Avatars in LivingSpace

Mike Wray, Vincent Belrose
Extended Enterprise Laboratory
HP Laboratories Bristol
HPL-98-182
October, 1998

E-mail: [mjw,vinbel]@hplb.hpl.hp.com

virtual humans, avatars, distributed virtual environments, dead reckoning, VRML

We describe how we implemented the avatars we use to represent users in LivingSpace, our implementation of the Living Worlds standard for multi-user distributed VRML worlds. LivingSpace allows multiple users to interact in a shared VRML world and communicate using spatialised audio. Avatars execute walking animations when the user moves and are capable of sitting, waving, and nodding as well as tracking a surface with a hand. Avatar position and orientation is predicted using dead reckoning based on velocity, curvature and angular velocity. We describe the problems we found, and draw some conclusions such as the need for an Inline that preserves the fields of an imported node and more control of VRML worlds over user interaction. We also discuss the use of avatars in multi-user interfaces, such as our VRML conferencing world.

Internal Accession Date Only

Avatars in LivingSpace

Mike Wray*(mjw@hplb.hpl.hp.com)
Vincent Belrose (vinbel@hplb.hpl.hp.com)
Hewlett-Packard Labs (Bristol),
Filton Road, Bristol BS34 8QZ, UK

Abstract

We describe how we implemented the avatars we use to represent users in LivingSpace, our implementation of the Living Worlds standard for multi-user distributed VRML worlds. LivingSpace allows multiple users to interact in a shared VRML world and communicate using spatialised audio. Avatars execute walking animations when the user moves and are capable of sitting, waving, and nodding as well as tracking a surface with a hand. Avatar position and orientation is predicted using dead-reckoning based on velocity, curvature and angular velocity. We describe the problems we found, and draw some conclusions such as the need for an Inline that preserves the fields of an imported node and more control of VRML worlds over user interaction. We also discuss the use of avatars in multi-user interfaces, such as our VRML conferencing world.

Keywords: Virtual humans, avatars, distributed virtual environments, dead reckoning, VRML.

1 Introduction

The aim of this work was to implement human-like avatars in distributed multi-user VRML worlds. The specific system we used was LivingSpace[18, 8], our implementation of the Living Worlds (LW) specification for multi-user distributed virtual environments (DVEs) using VRML [17, 10].

The problem of creating virtual humans in a DVE has three main parts: modelling the geometry of the human, creating the behaviours, and distributed coordination of the behaviours. We will not go into modelling here, one useful text is [14]. An overview of the whole problem can be found in [2, 16], some more recent work on the Jack system is [1].

Defining an avatar behaviour includes animating the avatar to execute motion. Methods for animating virtual characters can be roughly divided into approaches based on motion capture and those based on simulation. Motion capture uses data recorded from actual motion, and can produce extremely realistic and subtle animation. Motion capture is typically done off-line, with the recorded data replayed at runtime. The drawbacks of motion capture include the relatively high cost of capturing the data, the lack of flexibility of pre-recording, and the fact that the motion must be performable by someone.

Simulation relies on computation of a mathematical model to generate motion [9]. This can be done off-line and the motion data recorded, or on-line if the computational load is not excessive. Simulation can generate precisely controlled motion, with interactive control if done on-line, but it tends to look mechanical. It is relatively easy to adapt a simulation to variations in animation parameters, such as limb length or walking speed. Data from motion capture is much more difficult to adapt, though it can be done [6].

We needed interactive control of the avatar animations, and we wanted to be able to use the same animations for avatars with varying geometry. This lead us to base our approach on simulation rather than motion capture. Our initial requirements only included a simple set of animations which we felt we could simulate in real time.

Animation is not all there is to it though. Even if you have animations for the actions you need you still need to control when the avatar executes them, and how they are combined. If you have separate animations for walking and waving you need to define how to do both at the same time. You also need to define the control logic of the avatar.

Having constructed your avatar you have to make it work in a distributed multi-user virtual environment. The problems to be solved include keeping animation smooth in networked clients without using too much bandwidth, and coordinating which animations to run. We use an extension of the dead-reckoning technique used in systems such as DIS [4] to reduce the bandwidth consumed by position and orientation updates. Dead-reckoning using Kalman filtering to predict the joint angles of a moving avatar is reported in [3]. We also communicate the high-level avatar state and execute animations locally.

