A Client-Server Architecture for Rich Visual History Interfaces

Richard M. Salter
Human-Computer Interaction Laboratory, Institute for Advanced Computer Studies,
Department of Computer Science & Institute for Systems Research
University of Maryland, College Park, MD 20742
+1 301 405 4391
rms@cs.umd.edu

ABSTRACT
History-keeping has surfaced as a potentially valuable asset to educational and other software. Current research in *learning histories* considers the hypothesis that providing learners with a readable record of their actions may help them monitor their behavior and reflect on their progress. However, the scope of learning histories goes far beyond the means provided by an undo/redo or document-recall history system. In this paper we describe *Trails*, a component-based framework for constructing rich learning history modules based on the client/server model. Trails historians are loosely-coupled to their client applications and interact with them through a set of well-defined interfaces. Trail historians also provide ample means for history visualization and direct manipulation. The client-server architecture facilitates history extensions to existing applications, while the modular design promotes experimentation with different visualization metaphors.

Keywords
History, Component Architecture, Simulation, Java, Graphical User Interfaces, Software Engineering

INTRODUCTION
History-keeping has long been a part of many types of interactive systems. Document-preparation applications, (e.g. word processors, spread sheets, etc.) keep command histories to permit the undo and redo of edits. Navigational applications (e.g. Web browsers and Help programs) keep histories of visited documents for later recall. In their simplest form, histories permit user actions to be logged and recorded, making them accessible in various ways for recovery and backtracking [9, 23]. Recovery mechanisms themselves range widely in complexity, from simple stack-based undo/redo [1, 21, 24] to complex structures such as graphical trees [22].

With the advent of interactive systems and networking, rich history mechanisms have become essential for applications where navigation and orientation are critical. Various strategies have been developed for history dynamics in browser programs [2, 4, 19, 20] and for temporally-based history visualization [6, 8, 10, 14]. Such research has greatly enhanced the playing field for history systems that are still used primarily for recovery and recall. Recently, however, history-keeping has surfaced as a potentially valuable asset in the context of educational software. Current research in *learning histories* considers the hypothesis that providing learners with a readable record of their actions may help them monitor their behavior and reflect on their progress. Moreover, histories can facilitate active collaboration among dispersed learning communities [3, 11, 16].

The scope of learning histories goes far beyond the means provided by an undo/redo or document-recall system. First and foremost, learning histories must provide a strong visual representation of the application's activity, to facilitate review and understanding by learners and instructors. Learning histories must also be first-class objects, capable of being saved, sent by email to others, posted to websites, edited, extracted, combined, and searched. They should also be annotatable, and (ideally) facilitate macro-extraction [10]. The visual representation should support direct manipulation, wherever possible, to implement history functionality.

A history system is generally custom designed as an intrinsic part of a program's architecture. As a result, opportunities for code reuse among history systems have been minimal. Given the high level of functionality we expect from the learning historian, it is both impractical and wasteful not to consider a framework architecture that can achieve the goals over a wide range of applications. This paper describes one such architecture currently being developed, called *Trails*. The Trails architecture achieves many of the aforementioned goals for learning histories, and shows great promise for achieving the others. Most importantly, Trails is a framework for creating *attachable* historian modules: rather than being an intrinsic

feature, the Trails historian and its application take on a server/client relationship. Trails historians are loosely-coupled to their client applications and interact with them through a set of well-defined interfaces.

In the next section, we provide background and an overview of the Trails framework, including our design goals. This is followed by a detailed description of the architecture. Example applications using Trails are presented throughout. The paper ends with a prospectus on future development.

OVERVIEW OF TRAILS

Motivation

The Trails project is the outgrowth of recent efforts to implement a learning histories mechanism in the context of an application framework for constructing simulation-based learning environments. Modules developed with this system, called SimPLE (Simulated Processes in a Learning Environment, [17]), use dynamic simulations and visualizations to represent realistic time-dependent behavior. Our anticipated use for this initial history system was twofold: a) to provide the instructor with a demo-based tutorial composer [13] built on annotated histories; and b) to facilitate communication between instructor and student via recorded histories of student simulation runs. This initial implementation was designed with these goals in mind.

