Chapter 7.
History and Roll Overview

Topic: *History and Roll

When ACIS entities and APIs are used, history and roll back are supported. The history and roll functionality tracks all changes that occur in an ACIS session, which permits rapid change between states of an ACIS model. The *history manager* allows both branched histories and linear histories, which provides even more flexibility for rolling between states.

ACIS history and roll uses four levels:

Bulletin . . . . . Holds the changes to a single ACIS entity. Individual bulletins are not used outside the context of a bulletin board or delta state.

Bulletin board . . . . . Holds all bulletins resulting from an API call.

Delta state . . . . . The delta state is a user-definable, change-tracking structure of arbitrary size. It holds all bulletin boards resulting from a modification to a model. The delta state also has pointers to the previous, next, and partner delta states. The partner is a branch delta state at the same "sublevel."

History stream . . . . . Contains pointers to the initial delta state, to the last noted active delta state, and to the delta state under construction. The history manager handles the history streams.

History streams for a given model can be saved with the model in its save file. Later, this entire structure can be restored, permitting roll back to any point in development. Saving and restoring history streams is covered in more detail in *3D ACIS SAT Format*.

The application developer uses the bulletin board to monitor changes to models throughout the modeling process. This is particularly effective for incrementally updating application models, such as finite element mesh or graphical representation maintained for display purposes.

Bulletins

Topic: *History and Roll

The bulletin is the fundamental structure in the history stream. When an entity is created, deleted, or changed, a bulletin recording the action is added to the current bulletin board. A bulletin contains a pointer to the old entity and a pointer to the new entity. Bulletins contain sufficient data structure information to supply precise information about the change to an application program.

When an operation performs major changes to a model, each entity in the model that is changed has its own bulletin. These bulletins become part of a linked list whose head is the bulletin board. Bulletin boards are part of another linked list, called the delta state. Roll back and roll forward work on delta states, which use bulletin boards and the entity level bulletins.

Bulletins are generated for data entities in ACIS, and for extensions made by application developers. On a lower level, support for roll back can be achieved using ENTITY class methods.

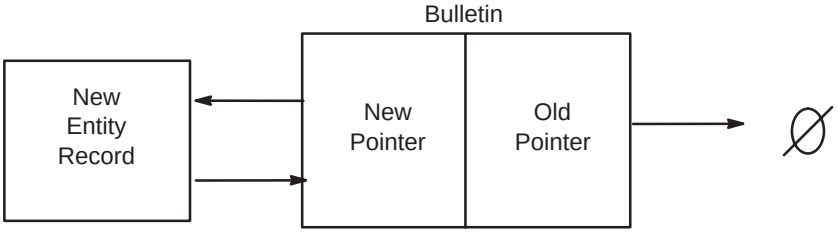

Figure 7-1. Create New Entity

When ACIS changes a model entity, it checks to see if the record is logged. If the record is not logged, ACIS copies the entity to a new record obtained from free store and changes the original record to reflect the change. ACIS then generates a bulletin that holds pointers to both the copied and changed versions of the record (Figure 7-2).

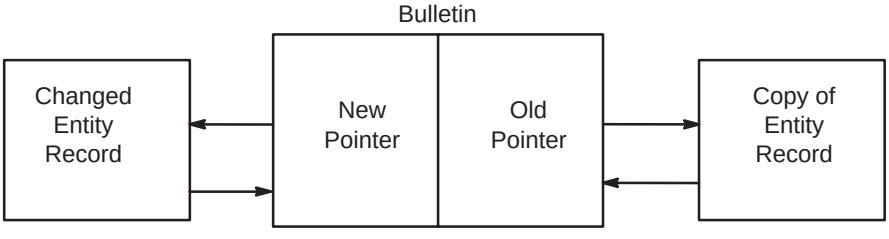

Figure 7-2. Change Entity

When ACIS deletes an entity, it generates a bulletin that holds a pointer to the old version of the record (Figure 7-3). Entities are not returned to free store, and the space is not available for reuse. To return the record to free store, delete the bulletin board and its bulletins using api_delete_ds or api_prune_history.

