Apollo

History specification

Confidential

# Disclaimer

This specification is not, by any stretch of the imagination, complete. It will need to be revised several times before it is complete. Currently several major parts are either missing or incomplete. This disclaimer will be updated to reflect any change in these sections. Finally a specification is supposed to be a ‘living’ document and therefore never complete.

# Introduction

The goal of the history system is to allow system to keep track of changes in the application so that these changes can be reverted either to the start state or to a state between the current state and the start state. Reverting the changes may be done on the users command (undo and redo) or when an undesirable state is detected. Examples of an undesirable state could be:

* Overwriting existing data with the data from a corrupted file.
* Having an iterative calculation diverge or become numerically unstable.

In normal cases state changes are only available as long as the application is running. The state history is discarded upon termination of the application. For the dataset applications it may be desirable that a change set survives an application restart. For this case it will be possible to take a snapshot of the current state through the history system. This snap shot will be written to permanent storage and can later be reloaded by a dataset application to continue after an application crash or as a cloned instance of the original dataset.

The ability to create a snapshot of the state of a dataset requires that each dataset application tracks its own history independent from the history of the main application. The history of the main application is linked against the histories of the loaded dataset applications. If the history of the main application is reverted (either partially or completely) then the linked histories of the dataset applications may (but not necessarily are) also reverted.

In order to roll-back history each of the history steps will be reverted in the opposite order of creation. Unlike a Version Control System (VCS) it is not possible to revert a step if this step is followed logically by one or more other steps. It is plausible that two or more steps are related and need to be played back or played forward together. For this situation the history system allows changes to be grouped into a single change set. All the changes in a change set are always reverted together. Note that reverting either a single change or a change set is not necessarily an atomic action.

Another limitation of the history system is that only certain actions are tracked. External actions which cannot logically be undone will not be tracked. Examples of actions that will not be tracked are:

* Any action of the file system, including but not limited to the plug-in assemblies. Removal or overwriting of a file (e.g. a data file or a plug-in assembly) in the file system will not be tracked in the time line.

As an even stronger limitation it is possible that external changes influence the ability to play back or forward along the current time line. For instance, if a plug-in assembly is removed then it may not be possible to play forward to a state where that plug-in assembly needs to be loaded. In this case the user will be notified that the time line is broken and the time line itself is adapted to the new state.

# Performance

The performance of the history system is crucial for the usability of the application. For instance the user should not notice any difference while using the application normally if the history system is actively tracking changes versus when it is not tracking changes. Since Apollo is aimed at numerical calculations it is also very important that the history system uses a minimal amount of CPU time and memory. The performance timings are set to:

* Get the current value inside a component that is being tracked by the history system: 1 – 1.5 x (the normal time for setting a member variable)
* Committing a change set: ??
* Revert to the last saved change set: ??
* Revert to a random earlier change set: ??

The memory performance demands are:

* Storage while not having any change sets:
* Storage while having a previous set: no more than 1 times the space necessary to store the changes versus the previous set.
* Storage overhead: minimal???

In all of this we assume that the nominal case is described by:

* History system actively tracking changes.
* User is only reverts small actions, but can do so on a regular basis. Slightly more irregular is the reverting of larger changes (e.g. undo / redo changes to a mesh).
* During simulation runs once in a while a big revert is required. While the simulation is running we want to see very little to no delay due to the change tracking. A big revert is allowed to take a measurable amount of time if there are a large number of change sets.

# Architecture

The history of a component is stored as a differential storage method, i.e. only the differences between revisions are stored. For example for an object with the following definition:

public sealed class FooBar

{

private int m\_Integer;

private double m\_Double;

....

}

If a change is made to m\_Integer then only the difference between the previous value and the current value will be stored. Because there was no change in m\_Double no change value for that member variable will be stored.

The use of differential storage does mean that objects no longer store the values of their member variables themselves. All storage has to be done by the history system so that it is possible to store the differential state at any point in time, without the owning objects having to deal with the complexities of this kind of storage. Another benefit of taking the storage away from the owning objects is that it will be possible to make changes to the storage method without having to change the implementation of every single class that is used in the application.

One set of problems with the use of the history system lies in the storage of object and collection references.