A historian was created to a) record the user actions applied to simulation controls at various points in model time; b) display the corresponding control state changes in a visual format that clearly showed all correlation between control state and simulation output; and c) implement a replay capability during which recorded state changes are applied to their corresponding controls at the appropriate points in model time. Since the simulation is deterministic, this is sufficient to faithfully reproduce initial simulation results. The historian also supports annotation of history records, and serialization of content (for disk storage, email, etc.). A complete description of this system is given elsewhere [16].

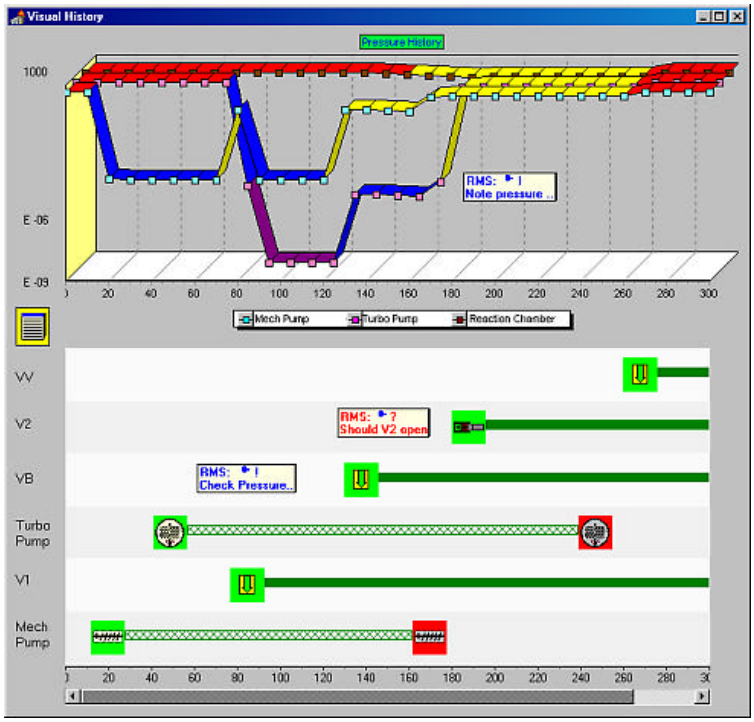


Figure 1. SimPLE Trails-based Visual History
Top display shows outputs corresponding to the control actions shown in the bottom display. Also visible in both displays are user annotations.

Figure 1 shows the visual history of a sample simulation run. In this simulation, only simple binary controls were required. Control action history over time appears in the bottom graph; in the top graph are corresponding output values. For each control, a square icon represents a state-changing action. A line connecting a pair of icons shows a period during which the control was in a positive state (i.e. *on* or *open*). An unterminated line indicates a control whose current state is *on*. The figure also shows user annotations. The user adds these after the simulation run, and they become part of the history record.

The history module is written as a separate unit extending the existing Delphi code of the SimPLE simulation application, with only minor changes to the existing code. In this setting, each individual simulation control communicates directly with a customized history recorder for that control called a *trail*. Each trail is primarily responsible for maintaining the list of time-stamped actions (called *trail items*) reported by its client simulation control. These data are used both to create an editable visual representation of the control's activities, and as a means of replaying the simulation.

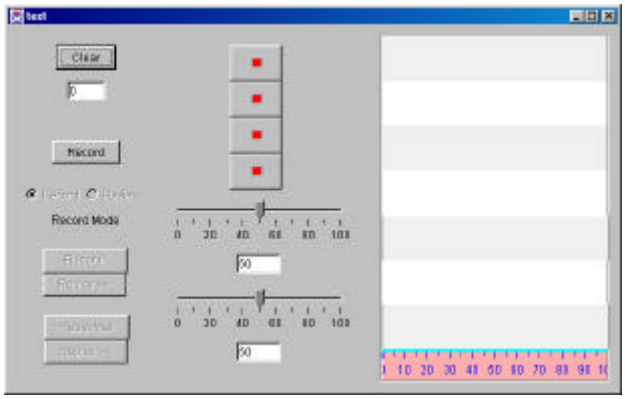
For this initial implementation, a rich set of Delphi components was created. These components enable drag-and-drop construction of a historian system for the SimPLE simulation application. Event dispatching code and call-back functions added to the simulation program complete the linkage between the historian and its client.