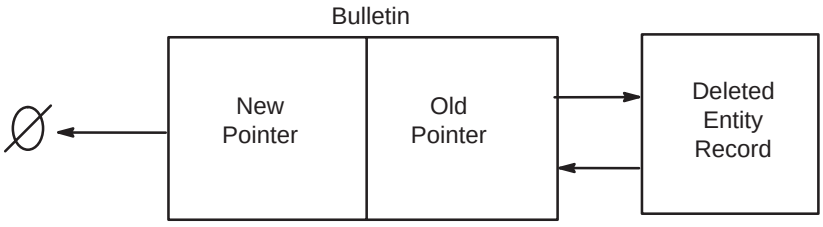

Figure 7-3. Delete Entity

Bulletin Boards

Topic: *History and Roll

When an entity in the ACIS model is created, altered, or deleted, a bulletin recording the data structure change is added to a bulletin board. In the modeler's default state (in which logging and auto-checkpointing are on), each call to an API function generates a new bulletin board that holds a list of bulletins for recording modifications to the model. The bulletin board holds pointers to the first and last bulletin, and to the next bulletin board (Figure 7-4).

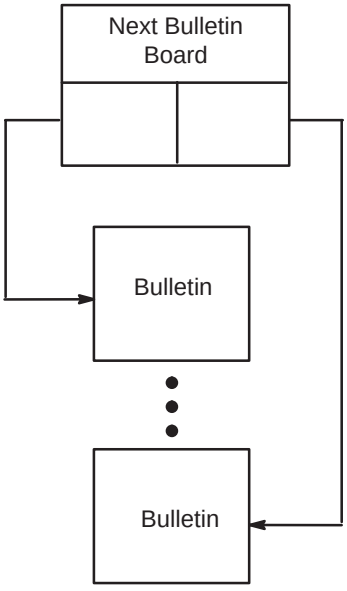

Figure 7-4. Bulletin Board Structure

The BULLETIN class provides functions that read the status of the current bulletin board: retrieve the first, last, next, and previous bulletin on a particular bulletin board; retrieve the old and new entity pointer from a bulletin board; and retrieve the type of bulletin.

Note *An application can read, but cannot alter, the records in a bulletin.*

The ACIS programmer has control over roll back through APIs (api_note_state, api_change_state, api_change_to_state, api_find_named_state, api_name_state, api_roll_n_states, and api_delete_ds) and macros (API_BEGIN, API_END, API_NOP_BEGIN, API_NOP_END, API_TRIAL_BEGIN, and API_TRIAL_END). ACIS is shipped with the default of api_logging TRUE. Therefore, the system automatically logs all operations. This is the primary control of whether the application will support undo/redo. When api_logging is set to FALSE, ACIS discards all but the current delta state.

Bulletin Board APIs

Topic: *History and Roll

API functions start by calling api_bb_begin and end by calling api_bb_end. The API api_bb_begin turns auto–checkpointing off; api_bb_end turns auto–checkpointing on.

When auto-checkpointing is off, the system adds bulletins to the current bulletin board rather than creating new bulletin boards. Because auto-checkpointing is turned off at the beginning of each API function, nested API functions do not create new bulletin boards.

Auto-checkpointing is an accumulating option. For example, if auto-checkpointing is on, then turned off three times by calls of api_bb_begin, it is not turned on until three calls have been made to api_bb_end.

If an API call fails, the current bulletin board is preserved to make it available for debugging. At the next call to an API function, the model is rolled back to the state before the API call failed.

Bulletin Board Macros

Topic: *History and Roll

There are three sets of macros for BULLETIN_BOARD creation, which differ in how they handle error conditions and stacked bulletin boards.

API_BEGIN / API_END are the basic macros used to wrap all modifications to the model. The auto–checkpointing counter is incremented in API_BEGIN and decremented in API_END. Be careful not to jump out of an API_BEGIN / API_END block using return or goto, because it gets the counter out of synchronization. Model changes not wrapped in these macros cause a sys_error(NO_CUR_BB) to be issued.

API_BEGIN opens a new bulletin board without any bulletins. However, API_END does not close that bulletin board. The bulletin board is not closed until the next API_BEGIN. This delay in closing the bulletin board is to handle error conditions that might have occurred while changing the model. A bulletin board is also closed after a note state operation (api_note_state). The option bb_immediate_close can be used find errors that result when a delta state and bulletin boards are logically closed.

For history and roll to function properly, functions that change the model must be enclosed within the two macros, API_BEGIN and API_END. To create one bulletin board for multiple API calls, nest them within these macros. In such nested cases, new bulletin boards are not created; instead, entity changes are attached to the bulletins within the current bulletin board. When using ACIS APIs to create modeling elements, each API generally opens a new bulletin board because it has API_BEGIN and API_END as part of its code.

If an error occurs, control jumps to the API_END. If an error occurs in a nested API_BEGIN / API_END block, control jumps to the API_END and proceeds from there. If the API_END was at the outermost level of nesting, the counter goes to zero and the BULLETIN_BOARD is marked as failed, then closed. The model will be invalid.

