Jim Purbrick
jcp@cs.nott.ac.uk

Chris Greenhalgh
cmg@cs.nott.ac.uk
School of Computer Science and Information Technology,
University of Nottingham

An Extensible Event-Based Infrastructure for Networked Virtual Worlds

Abstract

Many VR platforms emphasize extensibility to support as wide a range of applications as possible. The current trend is to move this extensibility to lower levels of the system to support extensibility of infrastructure mechanisms such as networking protocols. This kind of extensibility allows the runtime of the virtual environment system to evolve even while the system is running. This paper presents a new virtual environment platform that allows multiple infrastructure mechanisms to be added to and coexist within the running system, with different elements of the virtual world using different mechanisms. This allows the virtual environment system to efficiently support a wider range of applications by, for example, having only certain virtual objects use conservative consistency and persistence. It can also optimize the performance of the CVE by tailoring the infrastructure mechanisms according to the different roles played by different objects in the virtual environment.

1 Introduction

All VR systems that attempt to provide a general-purpose platform for a variety of applications must support some form of extensibility or reconfigurability to allow developers to customize the platform for each differing application. At the very least, the system must provide world authoring facilities that allow developers to describe the layout of the virtual world. Often systems also provide facilities for scripting interaction in the virtual world. The VRML (ISO/IEC, 1997) and X3D (Web3D Consortium, 2002) specifications provide for both of these mechanisms, whereas WorldUp (Sense8 Corp., 1998) provides powerful facilities for defining behaviors via a drag-and-drop interface. OpenWorlds (Diefenbach, Mahesh, & Hunt, 1998) goes further by providing facilities to customize the system by adding new scene graph nodes (for example, representing input devices such as trackers) or reimplementing existing nodes (for example, using a new graphics API).

In the case of large-scale networked virtual environments, it is essential that every aspect of the virtual environment platform is extensible and reconfigurable at runtime. When a virtual environment is used by millions of people simultaneously, shutting the system down to provide offline maintenance or upgrades is as impractical as shutting down the entire Web for maintenance. Capps, Watsen, and Zyda (1999) argue that "Cleaning must go on as in a 24/7 burger bar"; that is, high-availability, long-lived shared virtual worlds will demand mechanisms that allow the whole system to evolve while it is still

Presence, *Vol. 12, No. 1, February 2003,* 68–84

running. Although MUD and MOO (Curtis, 1997) text-based virtual environment systems provide rich facilities to extend the content of virtual environment systems at runtime, they stop short of allowing the infrastructure of the system itself to be extended or replaced at runtime.

This level of extensibility and reconfigurability is significantly harder to achieve. For arbitrary elements of the system to be upgraded, extended, or removed at runtime, information about the dependencies present between different parts of the system must be maintained and considered when making runtime changes to the system. It must be possible to reason about when elements of the system can be safely added, replaced, or removed. To make extension and reconfiguration practical, the system should provide a framework that allows developers to reason about the effects of changes without having to consider the details of every other element in the system. A number of systems have attempted to provide this level of reconfigurability to a greater or lesser extent. The NPSNET-V system (Capps et al., 2000) provides a particular pattern of extensibility (leveraging Java's class-loading capabilities) that allows network protocols, objects classes, and graphical models to be dynamically added to the running system. The DEVA (Pettifer, Cook, Marsh, & West, 2000) system adopts a model of long-lived server environments, which dynamically load and compose objects from behavior fragments, to create flexible and composable object and world behaviors. Sony's Community Place (Lea, Honda, Matsuda, & Matsuda, 1997) allows the runtime addition of applications that register with the Community Place server that then routes messages between clients and the registered application. At a lower level, the Bamboo system (Watsen & Zyda, 1998) provides a dynamic module loading system that spans multiple languages, and that is intended to support extensible virtual environment systems of this kind.

In this paper, we describe our own approach to creating a flexible and extensible infrastructure for networked virtual environments, based on the distributed event model employed in MASSIVE-3. In section 2, we present some background motivations for our own approach to these issues. Section 3 describes the first element of our approach: distributed event filters, the basis of our flexible infrastructure. Section 4 introduces the notion of "deep behaviors," which is the means by which we manage this flexible infrastructure. Section 5 illustrates this with some example configurations of our system. Section 6 presents some quantitative results of this approach. Finally, section 7 gives our conclusions.

2 Background and Motivations

This work grew out of our earlier explorations of persistence in virtual environments (Purbrick & Greenhalgh, 2001). In that work, we added simple facilities for persistence and in-world editing to MASSIVE-3 and arranged for a number of groups of users to explore and modify a virtual museum over a period of several weeks. Using the temporal link facilities of MASSIVE-3 (Greenhalgh et al., 2000 and Greenhalgh, Flintham, Purbrick, & Benford, 2002), we recorded all of this activity for subsequent analysis. These recordings contain all of the virtual world content and all of the updates that are applied to it, that is, everything that happens in the virtual world from the system's perspective.

In this analysis, we examined the different patterns of use (creation, update, and deletion) of the different kinds of data in the virtual environment, including users' avatars, walls, and other artifacts.

Over the whole experimental log, we found that 38 embodiments, each with two major sub-objects (body and hand) were updated (that is, changed) a total of 372,765 times. In contrast, 596 non-embodiment objects were updated a total of 39,665 times. So we found approximately ten times as many embodiment updates applied to fewer objects.

Figure 1 shows the distributions of the life spans of added items classified by geometry. The life spans are quantized to improve clarity. The graph shows clear differentiation in the life cycles of the added items. The Lichtenstein picture clings to the bottom of the graph, and 80% of these pictures remained at the end of the experiment compared to the less popular Miro picture, which tended to be added, evaluated, and deleted within a few seconds, with only 25% remaining at the

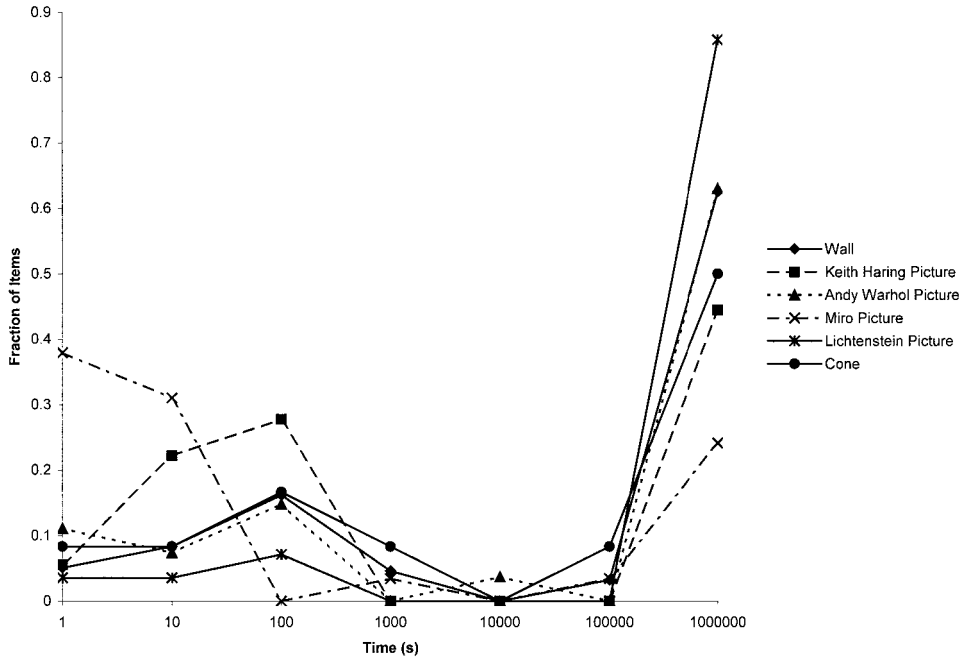

Figure 1. *Item life span by geometry.*

end of the experiment. Although walls were important landmarks that provided structure in the world, the graph shows that approximately 35% were deleted in the session in which they were created, with many people repeatedly adding, manipulating, and deleting walls before they ended up with the desired configuration.