*Corresponding author

We discuss the details of our approach to all these aspects in the following sections.

2 VRML and Avatars

We use the Hanim 1.0 standard [5] to describe the geometry of our avatars. This standard was produced by Human Animation Working Group of the VRML Consortium and defines a set of PROTOs to be used to represent a humanoid in VRML. The top-level is the Humanoid PROTO, which contains lists of Joint and Segment PROTOs representing the joints and body parts of the humanoid. The Hanim standard defines a joint hierarchy and gives the joints standard names by which they can be referenced. Sending set_rotation events to individual joints allows changing the posture of the body, while sending set_rotation and set_translation events to the HumanoidRoot joint allows positioning the avatar in the world.

You can create behaviour for the humanoid by routing into it, but this requires the animation and the humanoid be in the same VRML file. You can import a humanoid into another file using Inline, but then you have no access to its joints. We wanted to separate avatar behaviour from geometry to make it easy to use the same behaviours for different geometries, so this was a problem for us. We solved it by creating an Avatar PROTO having a url field to indicate which avatar to load. The script implementing Avatar loads the VRML file from the URL, and takes the first Humanoid node it finds there to be the avatar definition. Having got the Humanoid node the script can access all the external fields and make them accessible to its behaviours.

Our Avatar PROTO has our implementation of behaviour built-in, but we felt the loading capability would be generally useful, so we have also implemented a HumanoidImport EXTERNPROTO having all the eventOuts from Humanoid, plus fields for loading a URL and handling errors. A HumanoidImport loads the VRML file given in its url and fill in all the eventOuts from the first Humanoid it finds in the file. Using HumanoidImport lets us use the Humanoid interface while still being able to easily change which avatar geometry we are using.

This problem of abstracting by using a URL without losing access to node fields is common in VRML. One can construct an XImport PROTO for each node type X needed, but this is tedious, and we feel it would be worth considering adding a facility to VRML to import from a URL while preserving node fields.

The Hanim specification standardises the joint and segment names of humanoids, but it does not specify which ones have to be present. Luckily all our avatars had the basic joints such as hip, knees, elbows *etc.*, but we found considerable variation in the joints of the back. We therefore had to make our avatar animation code look for any one of several possibilities to use.

One critical parameter we needed was the eyeHeight - the distance from the avatar's eyes to the floor. This is not supplied in Hanim 1.0, so we had to find a workaround. The first problem was that we did not know the eye height for our avatars. We created a test world that imported a given avatar, allowed the user to move a marker (a sphere) until it was in front of the eyes, and printed the eyeHeight. We then edited a string of the form `"eyeHeight=x"` giving the value in metres into the documentation strings of the avatar's Humanoid node. Our avatar script extracts this when the avatar is loaded.

The Java control of the avatar's animation is built in 5 layers, which we now describe:

- **Layer 0**: the interface between Java and VRML. It is based on a Script node which has directOutput enabled and a field referring to the avatar body, an H-Anim Humanoid node. A Joint class reflects the Joint PROTO structure in Java.
- **Layer 1**: A set of Java classes implementing the architecture of an H-Anim 1.0 avatar. Further structure is added with classes handling the limbs (legs and arms). These classes use layer 0 to build an internal representation of the avatar from the data contained in the VRML file. As most avatars will not implement all the joints, the tree structure representing the body is simplified to retain only the meaningful information. The AvatarJoint class stores VRML information for each joint of the body as well as the tree structure. It allows access to the parent and all the children of a given joint. Further structure will probably be added in the future for better handling of the head and hands. Up to now, our avatars have not been detailed enough to need this.
- **Layer 2**: A set of Java classes using closed formulas for inverse kinematics on the avatar limbs. A range of useful positions is used: given some input parameters, we compute the missing parameters to achieve the desired position.
- **Layer 3**: A set of Java classes providing a set of motions for the avatar, based on the layer 2 IK formulas.
- **Layer 4**: User control over the avatar using a menu.