At the core of this construction is the relationship created between each individual client control (switch, slider, button, etc.) and its custom historian, or trail:

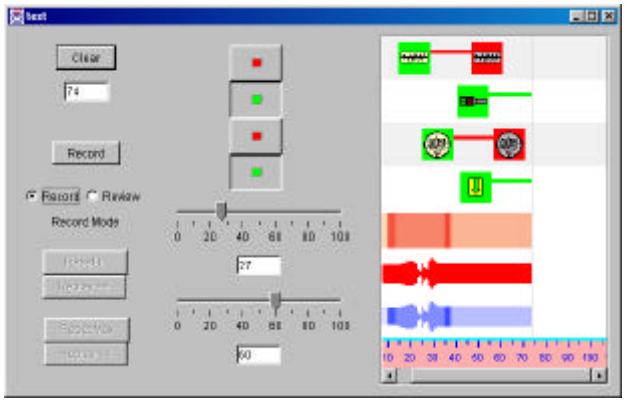
> *The* trail *is responsible for recording a trail item for each action reported by the client, making the list of trail items available for visualization and manipulation, and firing appropriate commands back to the client during session replay;*

> *The* client *is responsible for reporting all actions to the trail, and responding to trail commands during session replay;*

a.



b.



c.

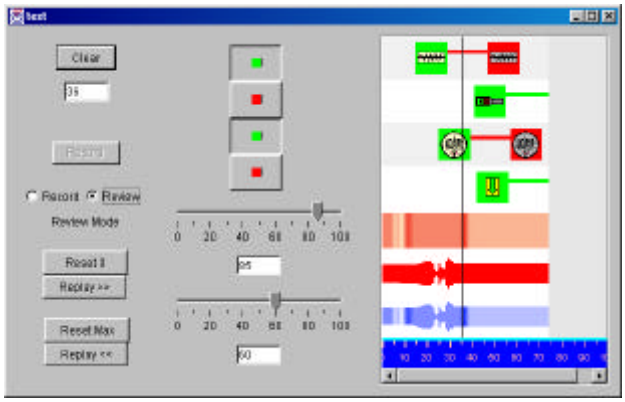


Figure 2. Example Trails Application
(Trail view panel is on the right)
a) Before simulation run
b) After 74 time units
c) Review at 36 time units

A subsequent translation of the original Delphi system to Java has permitted considerable refinement and expansion, and introduced interface definitions that make precise the properties that a potential client must have to be *trailable*. The Java model also includes an enhanced set of tools for history visualization (discussed further below). In addition, the Java model embeds the JFC/Swing *undo* package [5]. Any Swing control designed to use the undo package for managing edit undo/redo operations can exploit this capability within a Trails-based historian.

Historians built using the Java Trails package have been attached to several pilot Java applications, and, using a novel JNI-based Java-Delphi bridge [18], to a second SimPLE simulation. The description in the rest of the paper refers to the Java implementation. With the exception of the Java event model, the class structure is actually independent of any particular object-oriented language. Translation to any other object-oriented language should be straightforward once appropriate adjustments are made.

Example

Figure 2 demonstrates Trails at its current level of functionality. In the center of the panel are six controls: four toggle buttons and two sliders. For our purposes this control set represents the user interface for some simulation program. Each of these controls is attached to its own trail, which is an internal data structure that tracks that control's state. Each button can be in one of two states, either up or down, while the sliders can point to any value between 0 and 100.

The right panel contains a window with seven *trail views*. A trail view is not a trail, but rather is connected to one or more trails, and is responsible for rendering the data recorded by those trails. In our example, the four topmost trail views are each attached to the four respective button trails. Similarly, the next two trail views are each respectively attached to the two slider trails. The bottom-most view combines the data from the two sliders in a single display.

Note that each of the latter three views is presenting numerical values obtained from the sliders using a different metaphor. The first of these uses color saturation to create a spectrum-like visualization corresponding to different magnitudes, with light bands representing low magnitudes and dark bands representing high

magnitudes. The second view creates an envelope-like display, presenting a single color band that distinguishes magnitude by bandwidth. The bottom-most trail view shows the benefits of decoupling the trail from its visualization: this view is attached to both of the slider trails, pooling their data into a single display combining the spectrum and envelope views.

Figure 2a shows the (empty) history before any recording takes place. In Figure 2b, we see the state of the system after 74 time units have elapsed (each time unit is about 1/5 second). The pattern of user interaction with the buttons and sliders over the simulation run is clearly recorded in the history. Button history is visualized in the top four trail views using the same icon-based metaphor as in Figure 1. The next two trail views show, respectively, dark and light bands, corresponding to the first slider's movement, and a varying envelope corresponding to the second. These patterns are combined in the bottom-most trail view, showing the activity of both sliders.