A similar pattern is shown in figure 2, which shows item update times from creation, classified by geometry, for added items, quantized to improve clarity. The Miro pictures were hardly updated at all after the session in which they were created, as most were deleted, whereas the cone geometries that were left in the world after the session in which they were added continued to be edited in subsequent sessions—with 45% of their updates occurring in later sessions.

These results show that different types of items had a rather different characteristic "life cycle." The data representing a user's avatar was the most volatile and the least suitable for caching or being made persistent, and some types of added items were transient whereas others had longer life spans. Consequently, we sought to extend the MASSIVE-3 system to allow different items to be treated in different ways by the infrastructure. For example, we wanted to be able to apply different forms of consistency, persistence, access control, and caching to different items within the same virtual world. Rather than repeatedly extending and elaborating a monolithic runtime system, we chose to significantly reengineer the system to make it dynamically extensible and tailorable at the level of individual data items within the shared virtual world. This design is the subject of this paper.

3 Distributed Event Filters

The starting point for our new system was MASSIVE-3 (Greenhalgh, Purbrick, & Snowdon, 2000), which uses explicit event objects to represent all proposed changes to the shared virtual world. These are generated by the system API and routed around the distributed system in a well-defined way. This approach was designed to allow future mechanisms to adapt the system by using reflection to introspect the system (such as tailoring system performance based on the events be-

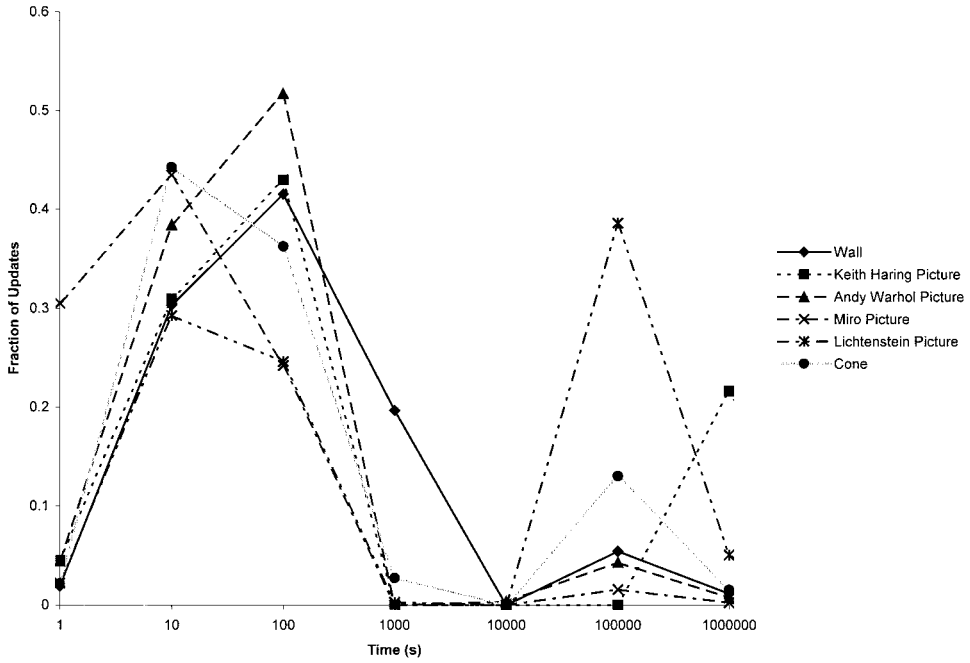

Figure 2. *Update times from item creation by geometry.*

ing generated or processed). Figure 3 shows the specific event distribution architecture used in MASSIVE-3. An application generates events (requests for changes to the shared virtual world) via an event-generating API. These events are passed on to a "sending" event pipe for distribution to the server, and to a "pending" event pipe for local enactment (actually updating the local database). Events sent to the server are redistributed to the other clients of the same locale (portion of a virtual world).

What we have done in the work described here is to step back from the specific behavior of MASSIVE-3 to view it as just one possible configuration of an extensible set of infrastructural components. MASSIVE-3 enforced certain infrastructural behaviors:

- Every event was passed to the sending and pending event pipes.
- Every event leaving the sending event pipe was sent to the server.
- Every event received by the server was queued to be sent to all other clients.

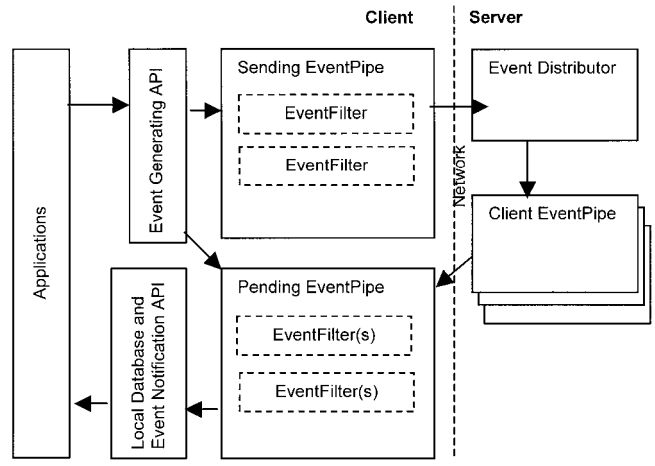

Figure 3. *MASSIVE-3's event distribution architecture.*

- Every event leaving the pending event pipe was enacted.

Our new approach has only one constraint: every generated event will be passed to a certain well-known event pipe. All of the rest of the system's behavior (that

is, everything that happens to the event in that event pipe and subsequently) is encapsulated in the "event filters" that populate this and other event pipes within the system, and is subject to dynamic customization. We term this approach *distributed event filters* (DEF), because the overall behavior of the system is the result of the coordinated activities of potentially many event filters distributed across multiple event pipes in all of the various clients and servers that compose the system. The DEF framework provides a number of facilities that ease the development, configuration, and deployment of individual event filters and complete infrastructure mechanisms. These facilities are as follows.

- Event pattern matching: the framework takes care of determining which filters apply to each event to avoid each filter having to test each event for applicability.
- Event list processing: the event filter interface provides an easy way for filters to add or remove events from an event pipe.
- Constraints and requirements: the framework has support for relative positioning of filters in an event pipe.
- Identification and versioning: support for identifying general or specific functionality of a filter, its version, backwards compatibility with earlier versions of the filter and identity.
- Communication: support for common patterns of communication between filters.

These facilities are discussed in detail in subsections 3.1 to 3.7. In addition to easing the task of developing and deploying infrastructure mechanisms, moving these facilities into a framework can lead to efficiency gains as shown by the filter list caching and prioritization mechanisms discussed in subsections 3.3 and 3.4. These optimizations are possible only because the framework takes care of the pattern matching required to determine which filters apply to a given event.

3.1 Event Pattern Matching

The various events that are generated and flow through the system may be treated in different ways, that is, have different sets of event filters applied to them. This is based on a simple pattern-matching mechanism that selects the filters to be applied to each event, typically on the basis of the virtual item to which the event applies. Therefore, we can create arrangements of tailored event filters that apply different forms of consistency, persistence, and access control to different items within the same virtual world.