2.1 Exact inverse kinematics

The avatars we used did not implement the complete structure of the Hanim 1.0 specification, so we had to use simplified models for some body parts, mainly the

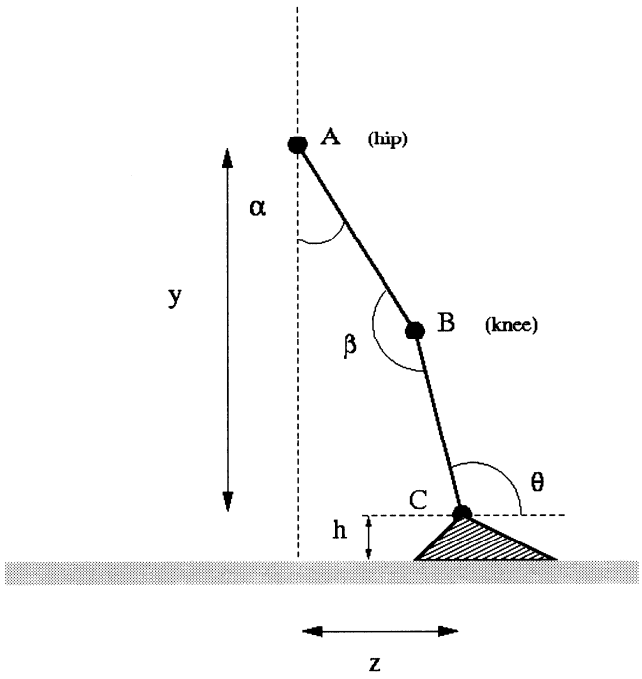

Figure 1: Leg model

hands, feet and the head. These relatively simple models allowed us to compute closed formulas for the limb positions during the motion.

Consider the model shown in figure 1. We want to compute the hip, knee and ankle rotations necessary to achieve the leg position defined by the input data z and y. We can solve for α, β, θ:

$$\begin{aligned} AC &= (z^2 + y^2)^{1/2} \\ \alpha &= \arctan(z/y) + \arccos\left(\frac{AB^2 + AC^2 - BC^2}{2\,AB\,AC}\right) \\ \beta &= \arccos\left(\frac{AB^2 + BC^2 - AC^2}{2\,AB\,BC}\right) \\ \theta &= \alpha + \beta - \frac{\pi}{2} \end{aligned}$$

Then the required hip, knee and ankle angles are $\alpha, \beta - \pi, \frac{\pi}{2} - \theta$ respectively.

2.2 Approximate motion

Although every human has his own way of walking, everybody has the same idea of what a walking motion should look like. When simulating walking with dynamics we tend to try to fit the motion to physics, when we only need to fit it to human perception. A walking motion is not good when it is physically correct, but when humans see it as a correct walking motion. Sources on human motion include the seminal photographs of Muybridge [12], and human kinematics texts such as [19].

Our method is based on this: human motions such as running, walking and kneeling, which are not too complex, can be decomposed into phases where it can be simply described. Using the exact IK formulas, we can then position our avatar during the different phases.

Imagine we want our avatar to turn θ degrees to the right. We can decompose the motion into 2 phases:

- phase 1: the avatar lifts its right foot, turns its body $\theta/2$ degrees to the right and puts its right foot in its final position. During this phase the left foot remains at the same position on the floor.
- phase 2: keeping the right foot fixed on the floor, the avatar lifts its left foot and moves it to the final position, turning its body $\theta/2$ degrees to the right.

Using this approach, we can produce real time motions which fit each avatar and allow interactions with the environment, such as positioning the hand on objects, or shaking hands with another avatar. Moreover, most motions can be parameterized; walking has a speed parameter for example. In the future we could also add different styles for motions, such as sitting, which may be performed in very different ways by different humans.

The main drawback of this method is that someone has to decompose the motion into steps, as might be done when animating a cartoon character. The realism of the motion depends directly on the skills of this "animator". Nevertheless, compared to pure physical simulation, this

allows "elegance" which otherwise would be very hard to code into physical equations.