In Figure 2c, the system has been switched from record to review mode. In this mode button activation times can be edited by dragging the icons left or right. The cursor can also be dragged using the mouse to any point in model time. As the cursor moves left, the historian will execute the necessary sequence of *retract* commands to undo state changes. Analogously, as the cursor moves right, a similar sequence of *replay* commands are executed to redo state changes. Following any cursor move, the system is returned to its state at the point indicated by the new cursor position. Thus, for example, to completely replay the sequence of actions, the system is reset to time 0, and the clock is used to step through each of the time points. The replay can run at a different speed than the original, either forward or backward. Any desired pattern or speed can be realized by simply dragging the cursor left and right with the mouse.

Of course, no non-trivial simulation system state is defined entirely in terms of its set of user controls, but several points are noteworthy. First, in any circumstance where state change is defined entirely in terms of "edits" (i.e. instantaneous incremental changes), this model is capable of capturing complete system history, and provides a rich user interface for movement in that history. In other applications, where user control state only partially defines system state, such as in deterministic simulation, we retain sufficient functionality to permit the replay of a given run. It will be the subject of future research to determine the extent to which the Trail model completely captures the state of various application domains.

Finally, we note that when the system returns to record mode following a history review, it will record from the end of the current history (i.e. as in Figure 2b). Except for the editing described above, history is currently available only for review and replay. We have deferred investigation into stack and tree-based history models to future work. However, it appears that the Trail client/server protocol is independent of any particular history recording model.

DESIGN GOALS

The Java-based Trails framework is designed to support the following goals:

- **Component-based framework.** All visual Trail components are derived from Java Swing superclasses. All Trail components comply with Javabean specifications. The Java Swing *undo* package is incorporated in the model.
- **Uniform functionality.** All Trail-based historians are expected to provide all the functionality required of a learning historian.
- **Lightweight cost.** Trail processing costs should be as lightweight as possible. This means, for example, that system state recording should not be redundant.
- **History model independence.** The Trail system is independent of any particular model of history recall, and is capable of implementing all.
- **Loose coupling with client.** The Trail system should be loosely coupled with the client application, using the underlying event architecture wherever possible.
- **Loose coupling with visualization.** A given trail record should be attachable to any visualization capable of displaying its contents.
- **Direct manipulation.** Wherever possible, trail functionality should be implemented through direct manipulation of the visualized history.
- **Hierarchical structuring.** Structuring tools for both trail recording and trail visualization should enable the creation of higher-order historians and history visualizations, respectively.

In summary, Trails is framework architecture, supplying a rich set of components for building history-keeping program modules. A historian can be custom-designed using Trails components according to the needs of some client application, to which it can then be easily linked. The historian supports a flexible and rich visualization that can be directly manipulated