3.2 Event List Processing

To allow filters to easily delete or synthesize events, a list is passed to the filter's processing method containing the single event to be processed. If the filter wants to stop the event being processed further, it removes the event from the list and returns the empty list. If a filter wants to create new events, they can be added to the returned list, whereas filters needing to change events can either rewrite the event in the list or remove it and replace it with a new event. The events returned to the event pipe are marked as having been processed by the filter and are then processed by the next applicable filter; thus, this mechanism is suitable for filters such as interpolators—generating new intermediate events that must not be interpolated themselves (to avoid infinite loops of events being generated). Filters can also generate events directly or indirectly through API calls (rather than in the returned event list) that are fully processed by the event pipe.

3.3 Filter List Caching

To accommodate potential changes to events, the event pipe, in principle, must reevaluate the list of applicable filters after any filter performs processing. To allow flexible filtering and reasonable performance, the event pipe caches the last used filter list, the next filter to be applied in the list, and the event parameters used to construct the list. If the next event to be processed has matching parameters, the cached filter list can be used. If most filters are passive and do not change events, this cache will normally avoid the reevaluation of applicable filters during the course of a single event's processing. The cached filters can also be used between

events whenever consecutive events share parameters, as is often the case in collaborative virtual environments where streams of updates to an object are often generated. Depending on the relative costs of evaluating cached filter sets, generating filter sets, and the likelihood of small clusters of event parameters being processed, the cache of filter sets can be expanded arbitrarily. This is especially useful when an event pipe in a client application is processing updates to the user's avatar: a large proportion of the events being processed may be updates to various parts of the avatar. By maintaining a cache as large as the number of avatar items, the event pipe very rarely needs to generate a new filter set.

3.4 Prioritization

Allowing filters to generate events that must be fully processed by the event pipe requires a choice to be made about the semantics of the event pipe: either the addition of a new event to the pipe causes recursive processing of the new event to completion, or the event pipe could prioritize events that are further along the pipe, so that, if a called filter generated an event, the new event would be processed only when the original event had moved completely through the pipe. The latter semantics are more appropriate for a number of reasons. Firstly, they minimize the latency caused by event processing as the pipe will always attempt to process the event that needs the least processing to move completely through the pipe. Where event pipes are connected to event buffers (which may be inspected by filters), the latter semantics also ensure that as many events as possible are available to filters for inspection. If a filter generates an event as a result of processing an initial event, prioritizing the initial event ensures that it is available for inspection in the buffer before the newly generated event is processed. Finally, the latter semantics are more efficient as fewer events remain partially processed at any one time, consuming memory rather than being completely processed and then deleted. To implement these semantics, the event pipe must maintain a buffer of events being processed, must always pick the event furthest down the event pipe for further processing, and must continue processing until no events remain in the buffer. Adding an event to an event pipe causes it to process events until no events remain in its buffer. Consequently, if an event is added to the event pipe when the buffer is not empty, the event pipe must already be processing events, and so the event can simply be added to the buffer and left to be processed along with the other events in the buffer.

3.5 Constraints and Requirements

Because the relative positions of filters in an event pipe are important (often dramatically changing the infrastructure semantics and sometimes being the only difference between two infrastructure mechanisms), rich support for specifying positions and dependencies among filters is provided by the framework. The framework permits filters to specify constraints. These describe which filters, if they exist in the event pipe, must come before or after the filter. Similarly, filters can specify requirements. These list filters that *must* exist in the event pipe before or after the filter. This system of constraints and requirements provides a simple yet powerful way of determining the relative positions of filters and the dependencies between them. To set up an event pipe in a certain configuration, the required filters are created, constraints and requirements are added to the filters, and then they are added to the event pipe. The event pipe then attempts to satisfy the constraints and requirements for each filter. If the constraints can be satisfied, the filter is added to the pipe; otherwise, the failure is indicated and corrective action is taken, either changing the requirements, aborting the initiation of the infrastructure mechanism, or halting system execution as appropriate.

Filter requirements are also used to ensure that the removal of filters from an event pipe does not break any dependencies. The event pipe attempts to satisfy the requirements of all other filters without the filter or filters being removed. If all requirements can be satisfied, the filter can be removed. These semantics for addition and removal of filters ensure that the event pipe remains in a valid state at all times.

3.6 Identification and Versioning

To specify constraints and requirements, filters must be able to reference other filters or types of filters. In addition, the versioning of filters must be supported to reason about the compatibility of old and new filters, to replace old filters, and to determine when old, unused filters can be removed from the running system. To allow for this, the framework supports a hierarchical naming scheme that allows the general and specific identity of a filter to be discovered. The name is made up of the form <Function>.<Version>.<Identity>, in which function is a sequence of strings describing the filter's function in increasingly specific terms and version is a sequence of strings describing the filter's version in increasingly specific terms. Identity is a single integer that is assigned sequentially to the filters created in an application, allowing the precise specification of an individual filter. Using this scheme, a filter may specify that it must be positioned before or after a general class of filters, before or after a specific filter of a certain version, or before or after a particular filter. Requirements should be made as general as possible to allow the reconfiguration of the event pipe while maintaining critical dependencies between filters. The model uses the Java versioning scheme (Gosling, Joy, & Steele, 1996) where 1.2 maintains backward compatibility with 1.1, whereas 2.1 breaks this compatibility. This allows substring matching of constraints to find all compatible versions. Version 2 of a *function* filter that supports the interface of version 1 should be named *function*.1.1, so that a filter with a requirement of *function*.1 will remain satisfied, whereas, if the new version behaves differently, it should be named *function*.2 to signal to the event pipe that the existing requirement can no longer be satisfied.

3.7 Communication

The same mechanisms that transport events relating to items in the virtual world can be used for communication between the filters that implement the infrastructure supporting the virtual world. The identification and versioning facilities (above) allow valid potential receiving filters to be found either explicitly, through querying event pipes, or implicitly by specifying constraints, which will assure the relative positions of multiple filters. Communication between filters can then be achieved by the sending filter, adding a special-purpose notification event either to the returned event list (to communicate with "downstream" filters further along the event pipe) or to an event pipe (to communicate with "upstream" filters positioned before the sending filter, or to filters in other event pipes). Although this method of communication is simple, controlling an infrastructure mechanism often involves disseminating control information to a number of event filters that are potentially distributed among a number of processes that implement the mechanism. To simplify this configuration, we instead embed a behavior node in the scene graph that takes care of adding and removing event filters from different processes and controlling those filters when the properties of the behavioral node change. We call the behavioral nodes that control the infrastructure elements of the system *deep behaviors,* which are discussed in section 4.

3.8 DEF MASSIVE-3

Figure 4 shows a distributed event filter (DEF) configuration that emulates the previous operation of MASSIVE-3. The API event pipe is the common starting point for all events. MASSIVE-3's default infrastructure activities have been encapsulated as the following event filters.