At this point one may look at the BULLETIN_BOARD and the (now invalid) model for feedback about the error. When the next BULLETIN_BOARD is opened, the changes in the failed BULLETIN_BOARD will be rolled back and discarded. It is up to the programmer to check the result and resignal the error (using sys_error), to avoid problems with an invalid model in later operations. If the error is not resignaled to the outermost API_END, the BULLETIN_BOARD will not be marked as failed and the changes will not be rolled back.

API_NOP_BEGIN / API_NOP_END are used for operations that change the model, but where the user does not want the model changed or any BULLETINs to be created. Typically they are used in query operations, including ray testing and evaluation that may add boxes to the model but make no user perceptible change.

API_NOP_BEGIN causes a new BULLETIN_BOARD to be created and stacked, regardless of the nesting level counter. API_NOP_END rolls back and discards the changes in the stacked BULLETIN_BOARD.

API_TRIAL_BEGIN / API_TRIAL_END can be used to test operations that may or may not work. For example, a user interface may allow a user to preview creation of segments of a sketch, but have the option to abort them. By wrapping the changes in API_TRIAL_BEGIN / API_TRIAL_END, one can cause the abort to occur by setting an error code in the outcome, or keep the changes by not setting an error. These macros are also useful for optimization. One may first try a quick algorithm that does not always work and roll it back and use a more robust but slower algorithm if it fails. Simply wrap the quick algorithm in API_TRIAL_BEGIN / API_TRIAL_END and check the outcome.

API_TRIAL_BEGIN stacks a new BULLETIN_BOARD, just as API_NOP_BEGIN does. In API_TRIAL_END, the result is checked. If the operation was successful, the resulting bulletins are merged with the BULLETIN_BOARD it is stacked on top of. If the operation failed, the changes are rolled back and discarded.

Bulletin Board Compression

Topic: *History and Roll

If an entity has been modified in several different API_BEGIN / API_END blocks, there will be several bulletins for that entity, each on a different bulletin board.

When the option compress_bb is on (default), at the end of each successful block the bulletins in the bulletin board created for that block are merged with those from the previous bulletin board, so they appear as though the operations occurred in the same block. This should save memory used by extra bulletins and backup copies of modified entities. It should also save time during roll back.

In some cases performance can be improved by setting the compress_bb option off, since the time to compress the bulletin boards can be prohibitive in cases in which a single delta state is used for many API calls.

Delta States

Topic: *History and Roll

One or more bulletin boards are grouped to allow the user to move back and forth through modeler changes in larger moves than from bulletin board to bulletin board.

When the application calls api_note_state, all bulletin boards made since the previous call to api_note_state (or since logging was first turned on) are returned in a delta state as shown in Figure 7-5.

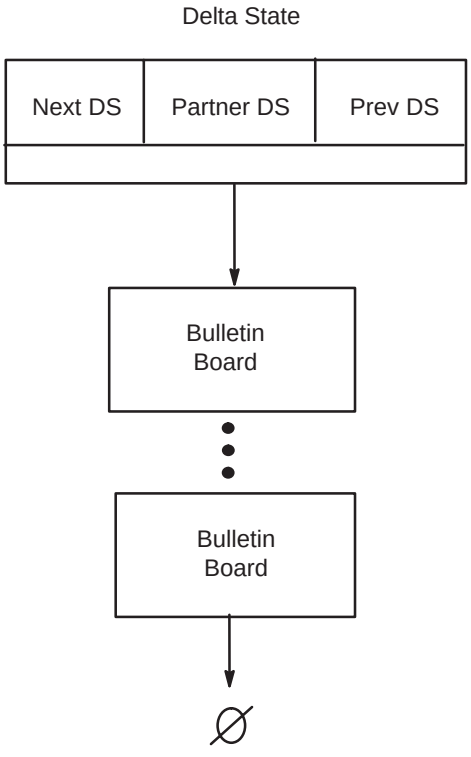

Figure 7-5. Delta State Structure

The *current* delta state is the delta state that is being built. This delta state is open. When api_note_state is called, the current delta state is closed and becomes the *active* delta state.

The active delta state is the most recently closed delta state made by calling api_note_state. Also, rolling to a particular delta state makes that state the active delta state. This state is active in the sense that it represents the state of the model directly after roll or calling api_note_state (before construction of a new delta state begins).

The delta state contains a pointer to the first in a list of singly-linked bulletin boards. Each bulletin board contains a pointer to the next bulletin board in the chain only.