2.3 Interacting with objects

We would like our avatars to interact with the world they are in, doing things like sitting in chairs, walking up stairs and even shaking hands. In a normal VRML world objects are just Shapes containing IndexedFaceSets, and there is nothing to distinguish a chair from the floor, a desk or anything else.

In everyday life, we all know a door is made to be opened or closed, a button to be pushed and a table to put other objects on. In our VRML world, we need to provide this information to our avatars. This is done by encapsulating each kind of object in a PROTO. For instance, there is a Chair PROTO, indicating that the object is actually a chair (which tells the avatar how they should use it) and describing the basic information needed to use a chair:

```
PROTO Chair [
     exposedField SFString   name           "Chair"
     exposedField MFString   info           "v1.0"
     exposedField SFBool     has_l_arm      TRUE
     exposedField SFBool     has_r_arm      TRUE
     exposedField SFVec3f    l_armCenter    0 0 0
     exposedField SFVec3f    l_armBBox      -1 -1 -1
     exposedField SFVec3f    r_armCenter    0 0 0
     exposedField SFVec3f    r_armBBox      -1 -1 -1
     exposedField SFVec3f    supportCenter  0 0 0
     exposedField SFVec3f    supportBBox    -1 -1 -1
     exposedField SFVec3f    backCenter     0 0 0
     exposedField SFVec3f    backBBox       -1 -1 -1
     exposedField SFVec3f    translation    0 0 0
     exposedField SFRotation orientation    0 0 1 0
     exposedField SFVec3f    scale          1 1 1
     exposedField MFNode     children       []
     ] {}
```

This PROTO reflects the fundamental structure of a chair, as needed in our virtual world: a chair has a seat, a back, and usually two arms. It also includes bounding boxes for its components (seat, back and arms). An avatar can then adapt its sitting motion to the information provided by the Chair PROTO, whatever the chair geometry, as illustrated in figure 2.

Constructing a PROTO for each object type allows an object's attributes to be recovered from its node, but an avatar still has the problem of finding the node in the first place. VRML does not allow code to arbitrarily tour the scene graph, so objects need to be put in a place the avatar can access. LivingWorlds uses a Zone node as a container for shared objects, so this could also be used to find objects. However, the problem of efficiently discovering the objects of interest among all the objects present remains.

This is a general problem, and our view is that VRML needs to move towards supporting objects, properties and affordances instead of geometry and the scene graph.

2.4 Touching objects

To reflect the user's intention, we want our avatar to be able to touch objects in the world when the user clicks on them. We want this capability to be independent of the objects' specific structures. If we attach a TouchSensor to an object we can retrieve the following information: which object was clicked on, where, and the surface normal at that point. Using the IK formulas, we can then position the avatar's hand on the object if it is reachable, or otherwise just point at it. We use the surface normal to orient the hand correctly with respect to the surface of the object.

3 Avatars in a DVE

We run distributed multi-user VRML worlds using our LivingSpace system, an implementation of the Living Worlds standard. Users loading the VRML file for a world running on LivingSpace are allowed to choose the avatar used to represent them to other users of the same world. The URL of the user's avatar is communicated to the other clients of the world so they can load it and move it around as the user moves. Users have a first-person view, so they cannot see their own avatars, only other people's.

This is not the place to go into the details of Living Worlds, but we do need a little background. Living Worlds (LW) defines a number of PROTOs for VRML nodes that are used to interface with an external application implementing the logic to handle sharing the state of shared objects in the VRML world. LW does not define how this is implemented, instead it assumes the existence of an implementation, called multi-user technology (MUtech), satisfying certain semantic constraints.

Each client connected to a LW world has its own copy of the world, and it is the information distributed by the MUtech that coordinates these copies to create the illusion of a shared world. We will not go into detail about LW and the MUtech here, suffice it to say that, among other things, the MUtech is responsible for monitoring changes in a user's viewpoint, using a ProximitySensor, and communicating its position and orientation to other clients of a world. The viewpoint information communicated by the MUtech is used to move the user's avatar around the world as their viewpoint changes.