- *ConstraintsFilter*: enforces an explicit ordering on all events (part of MASSIVE-3's exploration of consistency mechanisms)
- *LocalNowRouting*: sends a copy of the event to the local pending pipe for immediate enactment, and another copy of the event to the sending pipe for distribution
- *Unicast*: sends the event to the server over a TCP connection
- *UpdateSceneGraph*: enacts the event on the local database replica

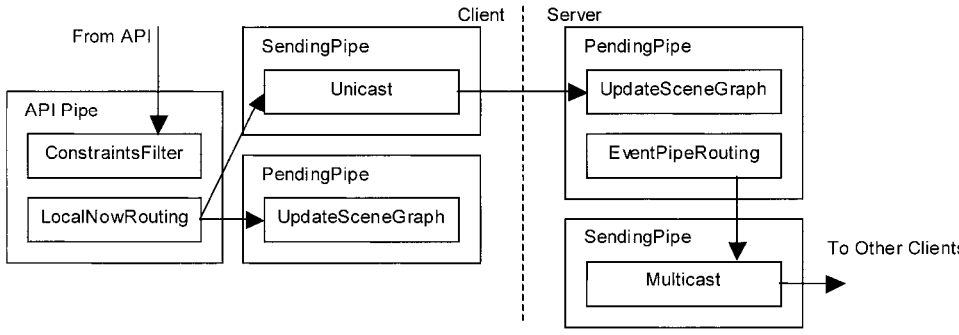

Figure 4. *Default DEF configuration, emulating MASSIVE-3.*

- *EventPipeRouting*: passes the event to another specified event pipe
- *Multicast*: sends the event to all connected clients (with the optional exception of the originating client)

The API pipe is always present in the system, whereas the other pipes are added dynamically as they are requested by the event filters.

Having reengineered MASSIVE-3's static infrastructure as a flexible and extensible infrastructure, we moved beyond this emulation of MASSIVE-3 to add other event filters and to experiment with other configurations of event filters, some of which are described in section 5.

4 Deep Behaviors

The previous section has outlined our distributed event filter (DEF) approach to constructing a flexible and extensible runtime infrastructure. Flexibility is achieved by applying different sets of event filters to different virtual world data items, and extensibility is achieved through the addition of new kinds of event filters and through the dynamic (re)configuration of event filters and event pipes within the running system. However, we still require a way to specify, modify, and realize the particular arrangements of event filters and event pipes to be used for particular data items or environments.

This leads to the second key element of our new architecture, which we term *deep behaviors*. In virtual environments, behaviors are typically pieces of executable program code that describe the dynamics of (part of) the virtual world (such as animations or responses to user interaction). So we use the term *deep behavior* to refer to pieces of program code (or the equivalent) that are used to describe the dynamics of the infrastructure, that is, the "deep" or low-level behavior of the system.

We have chosen to make these deep behaviors explicit within the shared world data as annotations that can be applied to the shared data items that compose the virtual world (for example, as shared scene graph nodes and annotations). A deep behavior provides a data item with infrastructure functionality—for example, making the item persistent, subject to transactions, or subject to total ordering consistency. It does this by manipulating the event filters that operate on the item. Where a normal behavior might manipulate an object's position to make it follow terrain, a deep behavior manipulates the filters that process the events describing the object's position (for example, controlling the way that position changes are propagated through the network).

By making (the declarations of) deep behaviors part of the shared state of the virtual world, we can exploit the normal (default) data distribution mechanisms to distribute deep behaviors around the system as required. This allows them to affect event filters and event pipes on multiple machines in a coordinated fashion. We can also apply deep behaviors to other deep behaviors, such

as to specify the persistence or consistency mechanisms to be applied to the deep behaviors themselves.

By providing deep behaviors as a layer of abstraction above the basic event filter infrastructure, we also hope to make extensibility and configurability more meaningful to users and world builders. For example, a deep behavior might be selected from a palette such as "trusted persistent," "important but slow," "unimportant and fast," and so on. We suggest that deep behaviors should specify the "mutability" of virtual items, which incorporates all aspects of creation and change. We prefer this approach to that of focusing down on the component elements of consistency, persistence, access control, and so on because these are typically intertwined.

Figure 5 illustrates the scene graph fragments in MASSIVE-3 corresponding to a default (nonpersistent) and a persistent virtual object. Each box represents a node within the shared scene graph; an Entity node specifies a 3D transformation, and a Geometry node specifies a 3D geometry (by filename in this case). From a programmer's or world author's perspective, a deep behavior is added to a virtual object simply by adding a DeepBehavior node to the corresponding Entity (which requires a single line of C++ code, an entry in a world definition file, or a mouse click in a graphical editor). When the DeepBehavior node is added to the local scene graph, it executes the corresponding deep behavior code, which in turn creates and configures event pipes and filters as appropriate. Similarly, when a DeepBehavior node is removed from the scene graph, the deep behavior code reverses this process.

Because deep behaviors are first class items in the scene graph, the deep behaviors of deep behaviors themselves can be specified. These associations allow a potentially infinite number of levels of meta-meta-information and a rich syntax for composing complex, parameterized deep behaviors from combinations of simple behaviors.

For example, if changes to a deep behavior might have potentially hazardous effects on the continued running of a virtual environment system, an access control deep behavior might be used to annotate it. The access control behavior could restrict access to the deep behavior item in exactly the same way as it would restrict access to any other item. Without changing either deep behavior, the combination of behaviors provides new and useful functionality. If the access control mechanism was later replaced with another mechanism, the meta-annotation could be replaced and the original behavior could take advantage of the new access control facilities without any change.

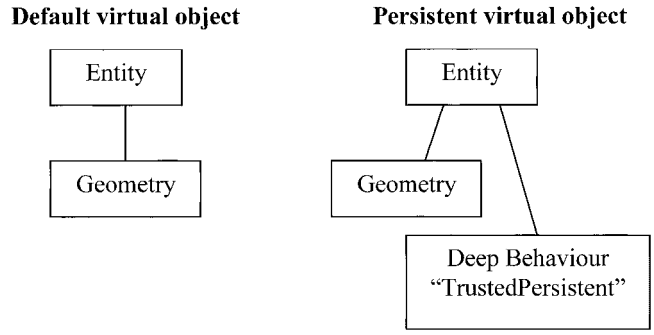

Figure 5. *Default and persistent virtual objects.*

There are situations in which the annotation of deep behaviors can lead to infinite regressions. In the preceding example, there initially seems to be no problem in annotating the access control deep behavior with another access control deep behavior—the second access control behavior specifies the users able to change the users able to change the root behavior. However, the leaf node in this scene graph must be an access control behavior that cannot be annotated, in order to provide a fix point.

5 Examples

This section provides five examples of prototyped deep behaviors, their corresponding event filter networks, and descriptions of how they could be used within virtual environments. Other examples include the default MASSIVE-3 behavior from section 3, and the delayed persistence behavior, evaluated in section 6. Recall that the world designer or programmer would specify the deep behavior for a particular data item or set of data items (that is, part of the virtual world's con-

tent), and that the result of the specified deep behavior within the running system would be to establish the corresponding network of event pipes and event filters to achieve the desired form of mutability.

5.1 Trusted Persistence

This deep behavior might be used for items whose persistence (and durability of change) must be assured, such as item representing major world features (such as landmarks) or items with financial significance (such as virtual bank accounts).

This is one of the most important examples of the DEF and deep behavior framework as it demonstrates the interdependence of infrastructure mechanisms. The mechanism comprises a total ordering consistency mechanism (quite conservative) and server-based persistence. The motivation is to provide server-side persistence for important objects in the virtual world. This could be achieved in a simple way by inserting a filter into the server pending pipe that wrote every event processed by it to storage. However, in this case, users would be unaware of when the important items were actually made persistent: the user would make an update and immediately see its results. However, only at some arbitrary time later would their update become durable, and the user would have no idea when this was. This simple persistence mechanism would effectively provide the user with a view of the *predicted* persistent state of the item through the immediate update of the local replica. If a failure occurred before the update was written to the server store, then the update would be lost and the prediction would be false. In cases in which the knowledge of the durable state of the world is more important than local interaction times (for example, when updating a virtual bank account), the system must route updates to the server, which makes them persistent, before returning the updates to the client where they are applied to the local replica. This mechanism would ensure consistency between the persistent state and the client's view of the world. The client can then trust the local state of the item as it is no longer a *prediction* of durability. These semantics are closer to those of a database than the typically optimistic mechanisms of virtual environment systems, but they would be useful for some items in some virtual worlds. By providing this behavior as an option for specific data items, the gamut of applications that can be implemented by the virtual environment system is increased.