The delta state holds two state identifiers that refer to the current (Prev DS) state and to the next (Next DS) state. State identifiers are internal modeler state names that are created when a state is noted. Each modeler state is expressed as a unique integer.

To permit the user to move between modeler states, the application must remember the delta states returned to it.

Note *The "next" state pointer in the model is with respect to* roll back*, as in "the state obtained when rolling the model* back *one step." Likewise, the "previous" state references "the last state obtained before rolling the model* back.*"*

Rolling to Delta States

Topic: *History and Roll

Moving to a state means "make the model data structure the same as it was when the state was noted." To move to any state, call api_change_to_state with the desired state as an argument. To move to modeler state 1, you can call api_change_state with delta state 1 as an argument. api_change_state is a read/write function that changes the model; it does not create bulletins or a bulletin board because they already exist in the delta state.

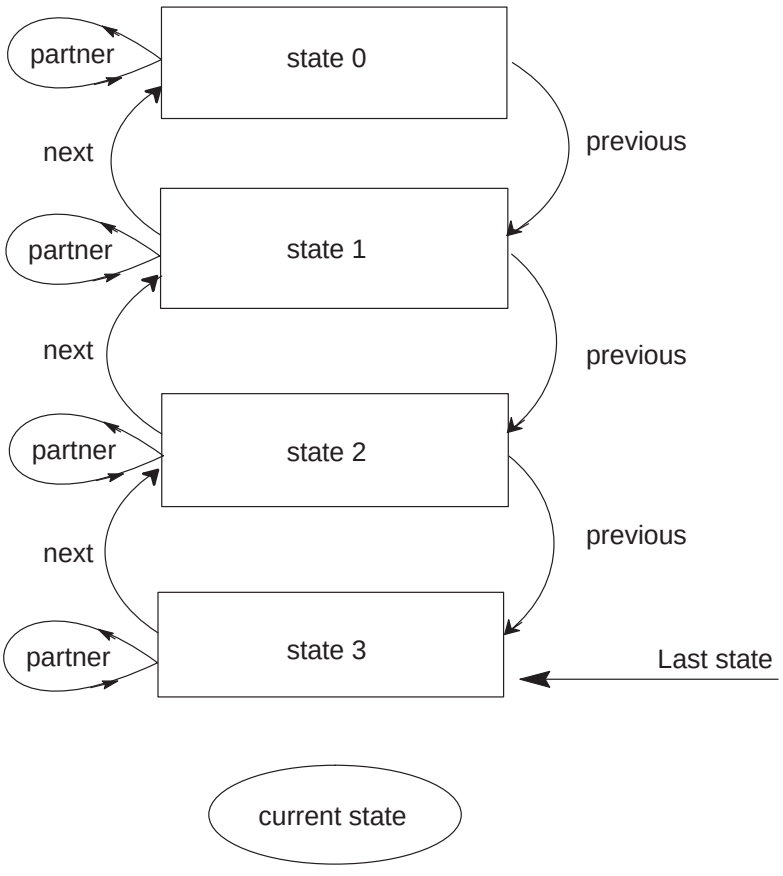

Figure 7-6. Rolling to Delta States

In a nonlinear or branching case, a delta state can have an additional *partner delta state* pointer. The partner pointer is part of circular linked list of states branching from the same state. The partner pointer is always set; if there is no other branch, the pointer points to itself.

The only way to reliably get backward or forward to a given state is to give it a name and remember that name, or to remember its pointer. This is then used as the argument in the roll function.

Rolling the model backward presents no problems or additional overhead to the application developer. However, because branching is permitted in the delta states, roll forward could present an ambiguous situation. How does ACIS know which state is referenced when a roll forward is requested? It doesn't. If the state is important, either the pointer or the name of a given state has to be remembered (e.g., api_note_state).

The delete_if_empty argument in api_note_state allows the caller to remove a delta state that contains no bulletins. So if this argument is set TRUE, some code may need to be modified to account for deleted empty states, for example, code that relies on a particular number of states on a stream.

Use the partner delta state pointer to branch to the various delta states. The partner list is a circular linked list of delta states all having the same previous state. Traversing into the various branch delta states uses this circular partner list, not just the next delta state pointer. When a branch of the history is pruned, the partner delta state pointers are also updated.

When experimenting with roll backward *S*-steps and roll forward *S+n*-steps with a state having branches, you may experience unwanted side effects. Specifically, there may be a difference between what happens when rolling backward *to* a branch state and then "blindly rolling forward," and what happens when rolling backward *past* the branch state and then blindly rolling forward. It is the "blindly rolling forward" and the ambiguities in the branch state which have the potential for difficulties. The solution is to *not* blindly roll forward, but rather to note the delta states of importance beforehand and then to explicitly roll to those states.