In LW a shared object such as an avatar has replicas running in several browsers. The controlling replica is known as the *pilot* and the others are known as *drones*. Our avatar implementation contains pilot code sampling

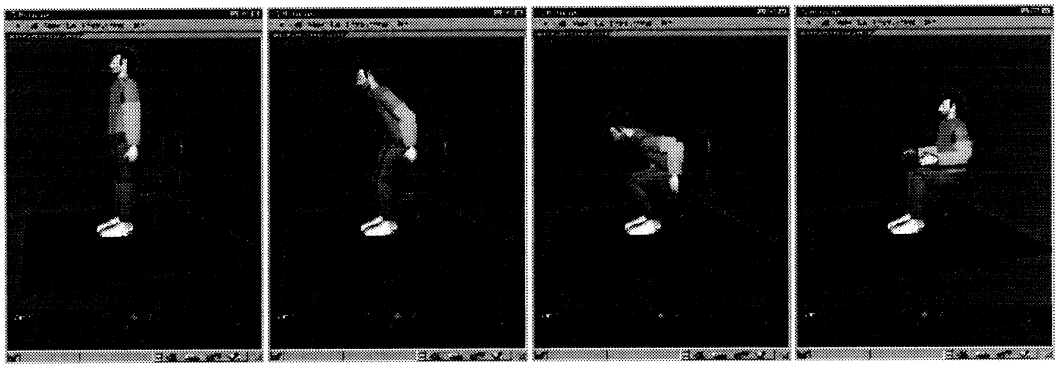

Figure 2: Sitting animation

the viewpoint and drone code predicting the viewpoint based on parameters sent by the pilot.

3.1 Dead-reckoning

When a user moves, the ProximitySensor supplies events at a high rate, typically the frame rate of the user's browser. Simply forwarding the events to other users at this rate would overwhelm the network. One could imagine forwarding events at some lower rate, but this would make the motion of avatars jerky in other clients' browsers. To avoid this we use dead-reckoning. Remote avatars predict the current position and orientation based on the last information supplied to them, updating their predictions in response to events from a TimeSensor. The local avatar performs the same prediction and compares it with the actual values, sending an update when the difference exceeds a threshold. Remote avatars use new parameters immediately when they get them. This can make them 'jump' if the error was large, but at least they are up-to-date. It would be possible to blend from old to new values, but we have not implemented this.

A ProximitySensor only supplies position and orientation information, and for prediction we need rate-of-change information as well. We get velocity by numerical differentiation. The velocity can be used to predict straight-line motion, but it does not predict motion along a curved path well. Since curved motion is fairly common, we also extract the angular velocity of the motion and use this in prediction. This allows us to predict motion along a circle as well as a straight line.

If the user's motion is like that of a tracked vehicle, the angular velocity derived from the user's path and the rate of change of the orientation are the same. However the two angular velocities can be different: when the user is turning on the spot, or moving from one viewpoint to another under control of the browser for example. We therefore compute the rate of change of the orientation and predict the orientation independently.

Dead-reckoning immediately causes a difficulty: we use a ProximitySensor to get position information, but ProximitySensors only generate events while the user is moving, they do not generate events when the user stops. If we do not detect stopping the remote avatars will continue moving. We therefore have to sample position and orientation from a separate thread so we can detect stopping. To improve responsiveness we give stopping a high priority, so that an update will normally be sent out immediately when the user stops moving. We do the same when motion starts too.

3.2 Dead-reckoning derivations

Suppose the user's position is described a curve $\mathbf{f}$ in R^3. We want to construct a local approximation to $\mathbf{f}$ that drones can use to predict the position. We first review some differential geometry [15]. Suppose $\mathbf{f}$ has arc-length function s. Then $s(v) = \int_0^v |\mathbf{f}'(u)|du$. We say that $\mathbf{f}$ is *parameterised by arc-length* when $s(u) = u$, *i.e.* when $|\mathbf{f}'(u)| = 1$. We can write $\mathbf{f}(u) = \mathbf{g}(s(u))$ where $\mathbf{g}$ is parameterised by arc length. We define functions $\mathbf{t}, \mathbf{n}, \mathbf{b}$ whose values are mutually orthogonal unit vectors, and real-valued functions κ, τ by

$$\begin{aligned} \mathbf{g}'(u) &= \mathbf{t}(u) \\ \mathbf{t}'(u) &= \kappa(u)\mathbf{n}(u) \\ \mathbf{b}(u) &= \mathbf{t}(u) \times \mathbf{n}(u) \\ \mathbf{b}'(u) &= -\tau(u)\mathbf{n}(u) \end{aligned}$$