To implement these semantics, a routing filter is added to the client application's API pipe before the standard routing event filter. (Flexible ordering of event filters is a key component of the DEF implementation.) Instead of copying the event to the pending and sending pipes, the filter just adds it to the sending pipe. A filter is added to the server's pending pipe that make the update persistent before applying it to the server replica, and a second filter is added after this that sends the event back to the client. This configuration is shown in figure 6.

5.2 Variants

The "variant" deep behavior demonstrates the flexibility of the framework by providing facilities not usually provided by virtual environment systems. Rather than allowing arbitrary updates to items, updates to items tagged with the variant behavior create "proxy" items related to the original item by a syntactic consistency mechanism (Terry, Petersen, Spreizer, & Theimer, 1998). Other clients viewing the item see its original state and can themselves create related proxy items representing their desired changes to the state of the item. The actual mechanism for creating these subjective views and relating the proxy to the original item will depend on the awareness management facilities of the virtual environment system, but the prototype implementation (Greenhalgh, Purbrick, & Snowdon, 2000) used aspects to create overlay environments for each variant. The awareness management facilities can then be manipulated by an administrator to view the different versions of the item and authorize some or all of the updates. This behavior is useful in situations in which user evolution of a virtual world is desirable, but control over the rate of change, and protection against virtual vandalism, is required. Instead of updating the shared state of the item, users create desired versions of

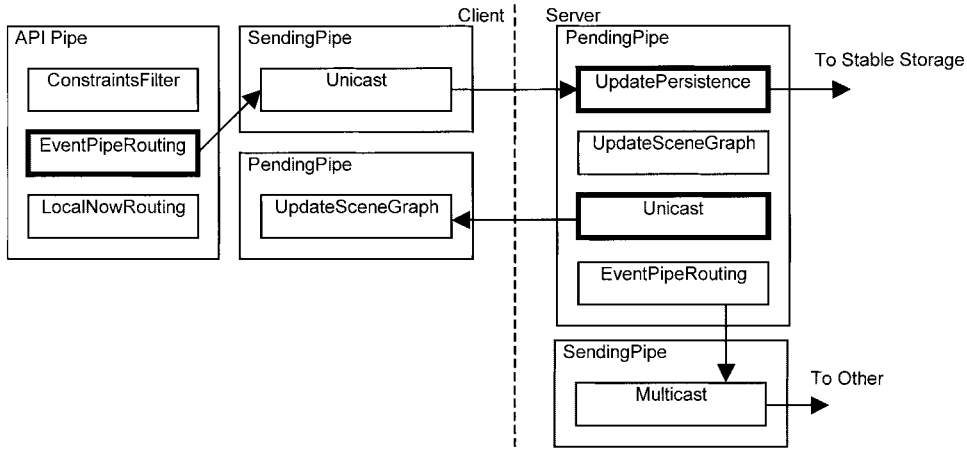

Figure 6. *Trusted persistence DEF configuration.*

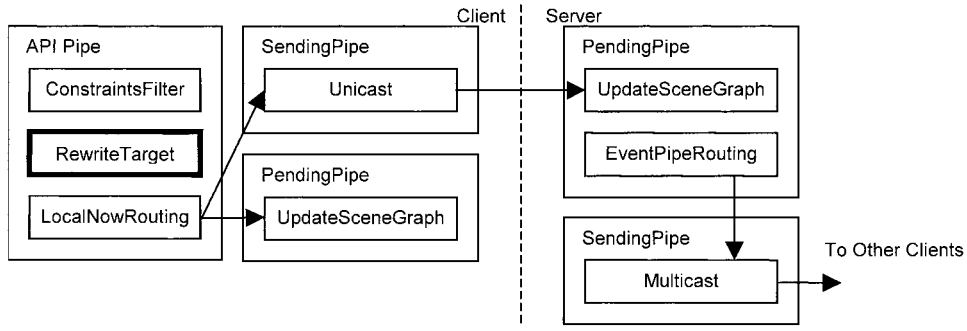

Figure 7. *Variant DEF configuration.*

items that must be approved before they become shared.

To implement variants, the deep behavior first creates a subjective proxy item and then inserts a rewrite filter in the client's API pipe that processes updates to the original item by rewriting the target of the update to be the variant item. This causes subsequent updates to the original item to be applied to the proxy instead. The filter configuration is shown in figure 7.

5.3 Leases

The lease deep behavior is used to provide a time-limited guarantee of immutability for items. The main motivations are to fix parts of the virtual world without committing to a permanently static state and to allow reasoning about the validity of the world for disconnected operation and intelligent caching of the world's state. Like Jini leases (Waldo, 1999), the semantics of the lease deep behavior are to declare the information annotated by the lease as valid for at least the duration of the lease. Whereas Jini leases guarantee the validity of a service for a time, the lease deep behavior guarantees the validity of the state of an item. Like Jini leases, the lease can also be extended. This is useful for defining parts of a virtual world as static for the foreseeable future, where the foreseeable future is the length of the lease. If at the end of the lease period the item should remain static, the lease can be renewed and clients can continue caching and using the item for disconnected

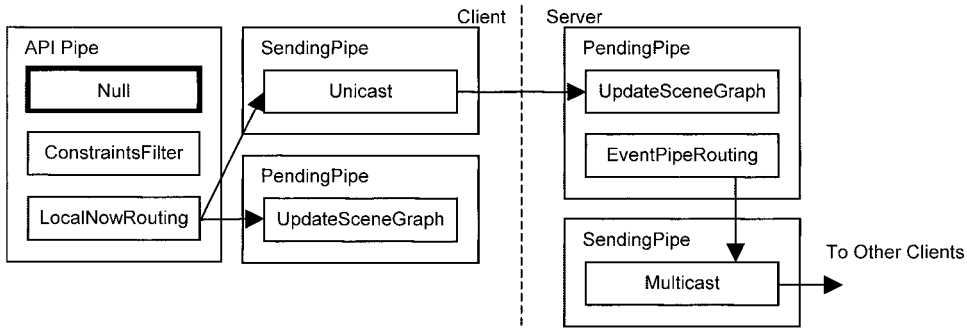

Figure 8. *Lease DEF configuration.*

operation. If during the lease period it is decided that the item should be changed, then the lease can be allowed to expire and the item changed. These "never say never" semantics provide a useful middle ground between declaring an item permanently static as in VRML (Carey et al., 1997) or always transient as in MASSIVE (Greenhalgh, Purbrick, & Snowdon, 2000).

These semantics are implemented by a simple NullFilter that removes all updates to an item that is inserted by the lease deep behavior on creation and removed on expiry. For most efficiency, the NullFilter is inserted as close to the source of updates as possible: at the front of the API pipe as shown in figure 8.