Integer identifiers, unique to a given history stream, can be requested for each entity. These IDs remain constant through roll, save and restore (with APIs api_save_history and api_restore_history). 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.

Keeping the Part and Display in Sync with the Model

Topic: *History and Roll

Call the PART::update method when an entity has been updated. The part itself doesn't really care, but this triggers the entity_callback mechanism with a pm_Update_Entity so that observers know that something interesting has occurred. During a later rollback, the entity_callback mechanism is triggered with a pm_Roll_Update_Entity.

PART::update does change the model and therefore starts a new DELTA_STATE if one is not already open. The reason for the model change is to create a change bulletin on the ID_ATTRIB (and the DISPLAY_ATTRIB if using the GI tool). This change bulletin is noticed during rollback and is used keep the PART (and display) in sync with the active state of the model.

PART::update should be called after each change to the model. Call PART::update and then api_part_note_state inside the delta state where the changes occurred. There is no need to call PART::update after roll.

Deleting Delta States and Freeing Memory

Topic: *History and Roll, *Memory Management

When delta states are no longer required, delete them by calling to api_delete_ds or api_prune_history. This call releases their space to free store, making it available for use in subsequent modeling. The delta state, bulletin boards, and bulletins are returned to free store, and entity records referred to in the bulletins are also released.

A DELTA_STATE will be automatically deleted if it is empty and was noted by api_part_note_state or api_pm_note_state. The criterion for empty is no BULLETINs. There may be one or more BULLETIN_BOARDs, but if none have any BULLETINs, then the DELTA_STATE is empty.

Other API calls are available to prune preceding states, following states, and all states, depending upon the roll back or roll forward information needed.

api_delete_ds is a procedural version of the DELTA_STATE class destructor. It releases memory associated with the DELTA_STATE including some BULLETIN_BOARDs, BULLETINs and backup copies of various ENTITYs.

So to release memory used by rollback, call api_delete_ds. To minimize the memory used by rollback, call api_logging(FALSE), which causes the system to automatically delete the DELTA_STATEs as often as possible, while still retaining rollback's role in error handling.

api_stop_modeler properly shuts down the modeler and frees all the memory used by ACIS. api_stop_modeler relies on the fact that if no DELTA_STATEs have been deleted, then the rollback history contains pointers to all the live ENTITYs as well as the backups. The key phrase here is "if no DELTA_STATEs have been deleted."

api_prune_history also deletes delta states, but first saves the information needed by api_stop_modeller.

There are three ways to clean up on exit:

- Do not bother; let the operating system handle it.
- Call api_stop_modeler, and whatever else you need to clean up your application.
- Keep track of all DELTA_STATE and ENTITYs and then explicitly call api_delete_ds and api_del_entity and whatever else is needed to clean up the application.

api_stop_modeler can make up for lax programming, but only if no DELTA_STATEs have been deleted. This includes explicit deletes, deletes through api_delete_ds, and implicit deletes due to api_logging(FALSE).

The developer concerned with memory should use one of these methods:

- Use api_prune_history(ds–>history_stream(), ds) after each call to api_note_state(ds). This uses a bit more memory, but retains api_stop_modeler's ability to clean up after some sloppy programming.
- Use hs–>set_max_states_to_keep(number) to provide limited rollback.
- Keep track of all DELTA_STATE and ENTITYs and then explicitly call api_delete_ds and api_del_entity and whatever else is needed to clean up the application.

Any of these can be used with the compress#_bb option, which merges multiple BULLETIN_BOARDs into one (on by default). This means that each delta state will have a single bulletin board, which saves a significant amount of memory, improves performance of roll and API_TRIAL_BEGIN/END blocks.

History Manager

Topic: *History and Roll

The history manager allows the user to roll directly to a named state even if it is across multiple branches. It is possible to cycle through all branches of the history within a single stream.

The history manager supports multiple history streams. As you create and work with different parts, the individual parts maintain their own independent history streams. As an example, when working in Scheme, parts are managed in independent windows.

The history stream maintains three pointers. The first pointer is to the root delta state. The second pointer is to the last noted active delta state. The third pointer is to the delta state currently being created.