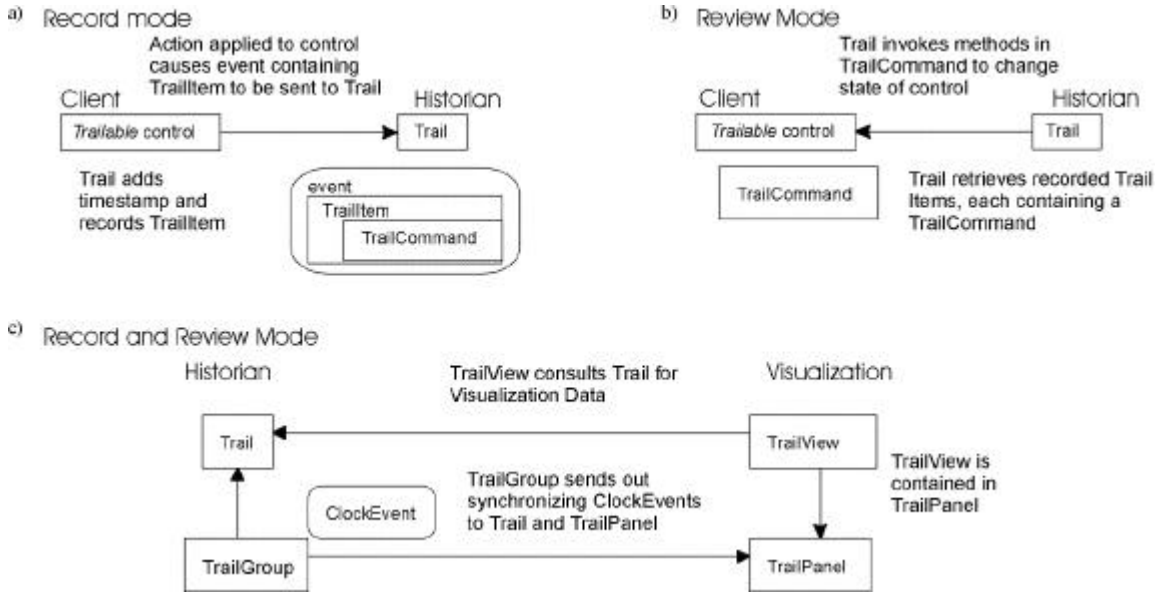


Figure 3. Trail System Data Flow
a) Recording control actions via event dispatch
b) Reviewing control actions using undo/redo commands
c) Visualization and synchronization

by the user to perform history functions (e.g. editing, undo/redo, etc.) Trail components can be added to the palettes of design tools such as Microsoft Visual J++ or Borland J-Builder to enable visual construction of historian modules.

As the project progresses, we expect to develop a recursive structure that will handle the problem of scale by managing history observation at different levels of granularity (e.g. history of the component verses the individual histories of its individual controls). We see great promise for this in the fact that a complete trail at one level can function as a trail item at another.

TRAIL ARCHITECTURE SPECIFICATION

In this section we outline the class structure and describe the behavior of the system.

Trail Class Structure

The Trails framework is an object-oriented system based on four categories of classes:

- **Client**

  The client classes are Trailable Java Swing controls (or other controls), extended to be capable of interacting with Trail objects.

- **Historian**

  These classes are responsible for recording events and making them available for review and display. The Trail object stores the history of the control to which it is attached. Each entry in that history is encapsulated in a TrailItem. The Trail maintains an enumeration of such entries.

  Additionally, each system has one or more TrailGroup objects to organize a set of Trails using a common clock. Each TrailGroup maintains its own clock, and is also responsible for managing disk storage/retrieval of the data contained in its constituent Trails, including any annotations.

- **Visualization**

  The TrailView class is the ancestor of all classes used to display control action data stored in a Trail.

  A TrailPanel object is a frame for displaying a set of TrailViews with a common clock.

- **Data and Communication**

  The TrailCommand interface specifies basic retract/replay functionality of client component actions.

  A TrailItem object points to its control of origin, and contains a TrailCommand and time-stamp.

There is also a set of event actions activated by dispatching a ClockEvent. These are used for system synchronization.

Communication Patterns

The three patterns of data flow in Trails are shown in Figure 3. Trails systems operate in two modes: *record mode*, during which client actions are reported to the historian, and *review mode*, during which the historian replays those actions on the client. In record mode, when a Trailable control wishes to transmit state-change information to its Trail, it dispatches an event containing a TrailItem that references the control and contains a TrailCommand for the state change. Any Trail registered to receive events from the control will consequently catch these TrailItems. When the event arrives at the Trail, the TrailItem is given a timestamp and added to the Trail's list of actions. This is pictured in Figure 3a.

During review mode, the actions earlier reported by a control and recorded by its Trail are fired back to that control. The Trail performs an undo or redo of an action using methods in the TrailCommand sent during record mode. This is illustrated in Figure 3b, and described in greater detail below.

The third locus of communication activity surrounds the event class ClockEvent. Each TrailGroup keeps a model clock, and fires ClockEvents in order to synchronize its member Trails, as well as the TrailPanels that hold their views. This occurs, for example, in review mode when the clock is changed through user manipulation of the cursor. It is in response to such events that a Trail will invoke TrailCommand methods to synchronize its client control to the new model time.

Client Controls

Several Trailable controls have been derived from Java Swing components such as JToggleButton and JSlider [5]. These controls form a separate package (HJComponent), and all follow the same elementary design pattern. Any standard Swing control can become Trailable through a straightforward extension.

Model Time

In a historian designed for tracking simulations, model time values are used to maintain the proper temporal separation between simulation actions. In such applications, an external source manages the TrailGroup clock during record mode, keeping it synchronized with the client application's model time.

