What + When = How: The Timelines Approach to Consistency in Networked Games

Cheryl Savery and T.C. Nicholas Graham
School of Computing, Queen's University
Kingston, Ontario, Canada
Email: {savery | graham}@cs.queensu.ca

***Abstract*—Consistency maintenance techniques used in networked multiplayer games require a tradeoff between the degree of consistency and the responsiveness to player commands. The choice of which technique is most appropriate depends upon the specific game situation. However, all techniques share the need to deal with time as well as with game state data. This can make implementing consistency maintenance techniques difficult. The solution is to have a programming model that is better able to deal with time. In this paper, we present such a programming model, *timelines*. Timelines allow for the explicit treatment of time and have been implemented as part of the *Janus* toolkit.**

I. Introduction

Consistency maintenance in gaming refers to the techniques used to ensure that different players see the same shared state. Networked multiplayer games use a variety of techniques for consistency maintenance such as dead reckoning [1], delaying local inputs [5] and lag compensation [2]. Each technique represents a tradeoff between the degree of consistency and the response time to player actions [8]. As a result, specific techniques are best suited to different game situations [4]. One of the challenges in implementing any consistency maintenance technique is that the programmer needs to deal with time as well as with shared state data, for example coordinating the time a shot was fired to prevent situations where dead men are able to keep shooting [7]. The solution is to have a programming model that is better able to deal with time.

In this paper, we present *timelines*, a novel programming model for shared-state in multiplayer games. Timelines expose the temporal dimension of shared data, allowing programmers to manipulate past and future state values. Timeline variables can be shared between remote players. This allows programmers to modify the rate at which time flows, and to create divergent timelines for different players.

II. The Timelines Model

In the *timelines model*, shared state variables are not represented as instantaneous values in time, but as values indexable by time. Variables represent all the values they have held in the past and all values they will hold in the future. Timelines are composed of:

- Get/set operations that access the timeline's value at a given time; and
- Interpolation and extrapolation functions that estimate values for times when no value is known.

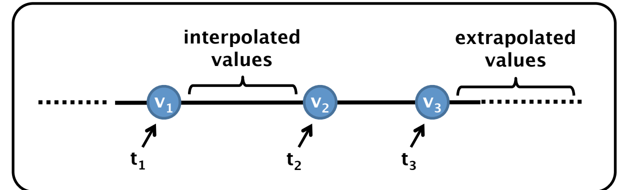

Fig. 1. Elements of the timelines model: a timeline includes a set of past and future values (v_1, v_2, ...), along with the times at which they hold (t_1, t_2, ...). Values between these times are computed using an interpolation function, and values after these times are estimated using an extrapolation function.

Figure 1 shows how these elements are combined. In the timeline, v_1, v_2 and v_3 represent values for times t_1, t_2 and t_3 respectively. These are *known values*, meaning that they have been explicitly set in the timeline.

Consider *fuel*, an integer timeline representing the amount of fuel in a spaceship. Storing values in the timeline is as simple as assigning those values at a specific time. Thus,

```
fuel(-100) = 230;
```

assigns 230 to the integer timeline at 100 ms in the past, and

```
fuel(0) = 200;
```

assigns 200 to the timeline at the current time. (Time references are expressed in milliseconds, may be positive or negative, and are relative to the current time.)

When the value for a given time is not known, it is interpolated or extrapolated from its neighbouring known values. The value of *fuel*(-50) is derived as 215 using linear interpolation between the two known values of 200 and 230, and the value of *fuel*(100) is extrapolated to be 170. Thus, although our timeline currently contains only two known values, any value from the past or future can be determined through interpolation or extrapolation. As more known values are added to the timeline, the interpolated and extrapolated values may change.

The timelines model is fully replicated with each client storing a local timeline object for each portion of the shared state in which it is interested. If two clients create instances of the same timeline, the timelines are automatically synchronized. Figure 2 shows two clients each with a copy of the same timeline. When Client 2 inserts a new value into the local timeline, the value is propagated over the network and a remote update function on Client 1 is invoked. The remote update

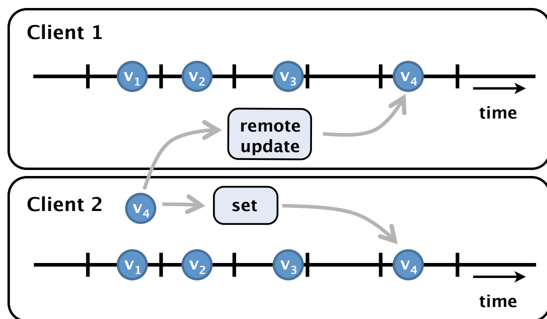

Fig. 2. Timelines are fully replicated. When a client sets a new value into the local timeline, the remote update function inserts the value into the same timeline on all other clients

function specifies how updates arriving over the network are to be applied.

