) { ¿ } // first call
```

The size method returns a positive integer of the acquired size of the restore file, or a zero if the entity count is available or the file size could not be obtained. The size method accepts an integer input size from the customer on the very first callback call useful to replace the default restore measurement.

An example call looks like this:

```
if( pd–>size() > 250000 ) {¿} // file size greater than
250000 bytes
```

The interval method returns a positive integer indicating the current callback frequency, which is set to 1 by default. The method accepts an integer indicating the customer desired callback frequency on the very first call to the callback. A frequency of 1 causes the callback to get called roughly every percent. A frequency of 2 causes the callback to get called for every entity restored, which could be quite often. Setting the frequency to zero will disable the callback for this restore only.

An example call looks like this:

```
if( pd->size() > 0 && pd->size() < 250000 )
    // don't bother with the progress meter if the restore is
minimal
pd->interval(0);
    else
        pd->interval(1);
```

Alternatively, one can simply return a –1 from the custom callback to disable the callback for the current restore.

Optional Sequence Numbers

Topic: *SAT Save and Restore

The indexing of the entity records depends on the active ACIS options when the model was saved. If they are indexed, the indexing is sequential starting at 0. All top level entities must appear before any other entities. Thereafter, the record order is not significant.

The optional sequence number represents the index assigned to a record and is intended to improve readability and simplify editing of ACIS save files. The option sequence_save_files controls whether ACIS writes sequence numbers to the part save file.

```
-0 body $1 $2 $-1 $-1 #
    .
    .
    .
-25 point $-1 10 0 25 #
```

In the example above from a SAT file, "–0" and "–25" are sequence numbers. In the first line, "$1 $2" happen to be pointers to records (not shown) with sequence numbers "–1" and "–2", respectively.

This sequence number itself can be turned on or off for the entity records in the save file. Even when the sequence number is not written to the file, it is implied by the order of the records in the file. Pointers to other records correspond to these implied sequence numbers. If sequence numbers are turned off, a record cannot be simply moved or removed from the save file, because this will create invalid index referencing when the file is restored.

If sequence numbers are turned on, an entity may be deleted by simply removing its record from the save file. Any references to the removed record's index become NULL pointers when the file is restored by ACIS. With sequence numbers on, records may also be rearranged within the file.

A mixture of records with sequence numbers and records without sequence numbers is permitted within the save file. Any record having a sequence number is given an index one greater than the previous entry in the file. Specified sequence numbers can be in any order. However, care must be taken that no sequence number repeat itself, either through manual specification of sequence numbers or the implied incrementing of other, nonspecified, sequence numbers.

Regardless of what sequence numbers they contain, the entities represented by the first records are assumed to be the top-level entities. Top-level entities are part of the ACIS topology. Also, if the total entity count was written in the header and the last record's sequence number is not one less than that count, a dummy record with a sequence number equal to the count must be added at the end of the file.

Backward Compatibility

Topic: *SAT Save and Restore

The ACIS version number for subsequent save operations is set using the function set_save_file_version, which takes two arguments, a major version and a minor version.

Throughout the system, all entity save functions, and related functions for curve, surface, etc., take into account the version number. This is the global value, save_version_number, which gives the version of save file format being written. This can be used to produce a save file in a previous version's format.

Objects can be modified to be compatible with an earlier version. This may or may not result in loss of information. If the modification involves loss of information or a possible error in the resulting save file, a warning is generated, but the save file is still produced.

When restoring objects with shared subtypes, such as int_cur or spl_sur, from a save file in which they are not shared, the default is to share identical sub-objects rather than to leave them unshared.

Note *The restore of pre-Release 1.6 save files causes the coedges in the entity restored to be ordered counterclockwise about their edge.*

The version number is normally the current version, but the version may be set backwards to simulate save files generated by previous versions. A major version of 0 or less causes the version to default to the current ACIS version.

Set the save file version number:

```
void set_save_file_version(
    int = 0,           // major version number
    int = –1           // minor version number, default gives
                       // error unless major version is zero
    );
```

Decimal Point Representation

Topic: *SAT Save and Restore

Some applications using earlier versions of ACIS were written using “internationalization” concepts and have written save files that contain representations of double precision numbers containing commas for decimal points. This occurs if the application writing the save file had made a call to the C standard library function setlocale to change the locale properties from the default C settings. Consequently, other applications not using this “internationalization” feature could not read these files.

Beginning with release 2.1, ACIS *always* writes a save file using the C locale, which uses the period representation for decimal points. ACIS makes a call to the function setlocale to change the environment to the C locale before writing a save file. The locale is reset to its original value after the file is written.

The option restore_locale enables applications to read pre-2.1 save files that were written with other locales. Before restoring the file, the option should be set to a string representing the locale in effect when the file was written. The option may be set with the function api_set_str_option, with the Scheme extension option:set. When a file is restored, ACIS makes a call to the function setlocale to set the environment to the value (string) specified by the option.

Option for Testing Shared Geometry

Topic: *SAT Save and Restore

The test_share option checks for shared geometry when restoring SAT files. As entities are read into ACIS, int_cur and spl_sur types are compared with those that have already been restored to determine if they are identical to a previously restored int_cur or spl_sur. They can then be restored simply by incrementing a use count instead of restoring the entire object. This option significantly reduces the size of retrieved bodies and aids subsequent operations, but it can be expensive and can become noticeable when restoring large parts.

The option can be turned off to speed up the restore process. However, the amount of memory required to restore a model will be larger, since sharing of geometry is not taking place. Also, evaluations of geometry during modeling operations may take longer because the test for coincidence will take place each time the objects are evaluated. This may cause the test to happen many times instead of once when the model is loaded.

If you are confident that your models contain little shared data or that the data has already been shared via modeling operations in ACIS, then turning the test_share option off may lead to faster load times.