In other applications, model time merely serves as a mechanism for totally ordering actions that occur at different controls and are recorded by different Trails. For example, a historian tracking the moves of a chess game might use a separate Trail for each player. A model clock for this application need only be incremented by one unit after each player's move in order to properly sequence the set of moves over both players. In the latter case the system runs in autoincrement mode, where following each recorded event the TrailGroup clock is incremented by one unit. No external clock source is required. By adopting such a broad concept of time, the Trails architecture accommodates both real-time simulation and non-simulation types of applications.

In all of our current implementations, only a single TrailGroup is required. A global model clock is therefore used for all actions. However, localizing the model clock to the TrailGroup class leaves open the possibility of designing historians for applications containing concurrent processes, using Lamport's clock algorithm for synchronization [12]. This is a subject for future research.

TrailCommands

The class TrailCommand encapsulates the undo and redo actions that manipulate the state of a control. A TrailCommand contains methods retract and replay, which are applied to the control in order to undo or redo the action, respectively.

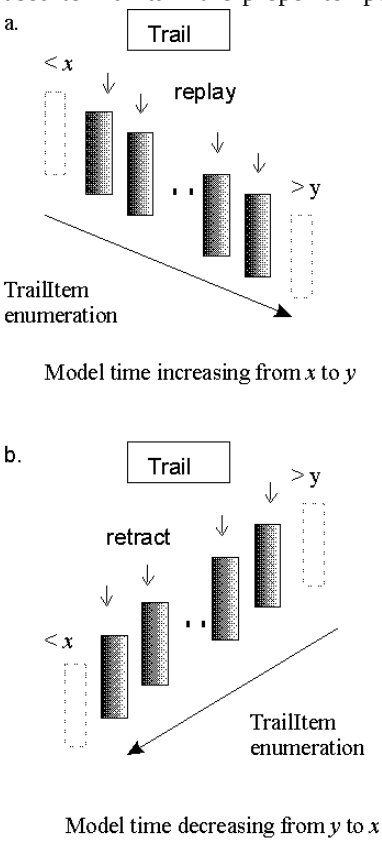


Figure 4. TrailCommand invocation in review mode
a) Increasing model time
b) Decreasing model time

When a state change occurs at a control, the control constructs a new TrailCommand object containing code for executing the state change in both directions. For example, if in a single action a slider value changes from, say, 10 to 12, the TrailCommand object fired from the slider has the following definitions for replay and retract:

```
public void replay(Trailable x) {
    ((HJSlider)x).setValue(12);
}
public void retract(Trailable x) {
    ((HJSlider)x).setValue(10);
}
```

During review mode, the Trail will prevail upon the TrailItem containing this code to invoke the appropriate methods on the control. Note that the design pattern used here follows the "command" prototype described in [7].

The Trail Class

The Trail class lies at the center of the architecture. It is the only class appearing in all three communication modes. During record mode, when a control fires an event containing a TrailItem, the Trail receiving the event consults its TrailGroup for the current model time, gives the TrailItem a time stamp and stores it in an enumeration structure (in this case a Java Vector). During review mode, when user interactions with historian functions cause the TrailGroup to fire clock synchronization events, the Trail iterates through the TrailItem enumeration, invoking retract or replay TrailCommand methods as needed. This is illustrated in Figure 4.

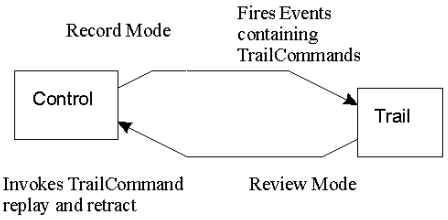


Figure 5. Summary of History Dynamics

It is interesting to note that the client control and its Trail can be viewed as representing the same entity in two different worlds – call them the "real" world and the "history" world. During record mode, when the "real" world is active, external forces manipulate the client, and the Trail's state changes only in response to such actions. This relationship is reversed during review mode, when the "history" world is active: external forces (in this case ClockEvents) manipulate the Trail, and the client is subsequently subject only to Trail-induced state changes. Figure 5 summarizes this duality.

TrailViews

Visualization of the history is the responsibility of the various TrailView classes. Each TrailView is a horizontal panel designed to display data provided by one or more attached Trails. Having localized responsibility for state capture to the Trail class, we are free to custom-design a TrailView panel to display the history over a wide range of visual metaphors and structures.

