Interactive Multiuser VEs in the DIVE System

Olof Hagsand
Swedish Institute of Computer Science

A virtual environment (VE) is a real-time simulation of a real or imaginary world where users navigate and interact with 3D objects within it. In a fully interactive multiuser VE, several participants connected by a network may meet, collaborate, and work.

An appealing characteristic of VEs is the ability to offer intuitive modes of interaction, analogous to the ways in which humans communicate with each other or manipulate objects in the real world. VE applications can use 3D spatial properties to represent users and to model interaction, offer direct manipulation interfaces that mimic actions in the real world, and use immersive techniques that give participants the sense of being embedded in the synthetic environments. As such, distributed VEs can be seen as powerful human-computer interfaces. They thus provide new ways to communicate remotely and to access information over networks.

The work presented here centers on multiuser VEs. The discussion includes how to design and construct networked, multi-participant, wide-area, on-line VEs as well as how to create useful metaphors and techniques for communication between participants and applications within such environments.

DIVE is a multiuser, distributed VE system developed by myself and colleagues at the Swedish Institute of Computer Science (SICS). DIVE differs from similar approaches in its dynamic and flexible—some say anarchistic—capabilities and its focus on interaction and human-human communication. Starting as a lab tool in 1991, it has evolved into a well-developed system implemented on many Unix platforms and used to prototype VEs at several research sites around the world.

The figures used in the examples are snapshots from existing, real-time, functional applications of the DIVE system. The system, along with some examples in this article, is available over the World Wide Web at http://www.sics.se/dive/.

Multiuser VEs

An interactive multiuser VE is a distributed application where multiple users are simultaneously present within a simulated 3D space. We call such a shared environment a (virtual) world. All users perceive the same world by seeing the same scenes and hearing the same sounds, albeit from separate spatial locations. Each user can navigate through 3D space and see, meet, talk, and collaborate with other users in the environment. Worlds are populated by objects with or without graphical representations.

An example of a multiuser VE, Figure 1 illustrates a world seen from a user's viewpoint. The scene includes two other users or automated players (actors) with simple representations (embodiments). The actors stand in a room containing a couple of plants and a lamp illuminating the scene. A landscape is visible through a window.

The actors may move freely through the world. They may explore the building, move objects around, communicate with other participants (with gestures, audio, or text), and dynamically create and modify objects on the fly, for example by collaboratively modifying the 3D scene. Additionally, objects can have autonomous behaviors associated with them. For example, the lamp in the figure can be turned on or off by pointing to it.

The embodiments in Figure 1, called “blockies,” are very simple representations of humans. The head of a blockie might correspond to a real user’s head wearing a head-mounted display (HMD), while the eyes represent points in space from which a user views the virtual world, possibly in stereo. The movement of a data glove or other input device can be coupled to a virtual pointing device (such as a hand). In Figure 1, the user on the left has just turned on the light. The user’s virtual pointing device appears as a white line.

Figure 2 shows a graphically more elaborate example: a conferencing application employing real-world metaphors extensively. (See the sidebar on p. 39—this example is on the Web.) Three actors communicate in three dimensions with text and...
audio in a traditional conferencing environment. The three documents on the table and the wall display are active, displaying text and graphics coupled to external tools in the outside (real) world.

**Interaction with individual objects**

An interactive multiuser VE, in which participants interact with objects constituting a 3D scene, must support the creation and modification of individual objects. Information is distributed on a per object basis, where individual objects are addressable and may be requested and retrieved over the network.

In a truly interactive environment, participants must be able to dynamically create, modify, and remove objects, as well as enable others to do the same. This means that objects are not owned by a creator—once introduced, an object may be accessed and modified by any participant.

If combined with autonomic behaviors, such a shared environment has enormous power. For example, an object created and released in a world can perform actions by itself and be picked up and used by other participants.

**Why networking is important**

Peers of such an advanced distributed application need to exchange large amounts of information, including object and world definitions, navigation commands, audio, and bitmaps. Over an internet, messages might have to be delivered among many participants over high-latency paths. Recent round-trip estimates measured in DIVE experiments range from 25 milliseconds within Sweden, 200 ms to American sites, and 800 ms to Japan.

In such an environment, it is crucial that participants experience “acceptable” delays. The ultimate endpoints of communication are the human senses and the brain’s actual perception of sound and pictures. Typically, end-to-end latency as experienced by a human user has an upper bound for acceptability. For audio this limit is roughly on the order of 100 ms, while interactive manipulation requiring feedback has an even lower bound. Unfortunately, network and physical realities make these limits unreachable for global distributed systems. Other properties, such as object motion and presentation, might be less sensitive to latency, especially if described by behaviors evaluated locally at each peer.

