LivingSpace: A Living Worlds Implementation using an Event-based Architecture

Rycharde Hawkes, Mike Wray
Extended Enterprise Laboratory
HP Laboratories Bristol
HPL-98-181
October, 1998

E-mail:[rjh,mjw]@hplb.hlp.hp.com

VRML,
living worlds,
distributed virtual environments,
internet notification systems,
computer graphics

We present LivingSpace, an implementation of the Living Worlds Working Group's draft specification for the use of Virtual Reality Modelling Language to support multi-user Virtual Environments. The description of the implementation focuses on the use of the Keryx Internet Notification System to provide the essential infrastructure. The usefulness of the resultant open architecture is demonstrated through an example multiuser world containing a shared slideboard and supporting communication between users using spatial audio.

Users of the world are represented using human-like models supporting animations such as walking and waving.

LivingSpace: A Living Worlds Implementation using an Event-based Architecture

Rycharde Hawkes* [1] & Mike Wray [2]

HP Labs (Bristol), Filton Road, Bristol, BS12 6QZ, UK

Abstract

We present LivingSpace, an implementation of the Living Worlds Working Group's draft specification for the use of Virtual Reality Modelling Language to support multi-user Virtual Environments. The description of the implementation focuses on the use of the Keryx Internet Notification System to provide the essential infrastructure. The usefulness of the resultant open architecture is demonstrated through an example multiuser world containing a shared slideboard and supporting communication between users using spatial audio. Users of the world are represented using human-like models supporting animations such as walking and waving .

Keywords: VRML, Living Worlds, Distributed Virtual Environments, Internet Notification Systems, Computer Graphics

1 Introduction

A simple definition of a Virtual Environment (VE) is a computer-generated simulation: the term usually implying the use of 3D computer graphics in the interface. VEs can in addition be multi-user, supporting multiple interacting users, and distributed, running on several computers connected by a network. It has become common to refer to a VE with both these additional properties as a Distributed Virtual Environment (DVE).

The World-Wide Web (WWW) can be thought of as a virtual space of documents. Although many users may be using it at once they are not aware of each other and cannot usually interact via the Web, so the Web is not multi-user in the sense used in this paper. One of the Web's strongest points is that almost anyone can run a Web server or add a document, and all you need to access a document is its Uniform Resource Locator (URL). You can also link your document into the virtual space using URL hyperlinks. So the Web has distributed construction, but not distributed execution. Other than conceptually, there is no totality of web documents. Even if you to try to list all the accessible URLs, the list changes as you enumerate it.

The Virtual Reality Modelling Language (VRML [1]) treats VEs very much as 3D Web documents: VRML worlds can include content from URLs and contain hyperlinks to other VRML worlds or anything else that can be described by a URL. The challenge for DVEs is to maintain the Web properties of distributed construction and connectivity, while adding multi-user interaction and distributed execution. Successful solutions have the potential to change the Web itself, taking it in the direction of the global cyberspace dreamt of by many [2][3].

Engineering a DVE is not easy. One has to attempt to maintain consistency of state across the DVE, keep latency within bounds making interaction possible, and use the minimum amount of bandwidth. Trade-offs obviously have to be made. A common approach is to sacrifice exact consistency at all times in favour of approximate consistency. This can improve latency and reduce bandwidth. Another common technique is to subdivide the environment into zones, pieces that can be treated more-or-less independently [4].

* Contact author.
[1] E-mail: rjh@hplb.hpl.hp.com
[2] E-mail: mjw@hplb.hpl.hp.com

Although there have been many attempts to develop DVE systems [5], they have not become widely adopted for many reasons. Distributed Interactive Simulation [6] is tailored to specifically to military applications to be of general use; even the US Department of Defense's High-Level Architecture [7] initiative - intended to support broadened goals for modelling and simulation - is still very much military-focused.

DIVE (Distributed Interactive Virtual Environment) is quite a successful system within academic circles for exploring the issues related to collaborative virtual environments [8]. Also worthy of note is Open Community from Mitsubishi Electric Research Laboratories which takes the form of a number of services and Application Programmer's Interfaces (APIs) and uses VRML as a front-end [9]. Considering that VRML is an established standard for 3D content on the Internet and has a large user base, extending it to support multi-user VEs should be a better way of implementing a widely accepted DVE system.

The core VRML standard enables content authors to develop VEs, but does not have the functionality to support DVEs (Open Community does not use VRML to model the multi-user content, just the user interface). This multi-user deficiency has been addressed by a number of companies, e.g. Sony with Community Place Browser/Bureau [10] and Blaxxun Interactive's Community Client/Server [11]. Both of these solutions have defined their own proprietary extensions to VRML using PROTOs, which define scripts that manage state sharing and user control by coordinating with a server. Having augmented the standard VRML nodes with vendor-specific extensions to create a multi-user environment, the user is restricted to using the servers from the same vendor.

It was these proprietary, closed systems that motivated the formation of the VRML Consortium's Living Worlds Working Group. This paper begins with a brief overview of the working group's draft specification for vendor independent multi-user extensions to VRML. The implementation of LivingSpace - a Living Worlds (LW) compliant system - is described, with specific focus on how the application-neutral Keryx Internet Notification System has been used to provide the communications and state-sharing infrastructure. The paper concludes with the description of a multi-user world and how it was constructed using the facilities offered by LW.