We have $\mathbf{n}'(u) = \tau(u)\mathbf{b}(u) - \kappa(u)\mathbf{t}(u)$. These are the well-known Serret-Frenet formulae. Putting $v(u) = s'(u)$ and suppressing parameters for brevity, we have

$$\begin{aligned} \mathbf{f}' &= v\mathbf{t} \\ \mathbf{f}'' &= v'\mathbf{t} + \kappa v^2\mathbf{n} \\ \mathbf{f}''' &= (v'' - v^3\kappa^2)\mathbf{t} + v(3v'\kappa + v\kappa')\mathbf{n} + v^3\kappa\tau\mathbf{b} \end{aligned}$$

So that

$$\kappa = \frac{|\mathbf{f}' \times \mathbf{f}''|}{|\mathbf{f}'|^3}$$
$$\tau = \frac{\mathbf{f}' \times \mathbf{f}'' \cdot \mathbf{f}'''}{|\mathbf{f}' \times \mathbf{f}''|^2}$$

These are the *curvature* and *torsion* of the curve respectively. The reciprocal of the curvature $r = 1/\kappa$ is the *radius of curvature* of the curve, and the circle radius r with centre $\mathbf{c} = \mathbf{f} - r\mathbf{n}$, axis $\mathbf{b}$ is the *osculating circle*. It is fairly easy to see that

$$r\mathbf{n} = \frac{(\mathbf{f}' \times \mathbf{f}'') \times \mathbf{f}''}{|\mathbf{f}' \times \mathbf{f}''|^2}$$

Typical viewpoint motions are planar, and so have zero torsion. If the curvature is non-zero we can approximate the curve $\mathbf{f}$ near u by a curve $\mathbf{h}$ using motion along the osculating circle of angular velocity

$$\boldsymbol{\omega}_1 = v\kappa\mathbf{b} = \frac{\mathbf{f}' \times \mathbf{f}''}{|\mathbf{f}'|^2}$$

So that

$$\mathbf{h}(x) = \exp(\boldsymbol{\Omega}_1(x-u))(\mathbf{f}(u) - \mathbf{c}(u)) + \mathbf{c}(u)$$

where $\boldsymbol{\Omega}_1$ is the skew-symmetric matrix related to $\boldsymbol{\omega}_1$ by $\boldsymbol{\Omega}_1\mathbf{p} = \boldsymbol{\omega}_1 \times \mathbf{p}$, and exp is matrix exponential [7]. In this case $\exp(\boldsymbol{\Omega}_1(x-u))$ corresponds to rotation about $\boldsymbol{\omega}_1/|\boldsymbol{\omega}_1|$ by angle $|\boldsymbol{\omega}_1|(x-u)$.

When the curvature is zero we use the obvious linear approximation:

$$\mathbf{h}(x) = \mathbf{f}(u) + (x-u)\mathbf{f}'(u)$$

We now turn to orientation. We represent orientation using quaternions [13]. Suppose the orientation at time u is given by $\mathbf{q}(u)$. The change in orientation from $u-t$ to u is $\Delta\mathbf{q} = \mathbf{q}(u)\mathbf{q}\hat{}(u-t)$. If $\Delta\mathbf{q}$ has axis $\mathbf{n}$ and angle θ this corresponds to an average angular velocity

$$\boldsymbol{\omega}_2 = \frac{\theta}{t}\mathbf{n}$$

We approximate $\mathbf{q}$ near u by a function $\mathbf{p}$ using its value at u and the angular velocity:

$$\mathbf{p}(x) = \exp(\boldsymbol{\omega}_2(x-u))\mathbf{q}(u)$$

One small point. The axis and angle of a rotation are not uniquely defined. Changing the sign of the axis and angle does not change the rotation for example. More seriously rotations about $\mathbf{n}$ by θ and about $-\mathbf{n}$ by $2\pi - \theta$ are the same. When computing $\boldsymbol{\omega}_2$ it is important to use the alternative with the smaller angle, otherwise the angular velocity may be wildly overestimated.