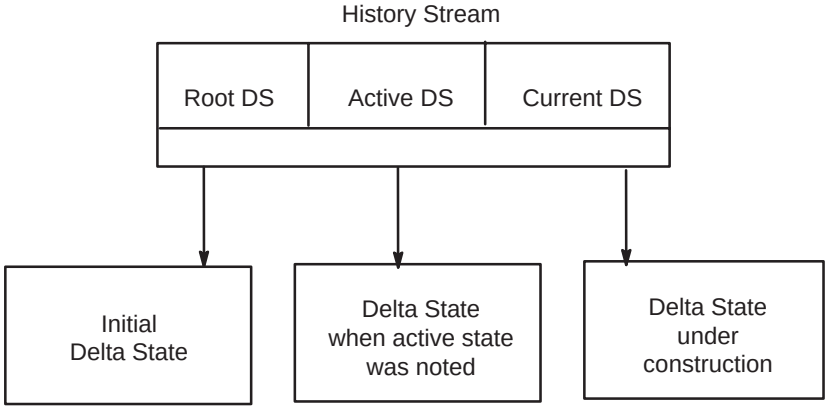

Figure 7-7. History Stream Structure

The following functions may be used with or without the Part Management Component.

Memory Management:

api_prune_history . . . . . Remove delta states in a history stream

Save/Restore:

api_save_entity_list_with_history . . . . . Save history information with the part

api_save_entity_list_with_history_file . . . Save history information with the part

api_restore_entity_list_with_history . . . . . Restore the part and history

api_restore_entity_list_with_history_file . Restore the part and history

History Management Functions with Part Manager

Topic: *History and Roll

The calls to api_pm_start_state and api_pm_note_state (or api_part_start_state and api_part_note_state) must be strictly paired regardless of errors. Start state and note state are paired by the use of a static level counter. If the note state were skipped when there was an error, the counter would be off by one and subsequent states would not be noted.

```
int depth;
api_pm_start_state(depth);
API_BEGIN

result = api_do_stuff_1(args);
check_outcome(result);     // If result is not ok,
                           // jump to API_END

// Alternate style of using check_outcome
check_outcome(api_do_stuff_2(args));

// Tell the part manager and graphics what happened
record_entity(new top level entity);
update_entity(modified top level entity);

API_END
api_pm_note_state(outcome(API_SUCCESS), depth);
```

If an error occurs, it will be caught by API_END. The api_pm_note_state is always called regardless of error. Note that the outcome is checked before recording or updating entities, so the part manager and graphics don’t see anything bad. The check level is controlled by the option history_checks.

You can also use API_SYS_BEGIN/END or EXCEPTION_BEGIN/TRY/CATCH/END with api_pm_start_state in the EXCEPTION_BEGIN block and api_pm_note_state in an EXCEPTION_CATCH(TRUE) block.

History Management Functions without Part Manager

Topic: *History and Roll

The Kernel has a set of APIs allowing advanced part management independent of the Part Management Component. For example, it is possible to use PART only as a port of entry to the Graphic Interaction Component. This permits roll back without using the Part Management Component.

Since Part Management uses the Kernel Component APIs, any use of the roll portions of the Part Management could interfere with tasks performed by the Kernel APIs. This includes indirect calls such as StartEntityCreation, EndEntityCreation, and the modification cousins. Searching for "api_pm_" should identify calls to avoid. Even innocuous APIs like api_pm_save_part and api_pm_load_part could cause problems, because they close delta states and could create new ones.

The following history-related APIs from the Kernel do not require anything from the Part Management Component.

The foundation of all history management:

api_note_state . . . . . Note delta states

api_delete_ds . . . . . Delete a delta state

api_change_state . . . . . Roll back and forward to a given state

api_bb_begin . . . . . Used in the API_BEGIN macro

api_bb_end . . . . . Used in the API_END macro

Advanced navigation in existing streams:

api_change_to_state . . . . . Roll to a given delta state

api_find_named_state . . . . . Recall a delta state with a given name

api_name_state . . . . . Name a delta state

api_roll_n_states . . . . . Roll back and forward

Support for multiple history streams:

The api_add_state and api_remove_state are relatively easy to use. After noting a state, remove it from the default history stream and add it to the stream it should belong to. api_distribute_state_to_streams is much more difficult to use and incorporates the StreamFinder class.

api_add_state . . . . . Add states between streams

api_remove_state . . . . . Move states between streams

api_distribute_state_to_streams . . . . . After noting a state, distribute its bulletins to all the relevant history streams

Disabling History

Topic: *History and Roll

History is not explicitly enabled or disabled. It is effectively enabled simply by making calls to functions to perform required tasks. ACIS provides two levels of disabling history.