2 Living Worlds

The conceptual framework that supports the group's proposed multi-user functionality addresses the common problems of avatar control, state distribution, concurrency control, security, etc. The LW specification describes a number of VRML PROTOs, their function and their relationship with each other. Some of these PROTOs provide the core processes and must be present in a LW world, some deliver optional functionality and others merely have a utility function.

At first glance, LW seems overly complex, but this is mainly due to three things. Firstly, the solution of these problems within a language that does not lend itself well to their implementation. Secondly, great lengths were taken to separate the functional elements from those of the implementation so that authors could write the content without worrying about who had implemented the multi-user technology (MUTech) that would be used to execute the world. Finally, steps were made to protect an authored world from malicious third-party content.

A full LW implementation would be very time consuming to complete and not all of its features would be used most of the time. At the time of writing, the contents of a core LW specification are being finalised which includes the minimum set of functionality needed to author a multi-user world. This paper is restricted to only discussing the contents of this core specification.

2.1 Structure

A LW world is subdivided into one or more *Zones*. A Zone is used to group all the elements of the VRML scene that will be shared: geometry, data, etc. Zones need not be defined spatially, but this is one obvious use for them[3]. Any VRML content not included in a Zone is of no interest to the MUTech. A Zone can contain any number of *SharedObjects*. Each SharedObject represents an entity whose state and behaviour should be shared between clients. In an effort to hinder any malicious content and abstract

[3] As in MERL's Spline system [12].

them from the underlying MUTech, these two entities have been separated into a number of components (Figure 1).

Zone and SharedObject (SO) provide a high-level interface for other non-LW VRML content to use; *PrivateZone* and *PrivateSharedObject* (PSO) contain the body of their respective entities' definitions; finally, *MUTechZone* and *MUTechSharedObject* represent the implementation-specific components. Consequently, the functionality of these lower level components is largely dependent on the LW implementor. When the MUTech initialises the Zones and the SOs in the world it detaches the private components of these entities from the scene graph. Now, any VRML content not authored at the same time as the Zones and the SharedObjects will not be able to get to their sensitive internals and violate the protocol of their use.

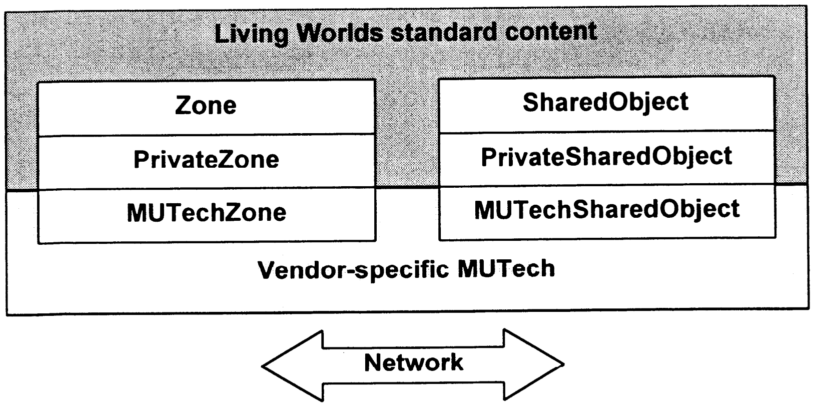

Figure 1. Decomposition of a LW SharedObject and Zone.

2.2 State Management

The state of a SO is stored in the PSO and includes a name, its visual appearance, position, orientation, scale, whether its an avatar, user-defined state, etc. It also exposes the low-level interface needed to manage the object's lock.

Each object in a Zone is replicated on each of the clients with users present in that Zone. One of these objects is nominated the *pilot* and consequently all the other replications are deemed *drones*. The drones, in their simplest form, are merely local surrogates for the object and simply reflect updates received from the pilot. VRML events routed to drones, e.g. the results of remote user interactions with the shared object, are forwarded to the pilot, via the MUTech, for processing. This ensures that the pilot acts as a synchronisation point for all changes to the shared object[4]. Should some other object in the world wish to perform an action on the SO that would cause a state change and requires that no other client may do the same concurrently, then it can acquire the SO's lock, make the change and then relinquish the lock.

The user-defined state mentioned earlier is described using any number of *NetworkState* nodes (Figure 2). A different NetworkState node has been defined for every primitive VRML type, i.e. *NetworkSFBool* (for SFBool), *NetworkMFString* (for MFString), etc., and contains some flags indicating how updates should be treated. When a value is sent to a NetworkState node a number of different things may happen depending on these flags:

1. The state change is forwarded to all the other MUTechs and then, after all deliveries have been completed, the update is reflected locally.
2. The state change is reflected locally immediately and then forwarded to the other MUTechs.
3. As in 2, the state change is reflected locally first, then forwarded to all MUTechs, but when the update has been successfully delivered a further update is sent locally.

[4] More autonomous drone behaviour is possible in LW, but this is outside the scope of discussion in this paper.

Which option to use depends on what compromise the author chooses to strike between latency and synchronisation. Option 1 ensures that all clients end up with the same value[5] but sacrifices latency. Option 2 opts for low latency but lets synchronisation slip and option 3 tries to strike a balance between latency and synchronisation (with a slight emphasis on better latency).