System designers must therefore address protocol and performance issues and design the communication between peers to take advantage of the network’s available bandwidth. In short, the amount of traffic between peers must be reduced and end-to-end latencies minimized.

Summarizing, two distinguishing features make interactive multiuser VEs possible: interaction with individual objects and scalable networking. We will later return to how these issues were addressed in DIVE.

For a discussion of networked multiuser VE systems, see the sidebar on the next page.
Networked Multiuser VE Systems

Several other multiuser VEs take distribution and multiuser aspects into consideration. The VUE system is a distributed client-server architecture where processes communicate via asynchronous message passing. In the Minimal Reality (MR) Toolkit, a set of master processes is connected pairwise to other master processes. Messages may be sent unreliably between one process and the other peers.

Several systems are based on an object-request broker approach. Their interfaces let objects be accessed remotely (by asynchronous message passing) through a client-server system. Massive and BrickNet are examples of such systems.

Many systems rely on the DIS protocol, originally designed for multi-party training in combat situations. With this approach, multiple immobile objects form a static background, such as landscapes and buildings, while the movements of a smaller set of dynamic objects, such as vehicles, are distributed to all connected peers. One such system is NPSNet (Naval Postgraduate School Net), in which large-scale distribution issues were taken very seriously. For example, NPSNet is currently addressing how to increase the number of participants and information space to the range of thousands of participants over an internet.

The DIVE software model

To understand the fundamentals of the DIVE environment, let's briefly consider the platform's software model.

Distributed entities

DIVE entities form the basic units of distribution that can be addressed, requested, and distributed. Figure 3 shows a class hierarchy (used only for modeling—DIVE is implemented in plain C) where the entity class is the top-level abstraction. An entity has a globally unique identifier used for addressing, a name, a behavior description, and a set of properties usable for application-specific data. The entity and view classes are abstract classes; that is, no instantiations are allowed.

Entities are structured hierarchically in a tree: a world is a root, while dive_objects are nodes, views are leaves with 3D graphical representations, and lights are leaves with a light model definition.

Figure 4 shows an example of an entity hierarchy of the active lamp object shown in Figure 1. lamp and active are examples of dive_objects, while views and a light (a light bulb) are depicted graphically. The active object has an associated behavior triggered by user interaction. For example, selecting or stepping up close to the light activates the light bulb.

DIVE objects

DIVE objects carry the essential logical, interaction, and dynamic information in a world. This includes geometrical orientation, material descriptions, and variables controlling interaction and rendering. When DIVE objects are composed hierarchically, their own geometrical transformation is composed with the rotation and translation of the object at the next level in the hierarchy.

The following DIVE file format definition specifies the lamp object with a default black material and a displacement of 10 meters in the z-axis direction:

```
object {
    name "lamp"
    translation v 0 0 10
    material "black"
}
```

Graphical representations: Views

Lines, spheres, cylinders, boxes, grids, and polygons are examples of subclasses of the view class. Views are passive graphical 3D representations that may be dynamically created and modified. When a user interacts with a view, typically by pointing to it, the DIVE object closest to it in the hierarchy handles the actual interaction. For example, the active object in Figure 4 defines the behavior of the views and lights below it.

The lamp pole in Figure 4 is an example of a cylinder:

```
view {
    CYLINDER
    0.02
    0.02
    1.3
}
```

where 0.02 denotes the two radii and 1.3 the height of the cylinder.

Multicast domains: Worlds

A world represents a separate virtual space disjoint from other worlds, with its own set of objects, actors, and views. The world information is common to all entities within the space, such as background light, fog, and spatial boundaries.
The following "park" world is defined with a fog and a default (grey) background color:

```plaintext
world {
  name "park"
  start v 0.0 -5
  info "A test world showing a nice park"
  background 0.2 0.2 0.2
  fog 0.0002
}
```

A world defines a separate spatial domain and is therefore naturally assigned a multicast address. Only peers that have joined a world-specific multicast address can listen to events occurring in that world. Requests of entities belonging to the world can be made by sending a message to the world’s multicast address. A DIVE name server handles the assignment of multicast addresses and name-to-address lookup.

**User representations: Actors**

An actor, typically a user or an automated process, represents a process-bound entity that performs actions within a world. The actor modifies objects and parameters and sends messages to other entities within the world. Messages either result in concrete changes in the database, such as "moving an object," or are more abstract, such as "an actor has picked up an object."

Since objects can be modified by any actor, not only the creator, concurrent modification requests must be resolved. Here, we assume that actors "own" objects for a long time and that concurrent modifications seldom occur. Entities are therefore protected from concurrent "writes" by a simple object-based token-passing algorithm: In the rare case of a conflict, one actor blocks until it receives the token.