With api_logging(FALSE), bulletins and delta states are still created, but kept only long enough to support error recovery.

With set_logging(FALSE), no bulletins are created, but bulletin boards and delta states are still created. Modeling errors can not be recovered unless the application developers provide their own mechanism.

The developer must be careful about the use of API_BEGIN/API_END, and should still call api_note_state periodically to clean up memory used by empty bulletin boards.

Using History and Roll in Scheme

Topic: *History and Roll

History and roll functionality is supported by Scheme AIDE, which can be used to rapidly test changes between states of an ACIS model. The default settings for history and roll in Scheme AIDE can be obtained through the option:list extension.

A decision about the history management type has to be made before modeling begins when a new part is created. With the exception of logging, none of the history stream options are designed to be changed during actual model creation.

Options for History Management

Topic: *History and Roll

The options that are important for history streams are:

history_checks . . . . . When TRUE (default), additional checks are done on the integrity of the history stream data structure. When FALSE, no checks are done.

logging . . . . . When TRUE (default), bulletin boards and delta states are visible at the application level (it logs a change to the history stream). The bulletin boards are not deleted when the next one is open; they remain available. When FALSE, each bulletin board is deleted as soon as the next is opened (logging to the history stream is turned off).
This is the one option:set parameter that can be changed after a model has been created.

Scheme Extensions for History Management

Topic: *History and Roll

Use the following Scheme extensions to manage multiple history streams when working with multiple parts.

Part histories contain roll back information for all entities in a part. Obtain this history with (history part).

history . . . . . Gets a history stream from a part or ID or string

roll:delete–all–states . . . . . Deletes all delta states

The *default history* contains history for entities without history and in a part without part history. When distributed history is off (FALSE), the default history contains all history. When distributed history is on (TRUE), the default history contains history for entities that could not be found in a part. Obtain the default history with (history “default”).

Start with a part:new

Topic: *History and Roll

The first step in establishing a history stream is to create a new part, using the part:new extension.

The part:new extension takes an optional argument, size, which must be a prime number. It specifies how large to make the table that stores the entities in the part. This number does not limit the maximum number of entities that can be stored. However, if a part has many more entities than this number, performance slows whenever your application has to look up the entities in Scheme. If the number is much larger than the number of entities referenced from Scheme, then more memory is used than is actually needed. The default size is DEFAULT_PART_SIZE, which is defined in the file pmhusk/part_utl.hxx (value is 1009).

```
; part:new
; Define a new part
(define my_part (part:new))
;; my_part
; Define a new view for the part
(define my_view (view:dl my_part))
;; my_view
```

Turn on History and Roll

Topic: *History and Roll

When the default settings are used:

- History stream is turned on
- The history stream is linear
- The history stream is for the part and is independent of other part history streams
- The history streams can be saved and restored with the part

Once the new part has been created, some of its history stream settings can be changed using the option:set extension.

For the settings to become active, the options must be changed in a new part before any model geometry has been created.

Use Distributed History Streams

Topic: *History and Roll

Distributed history enables part history. After distribution is set (either on or off) it cannot be changed. Distributed history is off by default, and the first bulletin created in for a part's history stream locks it this state.

```
; part:new
; Define a new part
(define my_part (part:new))
;; my_part
; Define a new view for the part
(define my_view (view:dl my_part))
;; my_view
(env:set-active-part my_part)
;; ()
(part:set-distribution-mode #t)
;; #t
```

To use distributed history after it is enabled, create model geometry. At some point in the modeling, name it using roll:name–state. From that state, you can roll to some previous state using roll. Any new modeling operations performed from this state on are created in a new branch.

```
; roll:name–state
; Define a new part
(define my_part (part:new))
;; my_part
; Define a new view for the part
(define my_view (view:dl my_part))
;; my_view
(env:set–active–part my_part)
;; ()
(part:set–distribution–mode #t)
;; #t
(define my_sph1 (solid:sphere (position 0 0 0) 10))
;; my_sph1 => #[entity 1 1]
(define my_sph2 (solid:sphere (position 5 10 20) 10))
;; my_sph2 => #[entity 2 1]

; Name the current state.
(roll:name–state ”first_branch”)
;; ”first_branch”
; roll back to before this creation
(roll ”start”)
;; 2

; Any modeling created from now on goes into a
; new history stream.
(define my_block (solid:block
    (position –5 10 –20) (position 5 –10 20)))
;; my_block => #[entity 3 1]
; Name the current state.
(roll:name–state ”second_branch”)
;; ”second_branch”
(roll ”first_branch”)
;; 3
(roll ”second_branch”)
;; 3
```

Roll Through States

Topic: *History and Roll

Once states and branches have been defined, they are easily accessed through the roll extension. Several operations can be grouped together as one operation to roll back using the roll:mark–start and roll:mark–end extensions.

roll ..... Rolls to a previous or later state

roll:mark–end ..... Marks the end of a block of functions for rolling

roll:mark–start ..... Marks the start of a block of functions for rolling

roll:name–state ..... Attaches a name to the current state for rolling

The easiest way of accessing branched states is to name their states and then start roll:name–state with the respectively named states. However, a branch does not have to be named to be accessible.