When the remote NetworkState nodes receive the updates, they generate an event which is usually routed to some other piece of VRML content. An optional flag indicates whether updates can only be sent from pilots.

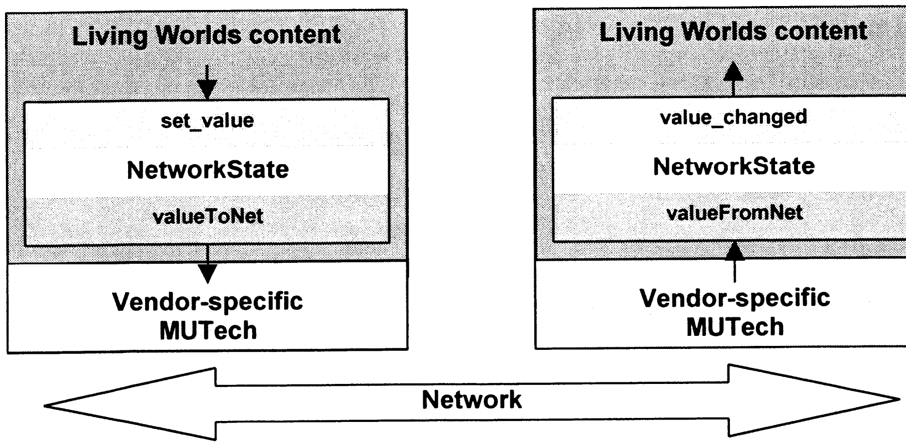

Figure 2. Using NetworkState nodes to share local world state.

2.3 Messaging

SOs can send messages to other SOs on the local client, from the pilot to their remote drones, or from a drone to its corresponding remote pilot. These messages take the form of *Message* nodes and can contain any kind of content that may be encoded using VRML primitive types. LW authors may provide *MessageHandler*s that are used to dispatch incoming messages to authored scripts. These scripts may manipulate the scene and/or return a message to the sender. Each SO has two MessageHandlers: *public* and *private*. The public message handler is used to deal with messages sent by SOs on the local machine, while the private handler dispatches messages sent from Selectors (see below) and received from their respective pilot or drone(s).

2.4 User Interface

Each SO may have a number of pre-programmed behaviours. These can be triggered by NetworkState updates, Messages or through a user-interface. *Selector*s offer the facility to construct menus that may be associated with SOs. Their presence and the functions they offer is up to the author but, typically, a SO presents different interfaces to the pilot and the drones. The pilot selector is displayed automatically on the client where the SO's pilot is created, but the drone selector is displayed only when the user clicks on the SO's drone. These pop-up menus appears offer the service the author wants to make available, e.g. the ability to change the object's colour, perform a simple animation, and so on.

2.5 Avatars

The user's representation within the world is defined by a SO, but there are a number of specific functions built into PrivateSharedObject and PrivateZone to acknowledge that this is a special type of object. LW does not place any restrictions on the definition of an avatar, i.e. its looks, behaviour or structure. Its only requirement is that it responds to orientation and position updates sourced either from the result of the user's interactions with the browser if it is a pilot, or from the updates sent by the MUTech if it is a drone. Everything else is left up to the author who may use NetworkState nodes and Messages to supplement the basic state sharing, and Selectors to augment the basic browser interface.

[5] In a single server system.

3 Core LW Implementation

Our implementation of the draft standard for Core LW is called LivingSpace. It is divided into two parts, a zone client and a zone server. There is a zone client in every VRML browser connected to a world, but there is only one zone server for each zone. The zone clients and the zone server communicate using the Keryx Notification System (KNS), as shown in Figure 4. KNS is an event notification system which connects clients via Event Distributors (EDs). The zone server is an ED plugin, extending its functionality with DVE-specific interfaces. KNS is described further in the following section.

The layers making up a LW client are shown in Figure 3. The top layer represents the standard VRML content used to describe the world. Each client of the world has its own copy of the content. Changes to the content will not be communicated to other clients unless the world contains SharedObjects. SOs are exported to other clients by being made children of the Zones in the world.

We used ECMAScript and Java to implement all the LW PROTOs, including SharedObject and Zone. ECMAScript was used wherever possible, but all interfacing to the client-side of the MUTech had to be done in Java, since it required network access. Conceptually, at least, the MUTech client is the last layer within the VRML browser; KNS is available to any process on the client machine, not just the browser.

3.1 Keryx Notification System

We only have space to give a summary of the application-independent Keryx Notification Service here, so we will limit the description to those features relevant to our DVE work. More detail on KNS can be found in [12], as well as a downloadable Java implementation.

KNS supports anonymous interactions between loosely coupled parties by routing notifications between them, based on the content of the notifications. The producer of a notification does not have to know who the receivers are, or how many there are, and the receiver of a notification does not have to know where it came from. This is in contrast to a distributed object model, which typically assumes direct coupling between interacting parties.

We think of the notification service as a cloud of events. Sources inject events into the cloud, and the notification service takes care of delivering them to clients. The internal structure of the cloud is of no concern to clients. Other event-oriented notification systems include TIBCO's The Information Bus [14] and the recent submission to the Object Management Group's RFP for notification services [15].