5.4 Triggers

Where leases guarantee the immutability of an item for a certain period of time, trigger behaviors indicate a scheduled change to the item they annotate and provide a mechanism for that change. Like leases, triggers have an expiry time and can be renewed. When the trigger expires, it performs an action by injecting arbitrary events into arbitrary event pipes. This mechanism allows triggers to be as general as possible as they rely only on the existence of event pipes and events, yet they can perform any action the system API can perform by the arbitrary sequencing of events. The motivation for triggers are the results of the experiments described in section 3: many items were created, heavily modified, and then discarded in a short period of time, whereas items that survived this initial period tended to exist for a much longer period of time. By annotating new items with a trigger expiring after this initial "hot" period of manipulation and setting the trigger to add a persistence behavior to the item, the system can be significantly optimized: of the many updates made to items after their creation, only one state need be written to storage for each item that survives its turbulent youth. More generally, triggers provide a mechanism for managing the lifetime of objects by updating, adding, or removing other deep behaviors applying to an item based on time or events applied to an item. In this sense, triggers are mainly used as a meta-deep behavior that coordinates changes to other behaviors allowing the behavior of objects to vary dynamically through its life. Triggers are used to implement the delayed persistence deep behavior discussed in subsection 6.1. Items are initially created with a trigger annotation that annotates its parent with a ServerPersistence deep behavior when the trigger's timer expires.

5.5 Batch Updates

The batch updates behavior is an example of a bottom-up deep behavior motivated by the desire to optimize the operation of the virtual environment system by restricting the way in which the environment can change. When a batch update behavior applies to an item, any update to that item is delayed to the next batch period. Effectively, the batch update behavior quantizes the times at which an item can change. If the deep behavior framework makes the batch times avail-

able in the virtual world (as in the prototype implementation), the limits on when changes can occur can be used to drive caching and disconnected operation. As the system knows that the item cannot change until the next batch period, its state can be cached without checking for cache consistency and can be presented to the user as valid during periods of disconnection. In addition, early updates to items buffered until the next update point can be discarded completely if new updates to the item are delivered before the batch point. Given a batch period of *n* seconds, a stream of updates is effectively rate limited to one update per *n* seconds. Where many items share a batch update deep behavior as described in the discussion on scalability, the effect of the update behavior is to create large batches of updates that are applied to large numbers of items in the environment simultaneously. Given sufficient behavior sharing and sufficiently long batch periods, the batch update behavior can be used to facilitate applications that physically mail out periodic updates on CD. The behavior ensures that the environment will remain static and so needs to be downloaded only once; then, when the CD arrives, updates can be applied en masse without the need to download them. This model is very attractive to applications presenting large, rich environments accessed over low-bandwidth connections. An obvious potential problem with the batch updates behavior is that the new state is not immediately seen by the user performing the update, but this can be solved using the proxy item techniques mentioned in the previous discussion on the variant behavior.

6 Validation

To test and validate our new architecture and implementation, we have again made use of the virtual recordings described in section 2. However, rather than simply replaying or analyzing the activity as it occurred, we use the recordings as input to our new prototype system. In this way, we can explore and measure the behavior of the system in different configurations against a repeatable and realistic corpus of virtual world activity.

6.1 Delayed Persistence

As already noted, our starting point for this work was our consideration of persistence in collaborative virtual environments. In the initial experiments, we observed that different kinds of items have different requirements for persistence. We were also able to analyze some of the temporal characteristics of virtual world updates in the experiments. For example, we observed that updates to virtual objects often occur in rapid sequences, with much longer gaps between these sequences. Each sequence of updates corresponds to a period of time during which a user is actively holding and manipulating an object.

This motivated us to consider a deep behavior that makes changes persistent only after a certain period of time has elapsed. In the event of a system failure, this approach would lose very recent updates but would retain updates that had been stable for longer. This deep behavior is implemented using a DelayedPersistence event filter in the server's pending event pipe.

We re-ran the recordings through our new system with this deep behavior for a range of different time delays to persistence. We measured the amount of data written to the persistent store (a relational database accessed via ODBC). The results can be seen in figure 9, for each kind of virtual item in the experiment.

We see that by not making embodiment items persistent we could immediately reduce the amount of database traffic by approximately 75% ("embodiments" versus "added items"). This is easily achieved by applying the persistence deep behavior to only the added items. We also see that a delay to persistence of 120 sec. (2 min.) more than halves the remaining database traffic, with only a limited effect on the long-term persistence of the system.

Note that this delayed persistence can be dynamically introduced to the running system for any data item simply by adding the corresponding deep behavior to that item.

6.2 Caching

In the previous sections, we have shown how deep behaviors can directly modify the runtime infrastructure

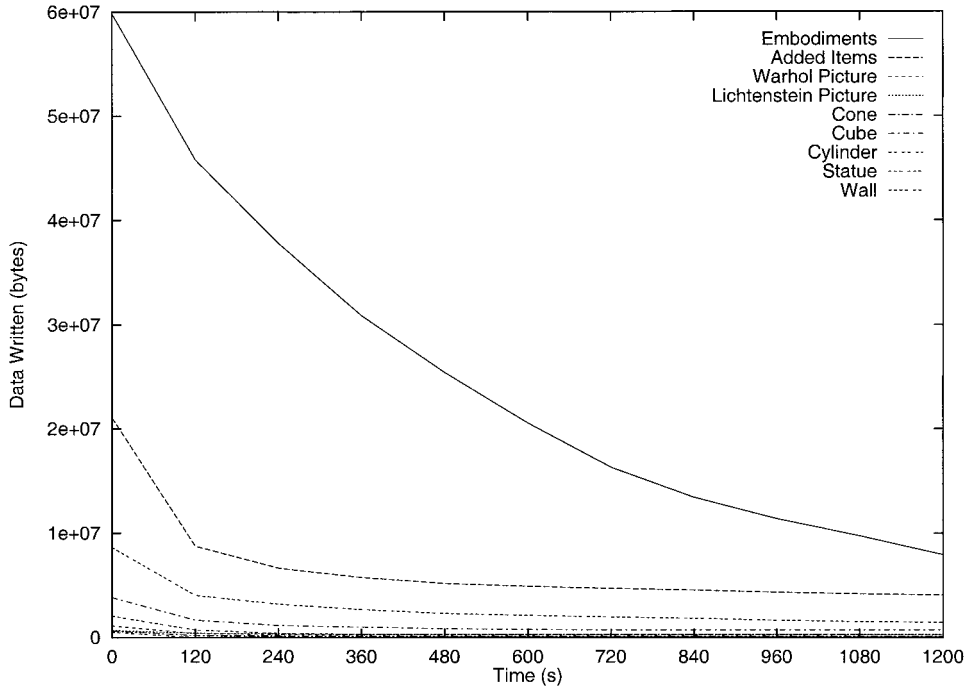

Figure 9. *Persistent data traffic versus time to persistence for the delayed persistence deep behavior.*

to achieve particular effects for data items. Beyond this, we can also exploit the presence of deep behaviors as a more general form of metadata within the shared world state. For example, suppose that each client of a virtual world maintains a cache of items within that virtual world when it was last visited. Without additional information, the client would not be able to prioritize the items to be cached. However, with the addition of deep behavior, the client could use those deep behaviors to inform the selection of items for caching. We have simulated both a least recently used (LRU) cache and a selective least recently used (SLRU) cache that uses deep behavior annotations to cache only persistent data items (which are more likely to persist and typically more important). Figure 10 shows the cache hit rates achieved for the virtual objects in the recordings already described, as a function of the cache size.

Using deep behavior annotations as metadata allows the cache to give consistently better performance. The naïve LRU cache is also caching nonpersistent items, such as the users' avatars, and therefore discarding at least a fraction of the more useful persistent items when the cache size is limited.

In addition to providing superior cache performance for a given cache size, the operation of the selective cache is also more efficient than the LRU cache. Figure 11 shows the number of times the two approaches replaced items for maximum cache sizes ranging from ten to 1,000 items. With a cache of ten items, the LRU cache performs 66,892 writes compared to 10,255 writes performed by the selective cache. This large discrepancy is due to the LRU cache being too small to hold all of the rapidly changing embodiment items in the environment, and so thrashing as items are replaced. The selective approach does not suffer from this problem because it does not cache the un-annotated embodiment items. When the cache size reaches 900 items, the activity of the SLRU cache levels out as it contains all of the annotated items in the environment and so never replaces items in the cache.