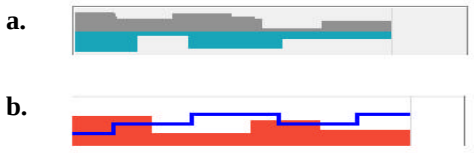


Figure 6. a) PolyTrail of two graph trails
b) OverlayTrail of a line and graph trail

To construct a set of TrailViews, we use a TrailPanel object, which is a special frame for holding them (TrailPanels can be seen on the right side of Figure 2, and in Figure 7 below). This process is well-supported in an IDE such as Visual J++ or J-Builder, where TrailView components can be dropped directly into a TrailPanel. The frame renders its views using a common time scale, and manages scrolling and other such functions.

TrailView subclasses can be richly structured. Base classes include ones to display a single Trail, as in the simple button and slider examples in Figure 2. Other example base TrailViews use various types of graphs for presenting numerical data.

In addition to the base view types, there are view types derived from the class MultiTrailView that render several trails in a single display. Recall that we saw such an example in Figure 2, in which two different dimensions of the rendering (color saturation and envelope width) were used to display the data from two different Trails. In this case the view had to be custom built. However, the process can be simplified by using one or more of the various structuring classes that build compound TrailViews out of TrailView constituents. They include:

- **PolyTrailView** vertically stacks any number of views in a single panel.

- **OverlayTrailView** overlays two or more views in the same panel.

These are illustrated in Figure 6.

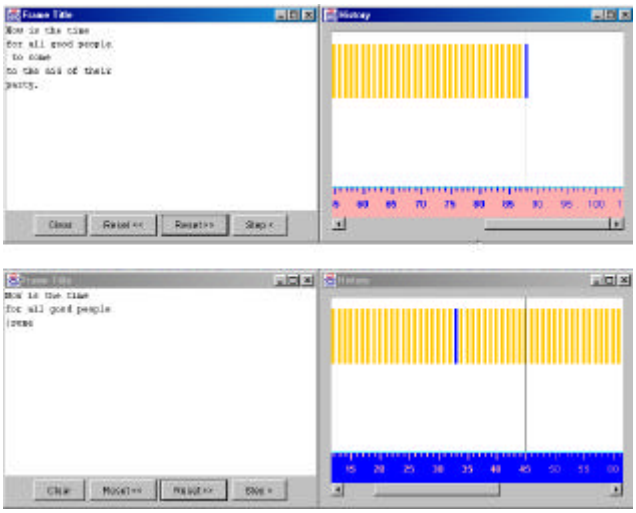


Figure 7. a) Record mode b) Review mode

EXAMPLE: TEXT EDITING

To demonstrate the range of this history model beyond the simulation domain, we attached a visual historian to a standard Java Swing `JTextArea` component [5] to capture all non-trivial edit events. These include character insertion and deletion, and block cut/paste. We took advantage of the Java *undo* package to implement the undo/redo commands. No effort was expended to design an informative visualization beyond showing each event, and discriminating between insertions and deletions. The simulation model clock is not necessary in this domain. Instead, it is sufficient to use autoincrement mode (which increments 1 unit after each event, as described above) to properly order events.

Figure 7 shows this system in two states. In Figure 7a, we have just completed a session of typing into the text window, and editing that window using cut and paste. Each of our actions was recorded and is displayed as a vertical line in the historian window (the darker lines represent deletes). In Figure 7b, we have entered review mode. As the cursor is dragged left and right, each of our actions is retracted or replayed. Our editing session is animated backwards and forwards as the cursor passes over each item.

CONCLUSION AND FUTURE WORK

We have used the current implementation to add history-keeping to deterministic simulations in several domains, including one that models traffic flow [15]. We are also currently constructing a Trails historian to track actions during an on-line database search. The need for proper TrailView visualizations has stimulated research into metaphors for presenting events in these domains.

At the same time, further expansion of Trails functionality is also going forward. We are looking at ways to vary the history model, to include stack and tree-based dynamics. Also, techniques in hierarchical organization are being developed to cope with scaling to larger domains.

In conclusion, the Trails system is intended to serve as the foundation for a robust historian design framework. The client-server architecture facilitates history extensions to existing applications, while the modular design promotes experimentation with different visualization metaphors. The system shows every indication of being adaptable to a variety of diverse domains.

ACKNOWLEDGEMENTS

The author wishes to thank Ben Shneiderman, Harry Hochheiser, and Julia Lawall for their very helpful comments. Thanks also to Ann Rose, Gary Rubloff, Catherine Plaisant and Sumeet Keswani for their contributions to this project.