A notification can be thought of as signalling the occurrence of some event. The notification contains information about the event in a structured and self-describing format. We often use the terms event and notification interchangeably.

Keryx events are encoded in SDR (Self-Describing Data Representation [16]), a textual syntax for structured data. A Keryx event is an unordered list of field name and value pairs, known as a *map*. Values in the map may themselves be maps, lists or unstructured elements such as numbers, strings and binary data. SDR supports flexible quoting making it straightforward to encapsulate strings containing data with its own syntax, such as VRML, without extensive processing. Counted binary data can also be used. Although KNS is implemented in Java, the simplicity of the SDR format for events makes it a simple matter to send and receive events from other languages. We have used Scheme, C/C++, and Perl as well as Java applets in our research.

Clients of the notification system are decoupled by Event Distributors (ED). Sources send events to an ED which delivers them to clients. An ED also provides services, by recognising some events as directed to its own event interfaces and processing them specially. Other events are treated transparently by an ED. In addition, the basic functionality of an ED may be extended by creating Java classes which implement an extension interface, allowing them to modify the ED's internal event handling and introduce new event interfaces. LivingSpace uses this mechanism to install its Zone server in an ED.

A KNS client requests delivery of the events they are interested in by sending a *subscription* event to an ED. The subscription is intercepted by the ED, which arranges for events to be delivered. A subscription defines the events of interest to a client by including a *filter*, a predicate over events. Only events that pass some subscription's filter will be delivered to a client. Filters can express a wide class of predicates over events. Among other things, filters can test for the presence or absence of fields, check equality of values and test for one list being a prefix of another.

An event source simply connects to an ED and sends events with the content it wishes distributed. The ED delivers the events to all clients having matching subscriptions. An event sender does not need to know who the receivers are, nor does an event receiver need to know who the senders are. As long as an event satisfies a client's subscription it is delivered, and the presence of data not examined by the filters has no effect on delivery. This means that fields can be added to events without disturbing existing clients. This can be useful when evolving an existing system, such as a VE.

Some DVE systems use IP multicast as a group communication mechanism. There are problems with this, such as the low penetration of IP multicast. IP multicast is also a datagram service, and so has no delivery guarantees. Reliable IP multicast is an active research area [17][18]. Wide-area event distribution can be seen as simulated multicast, however, EDs can use multicast itself where appropriate. In some ways, event filtering makes event distribution more powerful than IP multicast, since events will not go to all subscribers, only to those whose filters they pass. Clients can use filters to express directly the set of events they wish to see, regardless of the mapping of those events onto network addresses.

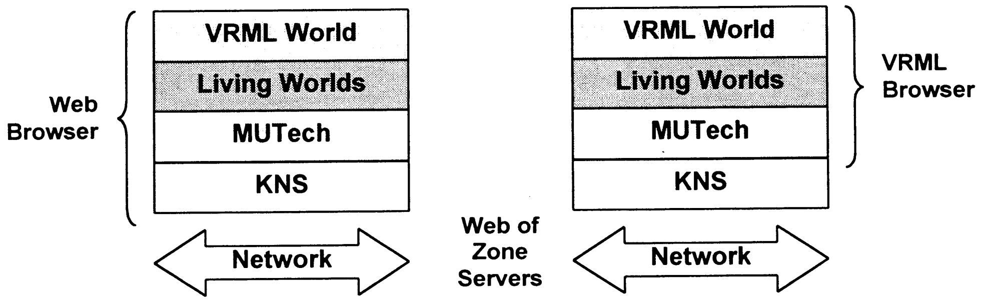

Figure 3. Composition of layers for a Living Worlds client.

3.1.1 DVE Support

We believe that DVEs must be open. By this we mean that they must essentially be defined in terms of the protocols they use, and that there should be no artificial barriers obstructing the implementation of those protocols. It should be possible to build a seamless DVE out of pieces having diverse implementations. We also believe that DVEs should be open in the sense of being able to interwork with external and legacy applications. KNS matches these requirements well, which is why we use it as the transport for our DVE protocols. In addition we find its properties match DVE requirements well. KNS supports group interactions, it has no bias towards graphical data, and scales to many parties.

3.2 Zone Management

In LW, a DVE is constructed of zones. A zone represents a collection of information of interest to participants in the DVE. Each zone has an Event Distributor acting as the coordination point for its events, with the whole DVE being a network of these. Each ED coordinating a zone runs a zone server extension to implement the zone-specific functionality.

A zone server implements a generic state-sharing protocol using events. There are events for initial object creation, differential state update, complete state update, object deletion, operations on objects, and they are also used to manage participation in zones. Events for all zones have the same content type, and are disambiguated by their zone ID field. Clients use their subscription filters to restrict their event flows to the zones they are interested in. The event distribution mechanisms do not interpret the events and place no restrictions on the contents of updates. It is up to the object receiving an event to interpret it, whether the event is a state update or other message.

We break activity in a zone into *sessions*. Initially, there are no clients connected to the zone, and there is no session active. A session starts when the first client connects, and lasts until all clients have disconnected.