All of the available branched states can be reached using the roll extension. When using roll, it is important to roll back past the state where the branching begins. Otherwise, roll forward and backward remain on the same branch.

```
; roll:name–state
; Define a new part
(define my_part (part:new))
;; my_part => #[part 2]

; Define a new view for the part
(define my_view (view:dl my_part))
;; my_view
(env:set–active–part my_part)
;; ()
(part:set–distribution–mode #t)
;; #t

; branching point
(define my_sph1 (solid:sphere (position 0 0 0) 10))
;; my_sph1 => #[entity 1 2]

; one branch
(define my_sph2 (solid:sphere (position 5 10 20) 10))
;; my_sph2 => #[entity 2 2]
; roll back to before this creation
(roll –1)
;; –1

; Any modeling created from now on goes into a
; new history stream.

; second branch
(define my_block (solid:block
    (position –5 10 –20) (position 5 –10 20)))
;; my_block => #[entity 3 2]

; roll back to before this creation
(roll –1)
;; –1
```

```
; third branch
(define my_block2 (solid:block
    (position 10 20 30) (position 15 30 45)))
;; my_block2 => #[entity 4 2]

; Go to a point before the branching
(roll ”start”)
;; 2

; take one branch
(roll ”end”)
;; 2

; Go to a point before the branching
(roll ”start”)
;; 2

; take second branch
(roll ”end”)
;; 2

; Go to a point before the branching
(roll ”start”)
;; 2
; take third branch
(roll ”end”)
;; 2
```

Save History Streams

Topic: *History and Roll

History streams are saved with the part when the part:save extension is used. The default for part:save is not to save the history stream.

part:save	Saves all entities in a part to a file
part:save–selection	Saves a list of entities to a file; can not save history
part:load	Loads a part from a file into an active part

If the history stream is no longer required, it does not have to be saved to a save file. This is an option on part:save. part:save has an option for the filename to specify both the path and filename. It has additional options for saving the part in a binary or text file, and for saving the history data.

If textmode is #t, the data is saved in text mode. If textmode is not specified, then the mode is determined by the extension of the filename. If the filename string ends in .sab (or .SAB), then the file is saved in binary mode; otherwise, the file is saved in text mode.

After setting textmode, a second Boolean specifies whether to save roll back history data. The default, #f, does not save history.

The “history save” APIs and Scheme extension support saving only the main history stream, without branches. This is done with a “mainline only” argument. If the argument is true, then rolled branches are not saved to the file.

```
; part:save
; Define a new part
(define my_part (part:new))
;; my_part
(part:set-distribution-mode #t)
;; #t
(env:set-active-part my_part)
;; ()
; ...

; Create model here
; ...

; Save the currently-active part to the named file.
; This saves the part defined earlier in text mode
; and saves history data.
(part:save "test" #t my_part #t)
;; #t
```

The saved file can be restored at any time using part:load. Obviously, a save file that is loaded that did not contain history streams cannot roll back to previous states.

```
; part:save
; Define a new part
(define my_part (part:new))
;; my_part
(part:set-distribution-mode #t)
;; #t
(env:set-active-part my_part)
;; ()
; ...
; Create model here
(define my_block (solid:block
    (position 0 0 0) (position 10 10 10)))
;; my_block => #[entity 1 1]
; ...
; Save the currently active part to the named file.
; This saves the part defined earlier in text mode
; and saves history data.
(part:save-selection my_block "test" #t #t)
;; #t
```

Pruning Branches and States

Topic: *History and Roll

It is often useful to remove unnecessary branches and states from a given history stream. The following extensions for maintaining states within history streams have various options for exactly what gets removed and from which history stream (if not the default).

roll:delete–all–states	Deletes all delta states
roll:delete–following–states	Deletes all delta states after the current delta state
roll:delete–inactive–states	Deletes all delta states not in the active path from the root to the current one
roll:delete–previous–states	Deletes all delta states before the current one