Time Travelling Animated Program Executions

Ken Kahn

Animated Programs and the London Knowledge Lab, The Institute of Education, University of London

kenkahn@toontalk.com

ABSTRACT

Visualizations of program executions are often generated on the fly. This has many advantages relative to off-line generation of animated video files. Video files, however, trivially support flexible viewing via controls that include reverse and fast forward. Here we report on an implementation of time travel that combines the best of both techniques.

In ToonTalk both the construction and execution of programs are animated. Time travel enables the user to move back in time and replay animated executions. The replay can be paused and the user can skip forward or further back in time. The implementation of time travel is based records of every input event and periodic snapshots of the state of the computation.

Keywords

Program visualization, time travel, programming languages for children, ToonTalk

INTRODUCTION

An ideal interactive visualization of a program execution allows the viewer to alter both viewing parameters and the course of the computation being viewed. This paper is about a technique we call time travel that records a visualization of a computation in such a way that the viewer can replay segments that need further attention and skip over segments that are not of interest. Unlike a video, time travel enables a viewer to resume control of the computation at any point in time. Furthermore, it enables the viewer to share with others such visualizations.

The implementation of time travel entails recording of every input event including those from the keyboard, the mouse, the clipboard, and network connections. It also entails the saving of the entire state of a computation periodically. Our experience with a non-optimized implementation is that the overhead of doing this is small enough for a large variety of uses.

A time travel archive is replayed by loading in the initial state and recreating the recorded input event stream. The input events were saved with time stamps so they are replayed with the same timing as the original. If the replay is occurring on a faster computer than the one used for recording the system can idle to slow down to the original timing. If the replay is on a slower computer then the animation can be slowed down so that the replay is reconstructed perfectly except that the playback is in slow motion. This technique relies upon the underlying engine being completely deterministic. If, for example, random numbers are needed then input events should be recorded to enable the reconstruction of an identical sequence of random numbers.

If during replay the viewer wishes to skip ahead and replay from an earlier time then the appropriate saved state is loaded and the input event stream is replayed from the timestamp of the saved state. In ToonTalk the state is typically saved every 5 or 10 seconds so the viewer can then jump to the playback point just prior to the desired playback point. In the worst case they will need to view a few seconds of playback prior to the desired point.

The interface to time travel can be very similar to that of a VCR to enable the viewer to jump to the beginning, jump back in time, resume playback, jump forward in time, jump to the end, and resume recording. Unlike a VCR this implementation scheme for time travel does not allow animated playback in reverse or fast forward. Instead still frames are displayed as the viewer jumps forward or backwards in time. Each jump changes the time by the period with which the entire state was saved (e.g. 5 or 10 seconds).

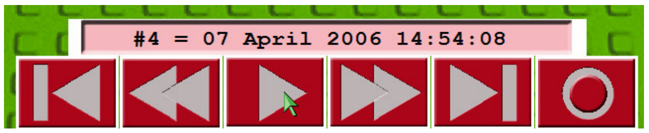

Figure 1: *The time travel interface of ToonTalk*

Various enhancements to the basic scheme have been implemented. These include the ability to add subtitles, recorded narration, or generated speech to the replay log so viewers see subtitles and/or hears narration as they replay time lines.

TOONTALK

ToonTalk has been described elsewhere [Kahn 1996a and Kahn 2006] and here we focus on the program visualization aspects. ToonTalk is unique in that programs are constructed in the same way that their execution is visualized. A user constructs a program fragment by training a robot to manipulate a box of objects. For example, if a robot is trained to remove the numbers in the first two holes of its box, add them together, and give the result to the bird in the third hole, then when that robot runs it recreates those actions. The exact timing and geometry may differ but otherwise program creation and program execution appear the same to an observer. Typically a ToonTalk programmer arranges for only those robots he or she is interested in to be visible while other robots are working unseen in other locations. Robots typically perform their actions from inside of a house, when viewed from the outside one sees only birds flying in and out of the house delivering messages and trucks leaving houses to spawn new computations. When a robot has completed a task it typically blows up the house it is in in order to reclaim resources. The spawning, communication, and termination of independent processes can be viewed from the street or at an adjustable coarser grain from a helicopter flying over a city of ToonTalk computations. [Kahn 1996b]