3.3 Joining a World

The first action of any LW client is to join an existing Zone (section 2.1) by connecting to the ED nominated in the MUtechZone PROTO and sending a connect event containing the name of the zone. This event is intercepted by the zone server. The zone server maintains a list of active zones and if the authored name provided in the MUTechZone definition is the same, it admits the client to the session. The zone server registers KNS subscriptions on behalf of the client (including relevant filters), so that the client is kept informed of activity in the zone. The zone server then sends the client a report containing the current state of all shared objects and a list of the other connected clients.

The client then begins to create its *initial objects*: those SOs that have been authored as part of the content of the world. Other clients will also create their own copies of these objects and they must be associated so that their state can be kept consistent. Each client creates its initial objects in the same order, numbering them as it goes, and allocating them a unique identifier. The clients communicate the initial object number and unique identifier to the zone server.

By default, the first client to create a given initial object is made its pilot. When another client creates its copy of the same initial object (identified by its number), the zone server sends it an event telling that someone else has already created that object using a different unique ID. The client then changes the object's ID to that sent by the zone server and makes it a drone. This ensures that all clients refer to the initial objects using the same IDs.

The new client then processes the report sent by the zone. This may include objects that have been created since the session began and are not part of the authored LW world. The client creates drones for these new objects which are predominantly (but not limited to) the avatars of other users in the system. The report also includes the current state of the SOs, so that the client can bring everything up-to-date.

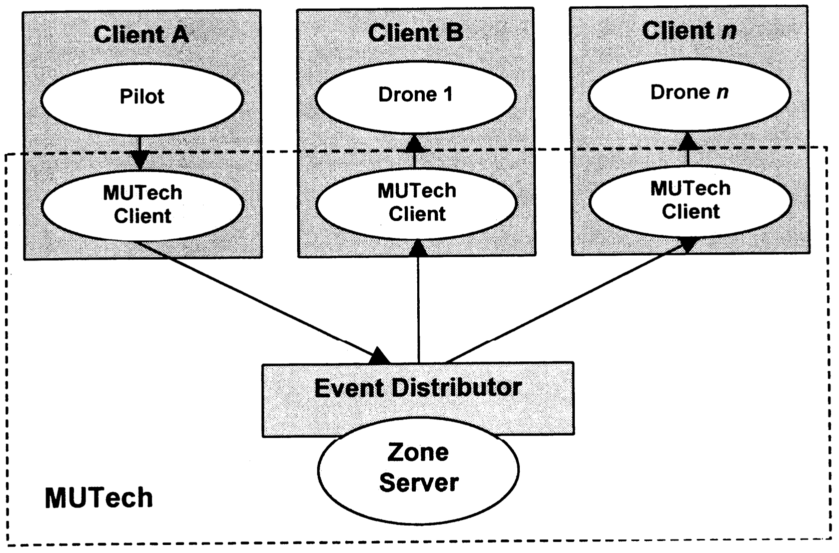

Figure 4. Event paths between three clients for a single object.

3.4 Avatars

During the initialisation process, the user on the client joining the session is presented with a list of avatars to choose from. When the user selects one, a SO for it is created and the relevant plumbing performed to ensure that movements of the user's viewpoint made using the browser's controls are translated into avatar movements. After the zone server has been asked to create the avatar, it notifies all other clients in the zone and they create drones in the normal way. The avatar's owner is then presented with a Selector menu which contains the various avatar-specific functions available to them. These are used to initiate behavioural animations in the avatar's drones: waving, nodding, jumping etc.

The PrivateZone monitors the user's viewpoint, using a ProximitySensor's events to update the position and orientation of the user's avatar. To avoid swamping the network with these updates (emitting from the ProximitySensor at the frame rate of the browser) we use a variation of the dead-reckoning technique used in DIS [6]. Drones predict avatar position and orientation using parameters sent to them. The pilot samples the actual values and compares them with the prediction, sending an update when an error threshold is exceeded. We use velocity, path curvature and angular velocity for prediction, as discussed in [19].

Avatar animations are executed locally by drones, with the avatar pilot controlling which animation is executed. The exception to this rule is the walking animation which is triggered automatically when the user's viewpoint moves.

3.5 Sharing State

The state of SOs is kept consistent across all connected clients. When the pilot for a SO changes its state, by modifying the values in NetworkState nodes for example, it sends a state update event to the ED for the zone. This event is then delivered to the other clients so they can update their local copies. The event is also handled by the zone server, which keeps a record of the current state of all SOs. If a new client connects it will be sent the current state of SOs in the zone report (as discussed in section 3.3).

If a client disconnecting from the zone has any SO pilots running, the zone chooses another client to become pilot and the SO state is preserved. At the moment, when all the clients have disconnected and the session ends, the SO state is lost.

Pilot movements of avatars or autonomous objects, state updates, messages, etc., are relayed by the LW nodes to the MUTech client which generates the appropriate notifications that are routed to the appropriate zone server. The relevant remote MUTech clients are informed and the event is transported back up through the stack and the LW nodes until the end effect on the VRML world is realised (Figure 3).

Figure 4 shows the path of updates from a LW pilot to its drones. In KNS parlance: a drone subscribes to the state update notifications that its pilot publishes. The KNS then ensures that all drones will receive changes in their pilot's state. The pilot in turn subscribes to updates generated by its drones, so that when a local interaction occurs on a remote client the drone can inform the pilot simply by generating an appropriate notification.