In our dead-reckoning code we numerically differentiate the position samples we get from the ProximitySensor to get $\mathbf{f}'$ and $\mathbf{f}''$ and use the expressions above for $\boldsymbol{\omega}_1$, $\mathbf{r}$ and $\mathbf{h}$ to predict position. This enables us to predict motion in a straight line or along a circle. We predict orientation independently using $\boldsymbol{\omega}_2$ and $\mathbf{p}$. We force updates once very few seconds, and are able to predict constant turn-rate motion for about 10 seconds. The update rate when the user is moving the viewpoint is typically about 4 per second.

4 Animating DVE avatars

We use human-like avatars, and we want them to move in a human-like way, in particular we want remote avatars to execute a walking animation when the user is moving. The avatar pilot code detects motion and sets the avatar motion state to moving. This is sent to the drones by the MUtech and triggers the walking animation. If the user selects a modifier such as waving this is sent separately and the drones execute a combination of the animations. When the pilot detects the user has stopped moving it sends an update setting the state to stopped, causing the drones to stop the walking animation. However they continue the waving animation (unless it has completed).

The walking motion works well when the user simply moves around (at the correct eye-height) or looks from side-to-side, but if the user looks up or down his whole avatar tilts up or down. This makes the avatar's feet come off the floor. We would like turn the avatar's head instead, but this is difficult because of the way the orientation information is communicated in LivingWorlds.

The code implementing the walking animation is parameterised by velocity, and capable of adjusting steplength and frequency to match the avatar velocity. The animation takes parameters such as the avatar limb lengths into account and tries to prevent the support foot from sliding on the ground. At the time of writing the LivingWorlds architecture does not support communicating velocity to drones, so we have implemented an extension to support this.

There are some difficulties in dead-reckoning and animating avatars. We are trying to reconstruct the user's motion from low-level information about their position. We have to differentiate this numerically to get velocity and rate of turn. The browser's GUI knows these parameters already, but we cannot access them.

Also the user can move at any speed and stop instantaneously, things it is hard to make an avatar do convincingly. It might be worth considering changing the VRML input model to allow more control, so that a VRML world could define what kind of motion happends in response to mouse motion. We would then be able to drive user motion using a direct avatar model, determining veloc-

ity from the user's input rather than numerically. This should improve the behaviour of dead-reckoning by removing a major source of numerical "noise". We could also provide a more direct avatar control model, restricting speeds to a range avatars can walk or run at, and providing control over head and body motion independently.

5 Multi-user interaction

When multiple users are present in a LivingSpace world they can see each other, represented by their avatars. They can also talk to each other, as we have integrated a 3d-audio conferencing tool called TalkSpace [11]. TalkSpace handles packetising audio and distributing it over the network. TalkSpace spatialises the sound from the participants, with locations controlled by LivingSpace. This makes each user's speech appear to come from their location.

The Living Worlds architecture does more than support avatars, it includes *SharedObjects* having state that is kept consistent across connected clients. We have created a VRML world suitable for conferencing in which we have used shared objects to implement a shared slideboard in VRML. The slideboard shows slides which the users can change or draw on, with the results made visible to all users.

However, users interact with the slideboard using the mouse to activate TouchSensors. Although you can see the slide change on the slideboard because someone has clicked the 'advance' button, you cannot see who it was. Since TouchSensors have unlimited range the user who clicked a button may not even be anywhere near it from your point of view. This can be confusing in an environment intended for collaboration. We plan to investigate ways of making the user's avatar reflect the user's interaction with the world. One approach we are considering is to limit the range of sensors on shared objects, and to animate the user's avatar when the user clicks on a sensor. We could make the user's avatar press the button if it is close enough, or point at the button with a 'laser beam' if further away.

6 Summary

We began with an introduction to the problem of implementing avatars in a distributed multi-user environment. We then described our approach to their implementation.