* Object references can become invalid (e.g. the object is deleted, changed or replaced before or after a roll back or roll forward action) which means that no object should directly store an object reference to another object. Instead of obtaining the reference directly an object reference should always be obtained indirectly through the use of an object ID.
* In order to handle collections the history system explicitly needs to be aware of the different types of collections so that the additions, removals and insertions can be tracked.

To reduce storage demands even further the changes to a variable are not normally stored unless the user indicates that a new history point should be marked. This method ensures that the history system consumes resources only when the user requires a history to be stored. When the user marks a new history point the history system determines if there has been a change to all the variables and it will store the differences if they exist.

To mark a history point users will create an instance of the ChangeSet class which automatically marks a history point when it is disposed. The user can nest as many ChangeSet objects as required. Nested history points are only reachable as long as the owning ChangeSet hasn’t been disposed of. As soon as a ChangeSet is disposed it will merge all the changes back into the time line marker and it will remove all the timeline markers of nested ChangeSet objects.

If an operation has side effects, e.g. the operation calls out to a dataset over the network, then the user can provide the current ChangeSet with a function that allows the ChangeSet to notify the side effect recipients of any roll back and roll forward operations.

To restore the state of the application to a previous history marker the user will select the correct history marker that should be restored. The history system then restores the desired state by either going backwards from the current state, or by moving forwards from an earlier snapshot, depending on which method is perceived to be faster. All history markers that have become invalid (because they were later in time than the marker to which the timeline was reverted) will be marked as redo markers. These redo time markers will be kept until a new time marker is created for the current changes.

Besides keeping track of change sets the history system also allows the creation of a ‘snapshot’ of the current state. These snapshots can be, but are not always, written to a permanent storage medium. The history system will create a snapshot (which is potentially expensive) at strategic times so that rolling back or forward (in a redo) is relatively cheap, even if the actions being reverted or redone are expensive. Hence snapshots will be made both at regular intervals, to simplify roll backs over a considerable number of markers, and directly before, for easy roll back, or after, for redo purposes, expensive actions.

Besides use by the history system for roll back purposes it is also possible to use a snapshot to create a clone of the current dataset.

It is possible that one or more of the history markers cannot be processed during a roll back or roll forward action. For instance due to the unreliability of network connections it is feasible that the remote dataset becomes unavailable, either before, during or after history is reverted. This means that a situation can arise where some history markers cannot be processed due to missing dependencies.

In order to handle this situation the history system marks every changeset with its dependencies so that it is possible to check if the dependencies are available to roll back or roll forward the change set. If the dependencies for a changeset are unavailable then several options exist depending on the nature of the dependency. These options are:

* The dependency does not block roll back but does block roll forward. This means that the changeset is ignored upon roll back, but it will block roll forward until either the dependencies are available or until the user decides to terminate the changeset. An example of this is a dataset that has gone offline from the perspective of the main application. Any changes to a dataset do not (normally) affect the main application, hence roll-back is possible. Roll forward however cannot be done until the dataset is back online again.
* The dependency blocks roll back but does not block roll forward. This means that the changeset will not roll back until the dependency is available but it could be rolled forward if the dependency is not available. An example of this is a plug-in assembly that has gone missing after the plug-in components have been removed from a dataset. In this case rolling back to a state where the component needs to be available is not possible, however it would be possible to roll forward to a state where the components do not have to be available.
* A combination of roll back and roll forward blocking.
* Neither roll back or roll forward blocking.

Depending on the type of dependency it is possible to delay a roll back or roll forward command until the dependency is available again.

The history system only tracks objects which wish to have roll back / roll forward capabilities. All other objects are ignored by the history system. This leads to the problem that during a roll back / roll forward it is not possible to determine if a non-tracked object reference has changed. The solution for this problem is to recreate the object each time a roll back / roll forward takes place. Obviously this means that the non-tracked objects should either:

* Not store any state
* Only store state that can easily be rebuild when the object is created

Finally in some cases it may be useful for information to survive a roll back / roll forward. For instance when running a simulation it is possible that the simulation ‘blows up’ which could trigger a roll back to a history marker where the simulation is known to be stable. If however the roll back destroys all the information, including the knowledge that the simulation will fail, then the simulation bound to repeat the same process, after all the situation hasn’t changed from before.

Under normal circumstances no information will survive a roll back / roll forward because all object state is either rolled back / forward or destroyed. This means that there is a need for a way for information to ‘travel through time’.