3.6 Persistence

The state sharing mechanism described above supports persistence within a session. Providing persistence when all clients disconnect and a session ends would involve transferring ownership to the zone server. It would be straightforward to maintain the current state by saving it when a session ends, and restoring it when a new session starts, though we have not implemented this. The strongest form of persistence involves keeping objects running even when all clients have disconnected. We do not envisage supporting this since it appears to require implementing the full VRML execution model in the zone server, and it is not required by Core LW.

3.7 Locks

Core LW requires support for SO locking. The zone server maintains a lock for each SO, and a queue of lock requestors. Clients can request an object lock by sending an event to the zone server. The server will reply, granting the lock if it is free, or add the requestor to the lock queue if someone else holds it. When a client releases a lock it holds, the lock is granted to the requestor at the front of the queue. The zone server releases all the locks a client holds when it disconnects. Lock events are generated by the MUTech in a client in response to a PSO eventIn, and the replies from the zone server end up generating a PSO eventOut indicating the lock status.

3.8 Zone monitor

Our Java code running in a VRML browser uses events to communicate with the zone server, but other applications can use the same interfaces too. We have implemented a zone monitor as a Java application. This application connects to the ED for a zone and displays the current state of the objects as text. Its user interface allows the user to manage the objects in the zone. This is very useful for administration and debugging.

3.9 Extensibility

The LW specification does not address interoperability of MUTechs, or the communication of VRML objects with external entities. We support external communication by using KNS directly. This is done by making calls to the KNS API from Java running in Script nodes. VRML objects (scripts) sending messages do not actually know whether the receiver is another VRML object or not. So in our architecture the receiver could well be an external application. Conversely, external applications can send messages to VRML objects. An example of this is given in section 4.4.

4 Applications

To put this work into context, this section will describe an example world that we have built which demonstrates a number of the features discussed in this paper. The environment includes a collaborative display, Human Animation (H-Anim) Working Group v1.0-compliant avatars and real-time audio conferencing between users.

4.1 Slideboard

The Slideboard is a combination of a slide projector and a whiteboard (Figure 5). The control panel at the bottom can be used to display the finite number of images that the whiteboard has in its memory. The object may be authored to load a pre-defined set of images upon initialisation, e.g a presentation. The familiar video-like controls enable the user to control this slide-projector-like functionality. Next to the movement controls on the bottom panel are buttons for removing the current slide from memory, acquiring the lock for this object and relinquishing it. The user that acquires the lock can take control of the presentation and has exclusive use over the other features of the whiteboard. The panel on the right hand side provides tools for modifying the current slide, including a colour palette to choose from when drawing on the main panel using the mouse. (As the user drags the mouse around the panel, the Slideboard converts the input into a series of indexed line sets.) The modifications operate on an overlay which may be erased or, if the changes need to be saved, can be combined with the current slide. An overlay exists for every slide in memory. It is also possible to load images from disk and to send the current slide out into the VRML world.

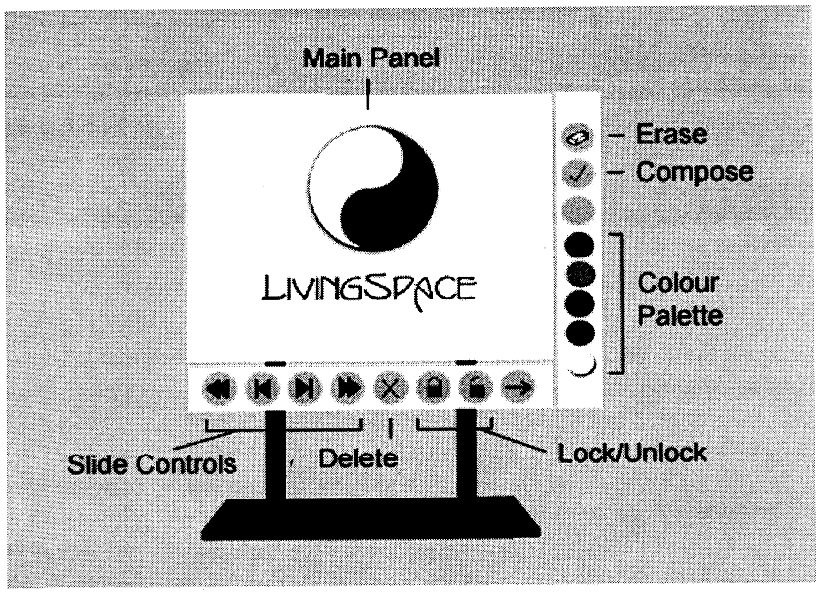

Figure 5. Main elements of the combined slide projector/whiteboard.

How has all of this been achieved? Each image is referenced by URL and the current contents of the Slideboard's memory are held in a couple of NetworkState nodes, an MFString for the URLs and an SFInt32 for the current slide number. As the user moves through the slide show, the value of the NetworkState node holding the current position is modified. This state change is reflected to the remote clients and they subsequently change the current slide being displayed. A similar process occurs when images are deleted or loaded from disk. The Slideboard overlays are composed of coordinates making the index line sets and are also stored in a NetworkMFString node, each SFString holding the details for a single overlay. If an overlay is combined with a slide (using a JPEG library), the resultant image is pushed to a web server, the new URL inserted into the MFString representing the board's memory and the display refreshed. This new set of URLs is, of course, reflected to all other clients.