Figure 12 shows the number of items stored by the LRU and selective caches for maximum cache sizes of

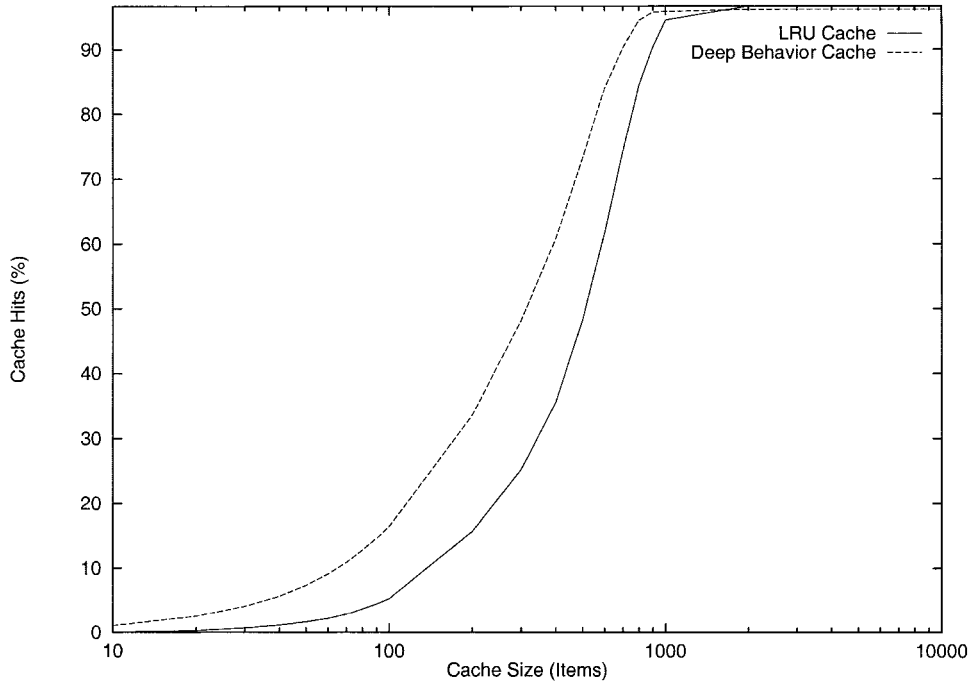

Figure 10. *Cache hit rates versus cache size for LRU and selective (deep behavior) caches.*

between ten and 5,000 items. For maximum cache sizes up to 100 items, both approaches utilize the entire allowed storage space, with LRU using the maximum space up to a maximum cache size of 300 items. With a cache size of 2,000, the selective cache reaches its maximum usage of space and contains 842 items compared to LRU, which contains 1689 items. With the maximum number of items set to 5,000, LRU uses three times as much storage space as the selective approach for a 0.4% advantage in cache hits.

The hints provided by deep behaviors can be used to drive other heuristics, such as reasoning about the likelihood that cached items are still valid to provide an offline view of a virtual world to a disconnected wireless client.

7 Conclusions

Many researchers are working towards the ideal of arbitrarily flexible virtual environment systems. A key problem in engineering "flexibility" into any system is finding a good balance between flexibility per se, and the amount of help that the system can actually provide. Arguably, the most flexible VE system is a C++ (or similar) compiler, whereas a general-purpose component mechanism provides the greatest runtime flexibility. However, these systems do not provide any VR-specific assistance to the developer or would-be user. Once a compiler or component system has been chosen, a framework still needs to be developed to leverage that flexibility.

In our approach (motivated by differentiated treatment of items within a single virtual world), we have chosen to adopt a distributed event filtering framework. We have demonstrated that this approach can realize a broad range of approaches to consistency and persistence in networked virtual environments. The distributed event filter model also has well-defined semantics for adding and managing event filters, and so serves as a basis for event filter composition.

In addition to the low-level mechanism for extensibil-

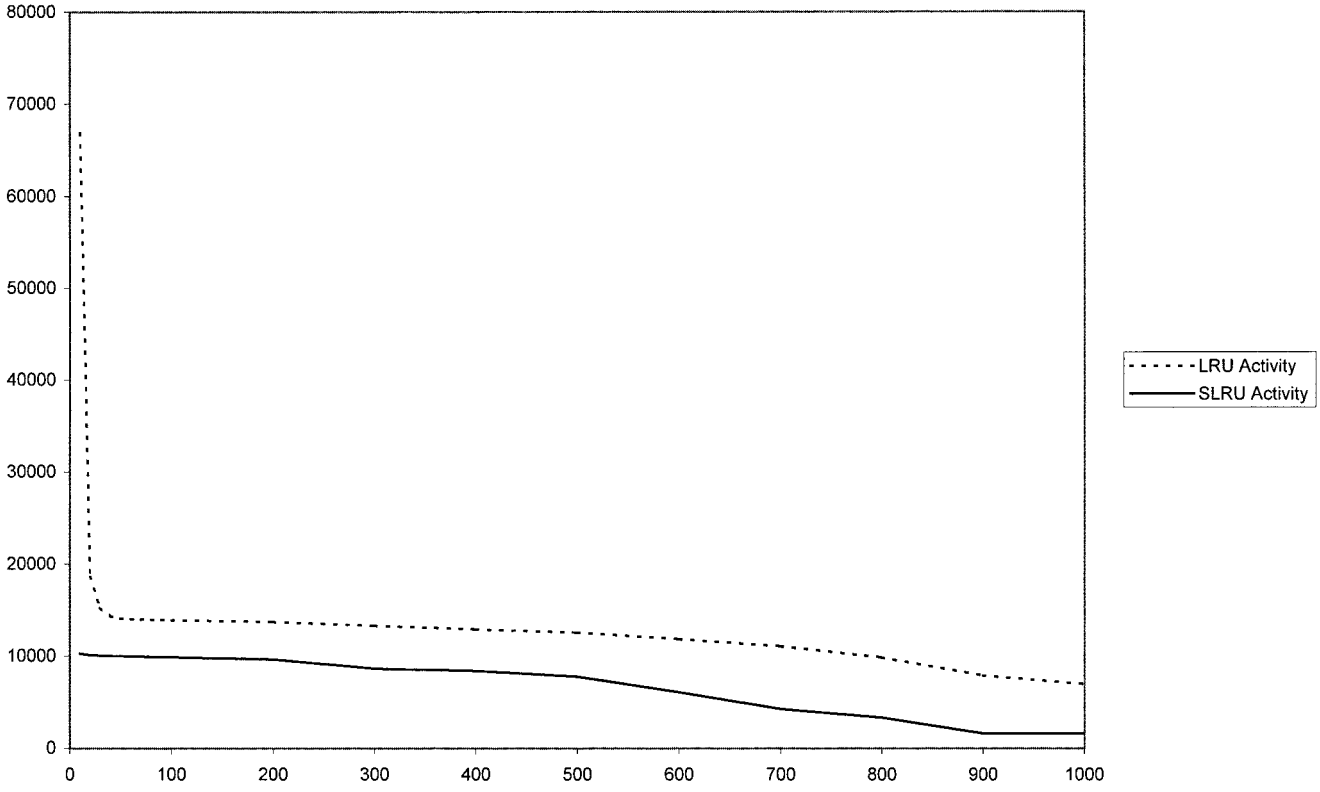

Figure 11. *Cache activity for LRU and SLRU caches with maximum cache sizes from 10 to 1,000 items.*