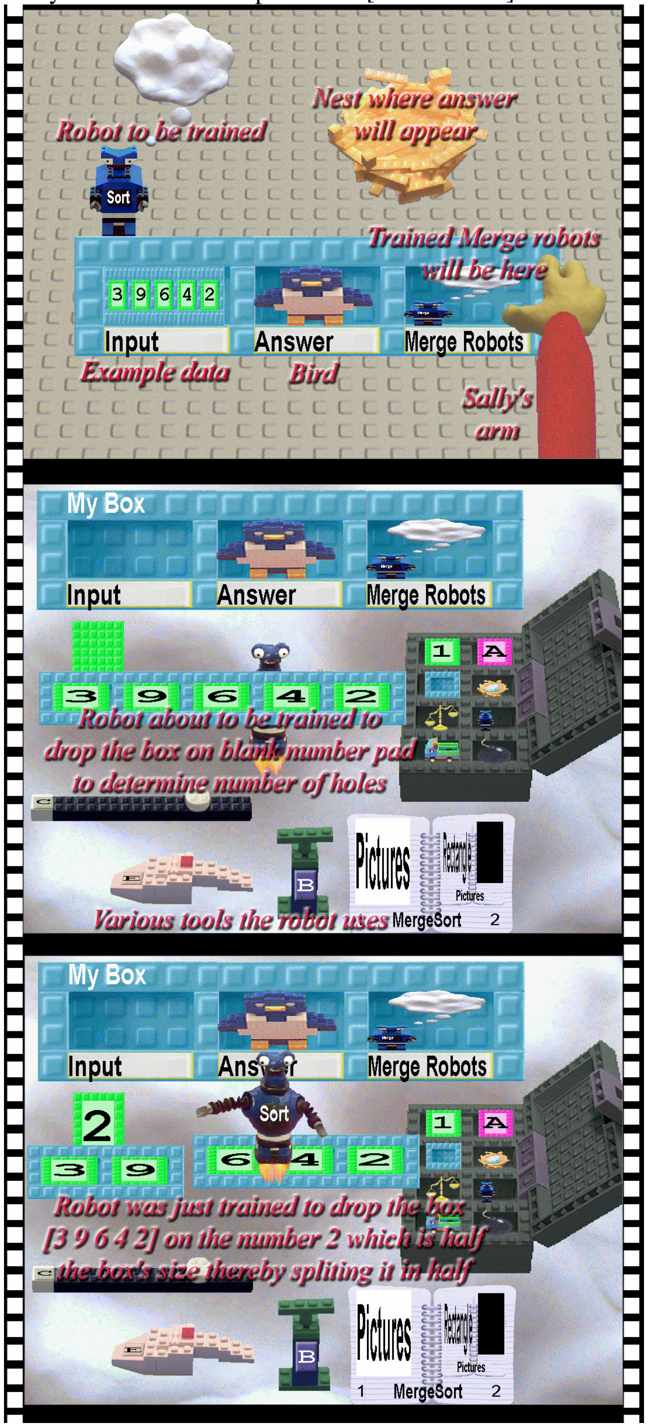
Robot to be trained
Nest where answer will appear
Sort
Trained Merge robots will be here
3 9 6 4 2
Input
Answer
Merge Robots
Example data
Bird
Sally's arm
My Box
Input
Answer
Merge Robots
1
A
3 9 6 4 2
Robot about to be trained to drop the box on blank number pad to determine number of holes
Various tools the robot uses
Pictures
MergeSort
2
My Box
Input
Answer
Merge Robots
Sort
2
3 9
6 4 2
Robot was just trained to drop the box [3 9 6 4 2] on the number 2 which is half the box's size thereby spliting it in half
Pictures
1
MergeSort
2

Figure 2: *Training a Sort robot to split the problem in half*

RELATED WORK

[Atwood et al 1996] describes an implementation of a related notion of time travel in a system called Forms/3. In this system most of the variables are declarative (pure functions of other variables). Those variables that are updated maintain a history of their values. The system is able to go back to previous states and replay an input log. Here we present a time travel technique that is completely general – ToonTalk is a programming language with communication and coordination between dynamically created processes. The technique works for languages with destructive operations on data structures and dynamic creation of objects and processes. Forms/3, in contrast, is very similar to a spreadsheet.

zStep is a Lisp debugger that records state changes to variables and data structures and is able to undo computations to restore previous states [Lieberman and Fry 1995]. It has no replay ability.

Director and Flash provide time lines to organize programs. They support a kind of time travel but only the movement in time between statically organized program fragments. State is not restored when one moves backwards in time.

DISCUSSION

While the implementation of time travel in ToonTalk described here will be available only in the next product release, it has been extensively used by both beta testers and participants in a large-scale European research project called WebLabs [WebLabs 2006]. It has been successfully used to generate scores of demos some as long as 15 minutes. It has been used to replay simulations with different initial parameters. To accomplish this a user runs the simulation, rewinds, changes some parameters, and runs it again. Some users used it as a flexible alternative to a traditional *undo* facility. It was frequently used as a way of reporting system bugs in a way that enabled the bug fixers to readily reproduce the bug. We observed no problems with children as young as ten with the functionality or interface to time travel.

We can characterize the uses of time travel as

- Reflective. Enabling the recounting and refinement of a programmer's actions.
- Constructive. Supporting tinkering with code.
- Communicative. Examples are demos and bug reports.