The result is a collaborative tool that can be used to make slide presentations, annotate them, save the results and load in other images for discussion. A user may join the session at any time and will always see the current state of the whiteboard thanks to the use of the NetworkState nodes and the persistence service offered by the zone server.

4.2 Avatars

The avatars used in our worlds are H-Anim v1.0-compliant. We have provided them with a number of pre-programmed animations for particular gestures, e.g. nod, shake, wave, etc. These are presented to the user in the form of a Selector, as shown in the bottom right of Figure 6. Pressing one of the buttons results in a Message being sent to the user's avatars on the remote clients to trigger the relevant animation. The user's movements through the world are intercepted by the dead-reckoning code built into the MUTech and used to synchronise a walking animation in each drone. The implementation details, issues and problems associated with using these avatars within the confines of LW are dealt with in [19].

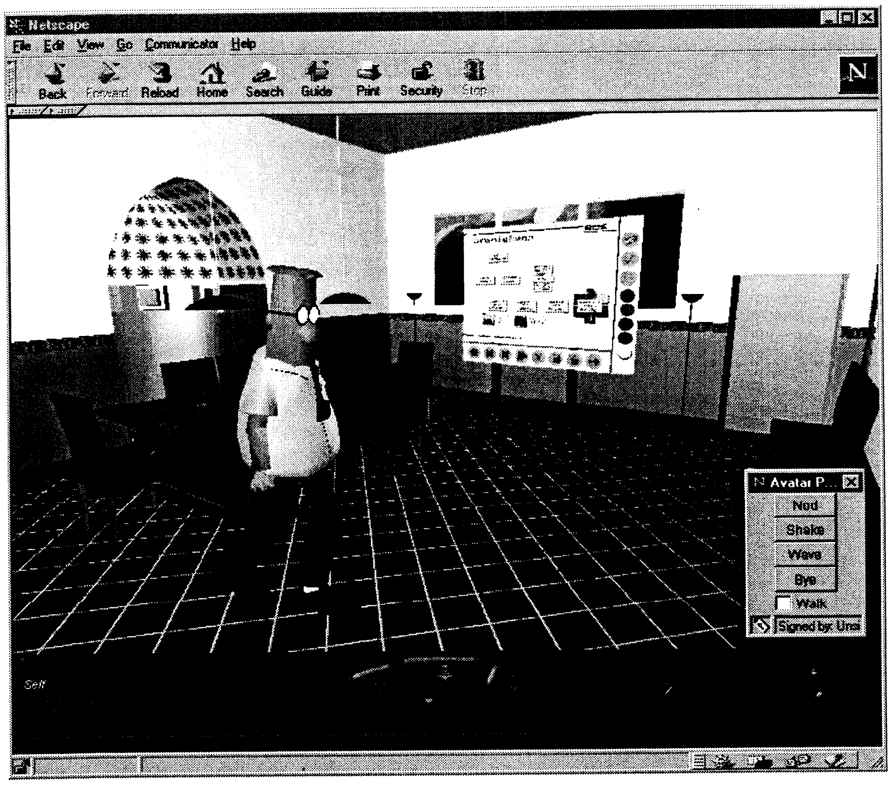
Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Avatar P...
Nod
Shake
Wave
Bye
Walk
Signed by: Unk
Self

Figure 6. Multi-user environment featuring SlideBoard and H-Anim compliant avatar.

4.3 Business Cards

By clicking on the avatars in the environment, e.g. Dilbert in Figure 6, the user can obtain the drone Selector associated with the avatar which may list a number of functions that can be performed in the context of the avatar. A common function is to obtain the virtual business card containing details about the user behind the avatar. When the user clicks the "get business card" button, a Message is sent to the drone's private MessageHandler which executes a script resulting in a Message being sent to the drone's pilot. Having relayed the Message across the network, the MUTech sends it to the pilot's private MessageHandler, which dispatches it to the relevant handler script that constructs a reply containing the user's information and sends it back to the drone. The drone receives this Message via its private MessageHandler and this results in a window displaying the business card.

4.4 Streaming Audio

So far, a user can walk around the world in the safe knowledge that other users can see their avatar walking around (rather than floating) and the user can trigger specific gestures as and when they wish. However, they cannot talk to each other.

TalkSpace [20] is a distributed audio rendering application that creates a shared synthetic 3D audio environment. The voice of each participant in the environment is digitised and streamed over the packet network to each client in real-time using multicast. When a client receives an audio stream, it passes it through a spatial filter to "position and orient" the audio in 3D space. By positioning each voice at a different location it is possible for the user to more clearly identify who is talking and when.