III. Implementing Timelines

We now discuss how timelines are implemented within our *Janus* toolkit. *Janus* has been used by the authors and by other developers to create several games based on Microsoft's XNA game development library.

Within *Janus*, timelines are implemented as objects descended from the *Timeline* class. Each timeline has a base type (the type of the timeline state) and methods implementing interpolation, extrapolation and remote update handling. The default values of these methods can be overridden to create arbitrary timeline types. Timeline objects also provide *Get* and *Set* methods for retrieving/modifying the timeline's state.

Each timeline object has a string identifier. If two clients create timelines with the same identifier, those timelines are automatically synchronized. As shown in figure 2, whenever a timeline is modified, an update is sent over the network to its remote peers. When the remote update is received, it is applied via the remote update handling function. By default, this function simply inserts the new state into the timeline at the correct time; overriding this function can allow easy programming of other interesting behaviours. To support synchronization, *Janus* uses a global clock, which we implemented using the Berkeley algorithm, whose accuracy is in the small tens of milliseconds [6].

Known timeline states (i.e., those that have been set with the *Set* method) are simply organized into a doubly-linked list, where each state is tagged by its time. The shared state object may be as simple as a single integer, or it may be any arbitrarily complex object containing multiple properties.

A timeline's *Get* method is used to retrieve its state at a given time (past, present or future). The Get method uses the timeline's interpolation and extrapolation functions as necessary to provide values at times when none is known. Default implementations of linear and stepwise interpolation/extrapolation are provided, and developers can override these functions to provide domain-specific behaviours.

To reduce bandwidth requirements, each client keeps track of which values have been sent over the network. Prior to sending an update, the client performs a check to determine whether or not remote clients can predict the new updated state. Only if the remote client is unable to predict the new state within a set error threshold will the new state be transmitted. Auto-adaptive dead reckoning schemes [3] can be implemented by changing the error threshold depending upon the game situation or factors such as network congestion and bandwidth availability.

IV. Strengths and Limitations

The power of the timelines model lies in its explicit treatment of time. Automatic interpolation and extrapolation allow programmers to easily access shared state data from any point in the past or future. This technique is extremely powerful for manipulating shared data, although it is not without limitations. The current timeline implementation does not support multiple clients updating the same timeline, as the updates from one client by default will overwrite updates made by the other client. Further research is required to look at how timelines can be adapted to handle such concurrency control problems. Fortunately though, in most gaming situations, updates for a given entity are controlled by a single client or game server.

Timelines currently require the entire shared object to be sent over the network for each update. This makes them unsuitable for large data structures. We plan to explore how timelines can be made more efficient by sending only changes to the shared state, as opposed to sending the entire object.

Timelines provide a novel programming model. For programmers familiar with message passing techniques, the shift to thinking about a shared state model indexable by time can be significant, perhaps analogous to the shift from procedural to object oriented programming. We have found that developers who dive into the model without carefully studying its documentation and examples make the mistake of trying to treat it like a message passing system.

The *Janus* toolkit and documentation are available for download at http://equis.cs.queensu.ca/ equis/Janus.

References

[1] S. Aggarwal, H. Banavar, and A. Khandelwal. Accuracy in dead-reckoning based distributed multi-player games. In *SIGCOMM'04 Workshops*, pages 161–165. ACM Press, 2004.

[2] Y. W. Bernier. Latency compensating methods in client/server in-game protocol design and optimization. In *GDC*, 2001.

[3] W. Cai, F. Lee, and L. Chen. An auto-adaptive dead reckoning algorithm for distributed interactive simulation. In *Proceedings of the Thirteenth Workshop on Parallel and Distributed Simulation*, pages 82–89, 1999.

[4] M. Claypool and K. T. Claypool. Latency and player actions in online games. In *Communications of the ACM*, volume 49, pages 40–45, 2006.

[5] C. Diot and L. Gauthier. A distributed architecture for multiplayer interactive applications on the internet. In *Network*, pages 6–15, 1999.

[6] R. Gusella and S. Zatti. The accuracy of the clock synchronization achieved by TEMPO in Berkeley UNIX 4.3 BSD. *IEEE T-SE*, 15(7):853, 1989.

[7] M. Mauve. How to Keep a Dead Man from Shooting. In *Interactive Distributed Multimedia Systems and Telecommunication Services*, pages 199–204. LNCS 1905, 2000.

[8] C. Savery, T.C.N. Graham, and C. Gutwin. Human factors of consistency maintenance in multiplayer computer games. In *GROUP*, pages 187–196, 2010.