An actor can change worlds by entering a gateway, an object serving as a portal to another world. When the actor passes through the object, a collision manager signals a collision, a name server is queried for a multicast group to join, and the actor is transferred with its embodiment to the new world.

**Dynamic interaction and behavior**

The software model described above forms the basis upon which essential VE functions are implemented, such as collision detection, dynamic object behavior, user interface support, and audio. I will now describe these functions as implemented in DIVE.

**Behavior in objects: The DIVE/Tcl interface**

DIVE worlds are not passive. Entities are somewhat autonomous—they react to stimuli, move, transform, and adapt to the changing environment. This degree of autonomy is achieved by associating Tcl scripts with entities. Typically, an event in the system triggers a DIVE/Tcl script, resulting in a set of actions. Because Tcl is portable and interpretative, scripts are replicated along with the entities and can be executed immediately on any platform without compilation.

In the following (somewhat simplified) DIVE/Tcl example script, an object selection starts a simple motion:

```plaintext
proc move-up {id type actor srcid }
  for {set i 0} {$i < 100} {incr i} {
    dive-sleep 100
    dive-move $id 0 0.5 0 LOCAL_C
  }
  dive_register select move_up
```

The `move-up` Tcl procedure is registered by `dive_register`, which is invoked when a DIVE `select` occurs at the object containing the script. When the `move-up` procedure is called, the parameters identify which actor and object invoked the script. The `dive_move` procedure itself displaces the object 0.5 meters in the local coordinate system every 100 ms. Really powerful event- and timer-driven behaviors are specified by having access to the complete DIVE functional interface in combination with Tcl iterations and functions.

We have found autonomous behavior to be extremely useful in "programming" VEs. Graphical modeling is a smaller problem; normally a standard 3D modeling tool will serve. A more important aspect of modeling is how to animate and put life into the objects constituting the VEs.

**Collision detection**

Collision detection is an essential service in a VE system. Consequently, collision detection forms the basis of many DIVE functions, including gateway detection, interaction, and communication support.

The Massive4 approach influenced us to have a collision manager. Processes register interest in certain objects and actors, and the collision man-
Figure 5. An actor approaching a lamp intersects a surrounding volume. As a result, the lamp is turned on.

ager generates collision signals for the registered entities as their volumes intersect.

In DIVE, system servers implement collision managers and the handling of collision events, so the semantics can be altered easily to satisfy different application needs. This approach supports different models, including the spatial interaction model developed by Fahlen and Benford, and the use of geographical areas in simulation.

Intersecting volumes and behaviors can be used in simple interfaces. As an example, Figure 5 illustrates the use of volumes in animating a VE scene. In the figure, an actor is approaching a lamp. The system detects the actor's penetration of the volume surrounding the lamp (an aura), indicated by the yellow lines. The lamp—an autonomous object—reacts to the collision by turning on the light, as defined with the following DIVE/Tcl script:

```tcl
proc turn_on {id1 id2 type} {
    set id [dive-find-subobj $id1 dive-light $id on "bulb"]
    dive_light $id on
}
dive_register collision-on turn_on
```

where the `turn_on` procedure is called when a collision signal occurs and `dive_light` turns the actual light source on.

User interface

A process with a 3D rendering module that associates a real human user with a virtual user—an actor—is called a visualizer. A visualizer presents a graphical representation of a virtual world and lets the user interact with that world by selecting and grabbing objects, sending messages to actors, setting up audio connections, and so forth. DIVE provides many visualizers, each suited for different interface requirements. For example, with an immersive interface, magnetic trackers monitor body movements, and an HMD provides the world view. With a nonimmersive interface, the regular mouse and keyboard are used, and the world is viewed on a regular screen.

Figure 6 shows a typical DIVE user interface. It has three buttons and two vehicle icons placed on a visor, which is just an invisible object acting as a placeholder for the user's icons. Usually, the visor is placed just in front of the actor's eyes, so actor and icons move together. The visor thus forms a "2D" working area providing easy access to control and monitoring icons.

Vehicles help users navigate in 3D space. The vehicle icons, placed above the three buttons, include the rectangle—a "walking-style" vehicle—and the triangle—a "flying-style" vehicle. For example, users interacting with the flying-style vehicle see their embodiments move as if carried by an airplane.

The icons in Figure 6 are autonomous objects defined by DIVE/Tcl scripts. When a user interacts with these objects, things happen—the user's embodiment moves, menus appear, and other objects are affected. The car in the foreground has a steering wheel, an ignition button, and a speedometer. These control the car's movement, and any user may manipulate them. An actor can attach to the car and follow it as it drives along. Other actors also may enter the car and control its movement.