Avatar geometry is represented using the Hanim 1.0 Humanoid, and imported from an Avatar node. The Avatar node uses Java classes internally to model the structure of the avatar. Avatar poses are computed using closed-form inverse kinematics, with the pose parameters under program control.

We enable avatars to interact with objects in the environment by using object PROTOs, such as Chair, to represent the object attributes and type.

Our DVE system, LivingSpace, is an implementation of the LivingWorlds specification, and uses the Avatar implementation to represent users. We execute a walking animation when the user moves, with other animations under direct user control. We use dead-reckoning based on veclocity, path curvature and angular velocity to reduce the bandwidth used by position and orientation updates, and to avoid jerkiness.

The LivingSpace system has been used to implement a conferencing application including a shared slideboard and spatialised audio conferencing. We hope to investigate further the use of avatars in multi-user interfaces, as well as adding more capabilities to our avatars.

References

[1] Norman I. Badler. Real-time virtual humans, *Proceedings of the fifth Pacific conference on Computer Graphics and applications*, pp. 4-13, IEEE Comput. Soc. Press, 1997.

[2] Norman I. Badler, Cary B. Phillips, Bonnie Lynn Webber. *Simulating humans*, Oxford University Press, 1993. ISBN 0-19-507359-2.

[3] Tolga K. Capin, Igor Sunday Pandzic, Nadia Megnenat Thalmann, Daniel Thalmann. A dead-reckoning algorithm for virtual human figures, *Proceedings of IEEE 1997 Annual Internation Symposium on Virtual Reality*, pp. 161-169, IEEE Comput. Soc. Press, 1997.

[4] DIS. 1278.1 IEEE standard for distributed interactive simulation - application protocols, ANSI, 1995.

[5] H-ANIM. Specification for a standard VRML humanoid, version 1.0, 1998, on-line paper `http://ece.uwaterloo.ca:80/~h-anim/spec.html`.

[6] Michael Gleicher. Retargeting motion to new characters, *SIGGRAPH 98 Conference Proceedings*, pp. 33-42, ACM SIGGRAPH, 1998.

[7] Ronald N. Goldman. Transformations as exponentials, *Graphics Gems II*, pp. 332-337, Academic Press, 1991. ISBN 0-12-064480-0.

[8] Rycharde Hawkes, Mike Wray. LivingSpace: a LivingWorlds implementation using an event-based architecture, submitted to VRML99.

[9] Jessica K. Hodgins, Wayne L. Wooten, David C. Brogan, James F. O'Brien. Animating human athletics, *SIGGRAPH 95 Conference Proceedings*, pp. 71-78, ACM SIGGRAPH, 1995.

[10] LW. Living Worlds specification, Draft 2, 1997, on-line paper `http://www.livingworlds.com/draft_2/index.htm`.

[11] Colin Low, Laurent Babarit. Distributed 3D audio rendering, *Computer Networks and ISDN systems*, 30(1998), 407-415.

[12] Eadweard Muybridge. *The human figure in motion*, Dover, 1955. ISBN 0-486-20204-6.

[13] Patrick-Gilles Maillot. Using quaternions for coding 3d transformations, *Graphics Gems*, pp. 498-515, Academic Press, 1990. ISBN 0-12-286166-3.

[14] Peter Ratner. *3-D human modelling and animation*, John Wiley and Sons, 1998. ISBN 0-471-29229-X.

[15] Michael Spivak. *A comprehensive introduction to differential geometry*, vol. 2, Publish or Perish, 1979. ISBN 0-914098-81-0.

[16] Nadia Magnenat Thalmann, Daniel Thalmann, editors. *Interactive computer animation*, Prentice-Hall, 1996. ISBN 0-13-518309-X.

[17] VRML. Virtual Reality Modeling Language, ISO/IEC DIS 14772, 1997, on-line paper `http://vag.vrml.org/VRML97/DIS/`.

[18] Mike Wray, Rycharde Hawkes. Distributed virtual environments and VRML: an event-based architecture, *Computer Networks and ISDN systems*, 30(1998) 43-51.

[19] Vladimir M. Zatsiorsky. *Kinematics of human motion*, Human Kinetics, 1998. ISBN 0-88011-676-5.