Various optimizations are possible to minimize the overhead of recording a time travel archive. The ToonTalk implementation periodically saves the entire state of a computation by serializing every object. Typically many objects will not have changed since the last time the state was changed so they needn't be serialized repeatedly. Another unexplored optimization is to save the state of the computation without temporarily freezing the computation. In order to save a consistent snapshot, any portion of the computation that attempts to make changes to an unsaved portion of the state needs to suspend until the state is saved.

Even a well-optimized implementation will encounter situations where the recording overhead causes unacceptably long pauses. ToonTalk gracefully degrades in this situation by repeatedly increasing the duration between the pauses caused by state saving at the price of less flexible temporal navigation on playback.

The ToonTalk implementation of time travel does not permit changes in the viewing parameters during replay. It would clearly be useful to be able to view the same computation from different viewpoints or with different display parameters. This would be consistent with a more principled view of time travel in a computational universe [Deutsch 1997]. Implementing this, however, may restrict the user interface. A mouse click is typically interpreted with respect to both the underlying model (here the computation being observed) and the view of the model that the user is interacting with. This problem could be resolved by recording the high-level consequences of mouse clicks rather the clicks themselves.

When a user travels back in time and clicks on the record button, a branch point in the time line is introduced. In ToonTalk this is implemented by creating a truncated copy of the current time travel archive. A user can then continue with the newly created branch or at any time revert back to previous branches. ToonTalk does not provide an interface to this functionality so a user must choose files in a folder using the operating system interface to move between branches. An interesting avenue for future research is to design and build an interface to these time line branches.

If the original computation was recorded on a slow computer (or a fast one that is slowed down by frequently idling) then when replayed it could run at full speed to implement a kind of “fast forward” capability. The ToonTalk implementation of time travel is capable of doing this but only a command-line interface to this functionality is provided.

An alternative implementation of time travel is to build upon a reversible computation engine [Wikipedia 2006a]. This would enable the viewing of a computation in reverse as an animation rather than as a series of snapshots. In theory it could eliminate the need to record input events or computation states except to support jumping backwards or forwards in time.

Time travel can be added to computer games but a general implementation may make games too easy. A game like *Time Splitters* [Wikipedia 2006b] is a time travel game but is very limited relative to the time travel functionality described here. A game with general time travel has the problem that no game decisions have consequences since they can be undone easily. Game designers generally prefer to have special save points that can be returned to. Only the game *Braid* [Experimental Game Play 2006] supports replay. In Braid the ability to move back and forth in time is essential to solve a series of puzzles.

The time travel technique described here does not apply well to programs that are networked. A single node in a network can be paused and moved back in time but it will not be able to undo the effects of the messages it sent to other nodes. It can however replay recorded incoming messages. When a node is replaying a computation it probably should not also resend outgoing network messages unless the system is designed so that the receipt of duplicates messages has no effect.

Adding the time travel facility to existing software requires that the existing software’s handling of input and output events be programmable either through edits of the source code or an API. It also requires an ability to periodically save the entire state of a computation and to load saved states. Finally, the software must be deterministic so that replay precisely recreates the recorded history.

This paper described a flexible way of viewing computations based upon the concept of time travel. Time travel can be added to any visualizer of computations. It has proved useful within ToonTalk as a means to create demos, run simulations, and debug one’s code. Perhaps it also provides a virtual environment within which one gains a deeper understanding of time and causality.

ACKNOWLEDGMENTS
I am grateful to Yishay Mor and Mikael Kindborg for their comments on this paper.

REFERENCES

[Atwood et. al. 1996] J. W. Atwood, Jr., M. M. Burnett, R. A. Walpole, E. M. Wilcox, and S. Yang, Steering Programs via Time Travel, *Proceedings of the IEEE Symposium on Visual Languages*, Boulder, Colorado, USA; Sept. 3-6, 1996

[Deutsch 1997] David Deutsch, *The Fabric of Reality*, Allen Lane, The Penguin Press, March 1997.

[Experimental Game Play 2006] www.experimental-gameplay.org/2006

[Kahn 1996a] Ken Kahn. ToonTalk™ -- An Animated Programming Environment for Children, *Journal of Visual Languages and Computing*, June 1996.

[Kahn 1996b] Ken Kahn. Seeing Systolic Computations in a Video Game World, *Proceedings of the IEEE Conference on Visual Languages*, Bolder, Colorado, September 1996.

[Kahn 2006] www.toontalk.com

[Lieberman and Fry 1995] Lieberman, H. and Fry, C. Bridging the Gulf Between Code and Behavior in Programming. *Proceeding of CHI’95*: *Human Factors in Computing Systems*, Denver, CO, May 7-11, 1995, 480-486.

[WebLabs 2006] www.lkl.ac.uk/kscope/weblabs/

[Wikipedia 2006a] en.wikipedia.org/wiki/Reversible_computing

[Wikipedia 2006b] en.wikipedia.org/wiki/TimeSplitters