The interface to the TalkSpace client was built using KNS and was based on the model of a VE. For example, each user is assigned an avatar which includes information about location, orientation, and parameters for the spatialisation of their voice, e.g. attenuation, cones of influence, etc. This meant that integration with our LW implementation was straight-forward. When a user joins a LW zone, their client also joins a TalkSpace session and makes the relevant bindings between the LivingSpace and TalkSpace avatars. Now users can walk around the world talking to each other and hear the other users' voices emitting from the same point in space as their avatar is occupying

5 Summary

This paper has given an overview of the Living Worlds Working Group's core specification for multi-user VRML worlds. LivingSpace, an implementation of a system conforming to this specification, has been presented and its utilisation of the application-neutral Keryx Internet Notification System described. To aid the reader's understanding of how LW may be used to construct a useful shared environment, the constituent parts of a demonstration world were outlined. These included a collaborative slideboard and real-time, spatialised, streaming vocal audio between participants. Of particular note was the ease in which LivingSpace could be interfaced with an external system through the use of KNS.

Future work will be centred on the further exploitation of KNS. Even though zones restrict the event traffic somewhat, we want to be able to further restrict traffic using spatial filters. A spatial filter includes a predicate saying that only updates with a position in a given region should be received, for example within a given radius of the client's position. This can greatly reduce update traffic. However updating the region involved in the spatial filter for a moving client could cause a lot of traffic. We envisage applying the general prediction technique to spatial filters, thus reducing filter update traffic.

Since KNS event filters are completely general we can filter events using any suitable criteria. This allows clients to filter out events from objects they are not interested in for any reason, not just their location. Filters give us new ways of grouping traffic based on interest, and in the extreme can give each user their own channel within the wider event flow. This is much more general than the dynamic allocation of multicast groups based on location, as discussed in [21].

We believe that events may also be a good approach to MUTech interoperability. Whatever a MUTech does internally, it only has to deal with our LW event interface in order to interoperate with our implementation. Although LW content is portable among MUTechs, without interoperability a given world has to execute on a single MUTech implementation. Interoperability will be central to scaling LW, and allows distributed construction and execution of large-scale worlds. After all, one cannot assume that a large-scale world executes on a single MUTech implementation.

6 References

[1] VRML. Virtual Reality Modeling Language, International Standard ISO/IEC 14772-1:1997. http://www.vrml.org/Specifications/VRML97/

[2] David Gelernter. Mirror Worlds: Or the Day Software Puts the Universe in a Shoebox... How It Will Happen And What It Will Mean. Oxford University Press, 1991. ISBN 0-19506-812-2.

[3] Myron Krueger. Artifical Reality II. Addison-Wesley, May 1991. ISBN 0-20152-260-8.

[4] J.W. Barrus, R.C. Waters and D.B. Anderson. Locales and Beacons: Efficient and Precise Support for Large Multi-User Virtual Environments. *IEEE Computer Graphics and Applications*, 16(6):50--57, November 1996.

[5] Rycharde Hawkes. A Software Architecture for Modeling and Distributing Virtual Environments. Ph.D. Thesis, University of Edinburgh, 1996. http://www.dcs.ed.ac.uk/~rjh/

[6] DIS. 1278.1 IEEE Standard for Distributed Interactive Simulation--Application Protocols (ANSI). DMSO. DoD High Level Architecture. 1997.

[7] DMSO. DoD High Level Architecture, 1997. http://hla.dmso.mil/

[8] C. Carlsson and O. Hagsand. DIVE - a Multi-User Virtual Reality System. *IEEE VRAIS '93 Conference Proceedings*, pages 394-400.

[9] Mitsubishi Electric Research Lab. Open Community. http://www.meitca.com/opencom/

[10] Sony. Sony Community Place, 1998. http://www.community-place.com/

[11] Blaxxun Interactive. Blaxxun Community Client/Server, 1998. http://www.blaxxun.com/

[12] R.C. Waters, D.B. Anderson, J.W. Barrus, D.C. Brogan, M.A. Casey, S.G. McKeown, T. Nitta, I.B. Sterns and W.S. Yerazunis. Diamond Park and Spline. *Presence: Teleoperators and Virtual Environments*, 6(4): 461-481, August 1997.

[13] Keryx. Keryx Notification System, 1998. http://keryxsoft.hpl.hp.com

[14] TIBCO. TIB/Rendezvous – Publish/Subscribe Middleware for Event-Driven Computing, 1998. http://www.rv.tibco.com

[15] BEA Systems *et al.* Notification Service - joint revised submission, OMG TC Document telecom/98-01-01, 1998. http://www.omg.org/library/schedule/Notification_Service_RFP.htm

[16] Colin Low, Jim Randell and Mike Wray. Self-Describing Data Representation (SDR). Internet draft: draft-low-sdr-00.txt. Work in Progress, 1997.

[17] S. Floyd, V. Jacobson, C. Liu, S. McCanne and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing, *ACM SIGCOMM 95*, pages 342 - 356., 1995.

[18] RMP. Reliable Multicast Protocol, 1998. http://research.ivv.nasa.gov/RMP/

[19] Mike Wray and Vincent Belrose. Avatars in LivingSpace. Submitted to VRML 99.

[20] Colin Low and Laurent Babarit. Distributed 3D audio rendering. *Computer networks and ISDN Systems*, 30: 407-415, 1998.

[21] M.R. Macedonia and M.J. Zyda. A Taxonomy for Networked Virtual Environments. *IEEE Multimedia,* Jan-March: 48-56, 1997.