This research is supported in part by the National Science Foundation under grant EECS 9872701.

REFERENCES

1. Archer, J.E., Conway, R. and Schneider, F.B. User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages and Systems, 6(1),* (1984) 1-19.
2. Ayers, E.Z. and Stasko, J.T. Using graphic history in browsing the World Wide Web. *Proceedings of the Third International World Wide Web Conference*, (1995) Darmstadt, Germany.
3. Carroll, S., Beyerlein, S., Ford, M., and Apple, D. The Learning Assessment Journal as a tool for structured reflection in process education, *Proc. Frontiers in Education'96*, IEEE (1996) 310-313.
4. Catledge, L.D. and Pitkow, J.E. Characterizing browsing strategies in the World Wide Web. *Proceedings of the Third International World Wide Web Conference*, (1995) Darmstadt, Germany.
5. Eckstein, R., Loy, M., and Wood, D. *Java Swing*. Cambridge Mass., O'Reilly, 1998.

6. Fertig, S., Freeman, E., and Gelernter, D. Lifestreams: An alternative to the desktop metaphor. *Proceedings of CHI'96*, ACM Press, 410-411.
7. Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, Addison-Wesley, 1995
8. Ginsburg, S. and Tanaka, K. Computation-Tuple sequences and object histories. *ACM Transactions on Database Systems, 11(2),* (1986) 186-212.
9. Greenberg, S. and Witten, I. H. How users repeat their actions on computers: principles for design of history mechanisms, *Proc. of CHI'88,* Acm Press, 171-178.
10. Kurlander, D. and Feiner, S. A history-based macro by example. *Proc. of UIST'92*, ACM Press, 99-106.
11. Lemaire, B. and Moore J., An improved interface for tutorial dialogues: browsing a visual dialogue history. *Proc. of CHI'94*, ACM Press, 16-22.
12. Lamport, L., Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2), April 1983,190-222.
13. Munro, A., Johnson, M.C., Pizzini, Q.A., Surmon, D.S., and Wogulis, J.L. A Tool for Building Simulation-Based Learning Environments, in *Simulation-Based Learning Technology Workshop Proceedings, ITS'96*, Montreal, Que., Can., June 1996.
14. Plaisant, C., Milash, B., Rose, A., Widoff, S., and Shneiderman, B. Lifelines: Visualizing Personal Histories. *Proc. of CHI'96*, ACM Press, 221-227.
15. Plaisant, C., Tarnoff, P., Keswani, S., Rose, A. Understanding Transportation Management Systems Performance with a Simulation-Based Learning Environment, *Proceedings of ITS'99, Annual Meeting of the Intelligent Transporation Society of America*, April 19-22, Washington DC.
16. Plaisant, C., Rose, A., Rubloff, G., Salter, R., and Shneiderman, B. The design of history mechanisms and their use in collaborative educational simulations. *CSCL'99*, to appear. Available as CS-TR-4027 at http://www.cs.umd.edu/hcil/pubs/tech-reports.shtml
17. Rose, A., Eckard, D., Rubloff, G.W. An application framework for creating simulation-based learning environments, University of Maryland Department of Computer Science Technical Report CS-TR-3907, (May 1998), College Park, MD. Available at http://www.cs.umd.edu/hcil/pubs/tech-reports.shtml
18. Salter, R. Achieving complete synchronization between Java and native processes. , in progress.
19. Shneiderman, B. and Kearsley, G., *Hypertext Hands-On! An Introduction to a New Way of Organizing and Accessing Information*, Addison-Wesley, 1989
20. Tauscher, L and Greenberg, S. How people revisit Web pages: Empirical findings and implications for the design of history systems. *International Journal of Human Computer Studies*, 47(1), (1997).
21. Thimbelby, H. (1990). *User interface design*. New York, ACM Press/Addison Wesley.
22. Toriya, H., Satoh, T., Ueda, K., and Chiyokora, H. Undo and redo operations for solid modeling. *IEEE Comp Graphics and Applications, 6*, 35-42. (1986).
23. Vargo, C.G, Brown, C.E. and Swierenga, S.J. (1992). An evaluation of computer-supported backtracking in a hierarchical database. *Proceedings of the Human Factors Society 36th Annual Meeting*, 356-360.
24. Yang, Y. Undo support models. *International J. of Man-Machine Studies, 28*, (1988). 457-481.