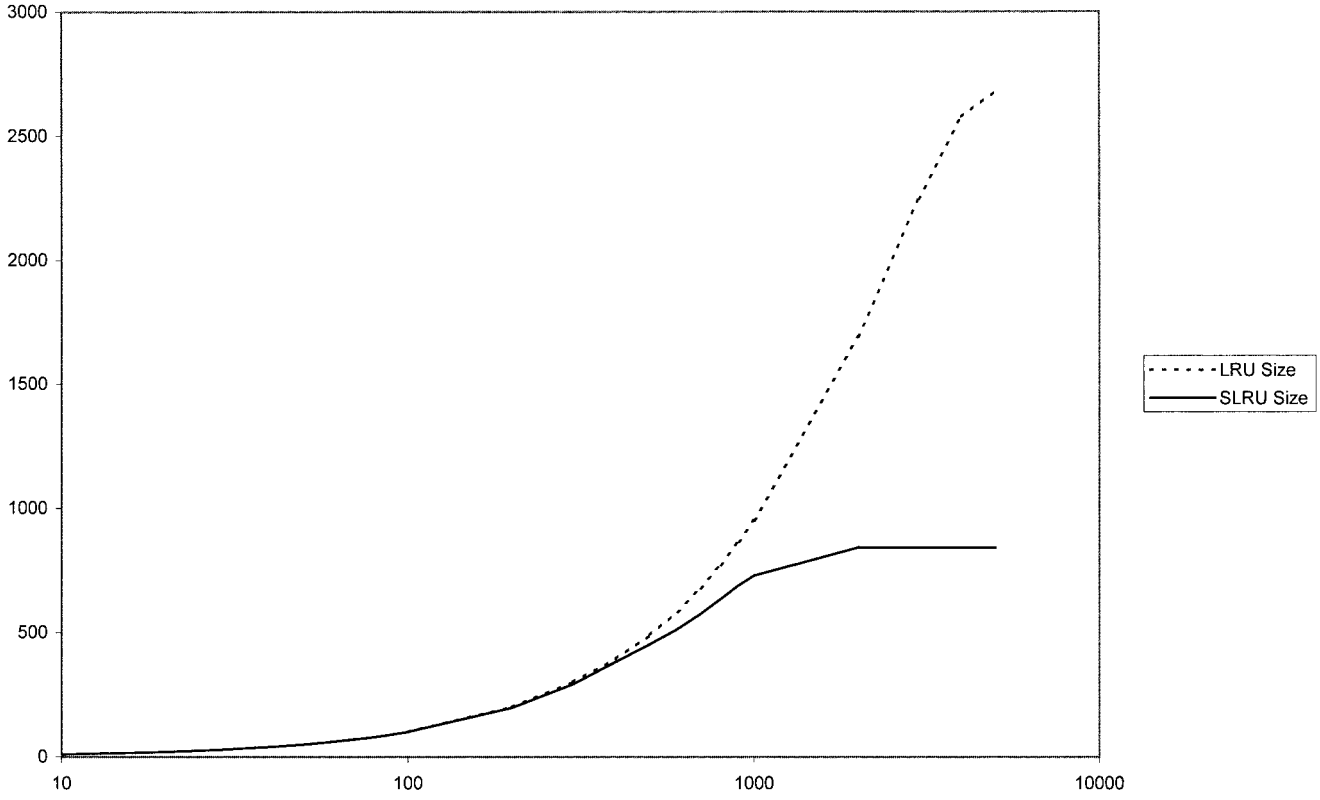

Figure 12. *Utilized cache size for LRU and SLRU caches with maximum cache sizes of ten to 5,000 items.*

ity and runtime modification, we have also found the need—for users, world builders, and system developers—to be able to specify and reason about system behavior at a higher level. Our solution to this is to provide deep behaviors. These allow annotations of the data model (for example, a scene graph) to determine the low-level extensions and modifications that are made to the runtime system, dynamically, and on a per-data-item basis.

We argue that this dual approach provides a good balance of flexibility, extensibility, and manageability. We have also shown that deep behaviors, viewed as a specific form of metadata, can be exploited to further optimize other elements of system behavior such as caching.

Acknowledgments

We gratefully acknowledge funding from UK EPSRC research grant GR/M09223, "The Nature and Utility of Persistence in Virtual Environments."

References

International Standard. *ISO/IEC 14772-1: 1997 Virtual Reality Modeling Language (VRML '97).*

Capps, M., McGregor, D., Brutzman, D., & Zyda, M. (2000). NPSNET-V: A new beginning for dynamically extensible virtual environments. *IEEE Computer Graphics and Applications* (Sep./Oct.), 12–15.

Capps, M., Watsen, K., & Zyda, M. (1999). Cyberspace and mock apple pie: A vision of the future of graphics and VEs. *IEEE Computer Graphics & Applications* (Nov./Dec.), 8–11.

Curtis, P. (2002). LambdaMOO programmer's manual. [Online Available at ftp://ftp.research.att.com/dist/eostrom/MOO/html/ProgrammersManual_toc.html (verified Oct. 3, 2002).

Diefenbach, P. J., Mahesh, P., & Hunt, D. (1998). Building Open Worlds. *Proceedings of the Third Symposium on Virtual Reality Modeling Language.* (Feb.) p. 33–38.

Gosling, J., Joy, B., & Steele, G. (1996). *The Java language specification*. Boston, MA: Addison-Wesley.

Greenhalgh, C., Flintham, M., Purbrick, J., & Benford, S. (2002). Applications of temporal links: Recording and replaying virtual environments. *Proc. IEEE Virtual Reality* 2002, 101–108.

Greenhalgh, C., Purbrick, J., & Snowdon, D. (2000). Inside MASSIVE-3: Flexible support for data consistency and world structuring. *Proc. 3rd International Conference on Collaborative Virtual Environments* (CVE 2000), 119–127.

Greenhalgh, C., Purbrick, J., Benford, S., Craven, M., Drozd, A., & Taylor, I. (2000). "Temporal links: Recording and replaying virtual environments. *Proc. 8th ACM Multimedia Conference* (MM 2000), 67–74.

Lea, R., Honda, Y., Matsuda, K., & Matsuda, S. (1997). Community place: Architecture and performance. *Proc. 2nd Annual Symposium on the Virtual Reality Modelling Language* (VRML '97), 41–50.

Pettifer, S., Cook, J., Marsh, J., & West, A. DEVA3: Architecture for a large scale virtual reality system. *Proc. ACM Symposium in Virtual Reality Software and Technology 2000* (VRST 00), 33–40.

Purbrick, J., & Greenhalgh, C. (2001). Collaborative creation of a persistent virtual world. *Proc Human-Computer Interaction — INTERACT '01,* 35–42.

Sense8 Corp. (1998). *WorldUp users guide*. release 4.

Terry, D. B., Petersen, K., Spreizer, M. J., & Theimer, M. M. (1998). The case for non-transparent replication: Examples from bayou. *Bulletin of the Technical Committee on Data Engineering, 4*(21), 12–20.

Waldo, J. (1999). The Jini architecture for network-centric computing. *Communications of the ACM,* 76–82.

Watsen, K., Zyda, M. (1998). Bamboo—A portable system for dynamically extensible, networked, real-time, virtual environments. *Proceedings of VRAIS '98,* 252–259.

Web3D Consortium. (2002). *X3D: The virtual reality modeling language — International standard ISO/IEC 14772: 200x.* Online. Available at: http://www.web3D.org/TaskGroups/x3d/specification/ (verified Oct. 3, 2002).