Figure 6 also shows another actor controlling a different car. As a consequence of the dynamic environment, an actor can change embodiments at will. For example, an actor could select the helicopter as a new embodiment.

Communication by text and audio

It is essential to support communication between participants in multiuser VEs. Dumb participants can only communicate through body language, which is extremely limited given the constrained embodiments. Therefore, we made it possible for DIVE actors to communicate directly using text messages and live audio.

The endpoints of audio communication are associated with entities in the 3D environment. Thus, audio sources, including microphones and Internet audio conferences, can have physical representations such as a mouth or a radio, respectively. The location and orientation of the entities can then be used to model 3D audio—or at least attenuate audio with distance.

In several 3D audio and speech synthesis experiments, the spatial properties of entities were given as input to external tools. Using this technique, current versions of DIVE use standard audio conferencing tools, such as Nevot (Network Voice Terminal), to convey real-time audio. Thus, DIVE does not distribute or process the audio data itself, but has the ability to feed external tools the necessary 3D location information.
Multimedia documents

An interesting capability is enabled by the DIVE Web interface. Apart from the fact that initial descriptions of DIVE worlds and objects may be read via Web protocols in VRML or DIVE format, general-purpose multimedia (Mime) documents can be presented by external tools. In this way, a DIVE world can contain virtually any multimedia document, including MPEG movies and bitmaps, that can be activated by DIVE/Tcl scripts. Sound is especially useful in a VE, since it can enhance worlds: the sound of a door opening, the click of a button, and background music.

Further, a standard Web browser can be interfaced to DIVE so that the browser accesses “traditional” 2D documents while VE documents are traversed in DIVE. This arrangement resembles the approach taken by many VRML viewers.

Let us now proceed to the distributed domain. First I’ll discuss general issues, then describe how DIVE entities are distributed and how networking capabilities are used.

Networking issues in shared VEs

Many existing approaches to VEs are inherently single-user approaches extended to the multi-user domain. These systems might offer some sort of primitive distribution mechanism such as asynchronous message passing, which gives little help to a programmer because all distribution and synchronization problems must be dealt with explicitly.

We believe that a distributed multiuser VE system should be designed from the start to be fully distributed, with the focus on performance and scaling considerations.

Programming distributed applications is difficult because of their inherent concurrency. This is especially true when objects are fully distributed and any process in the system can access them. Consistency requirements include (loosely) that all processes see the same data and that corruptions do not occur because of concurrent modifications of the same data. Several techniques exist to ensure consistency, such as distributed transactions, distributed read/write locks, and so on.

One important performance criterion for distributed VE systems is latency, that is, the time it takes for a message to reach its destination. Network latencies range from under a millisecond on local area networks up to several hundred milliseconds on a wide area network. A related performance factor is throughput. On a network with bounded bandwidth, only a limited amount of data may be serviced per time unit.

It is therefore important to design the distribution architecture of distributed VEs so as to minimize latency. At the same time, the applications should not violate the bandwidth limits of the internetwork, since doing so causes congestion and loss of packets, or overrun of the receiving hosts.

Distribution techniques

A client-server application is based on an asymmetric model where a large set of clients connect to a small set of servers, often only a single server. Replicas of data reside on servers, from which clients request the data. Updates from clients are sent to a server before they are distributed to or requested by other clients. Thus each message must be sent at least twice, which might increase latency. Client-server architectures have an advantage, however: Consistency requirements are easily satisfied, since data is modified centrally and then distributed to clients.

In peer-to-peer distribution, each peer in the distributed environment communicates with other peers on an “equal” basis. Replicas of data exist in each peer, and updates need only be sent once. A common peer-to-peer distribution technique is full replication, where each participating peer actively manages its own complete copy of the data. Applications modify data in what they perceive as a shared database, while modifications are transparently distributed among all peers. Full replication achieves a high degree of availability (no need for remote access) and some degree of fault tolerance. It is also conceptually simple.
Multicast protocols potentially use network resources better than unicast protocols and thus (indirectly) reduce latency and increase throughput on a network with limited resources. (Note that multicast on a transport level, such as reliable multicast, may or may not utilize network-level multicast.) In a client/single-server architecture, multicast provides no benefit when the client communicates with the server, since there is only one recipient. However, multicast can be used when sending to a group of peers, such as in server-to-client and peer-to-peer communication.

Therefore, peer-to-peer distribution combined with network-level multicast minimizes latency and uses system resources well. Thus it is a suitable distribution technique for large-scale distributed VEs.

**How VEs scale**

As the number of peers increases in a distributed VE system, the number of messages needed to communicate between these peers increases. This increases the load on sending and receiving peers, as well as on the network. A system is scalable if it can handle a large number of peers before its overall performance degrades. Even with network-level multicast and peer-to-peer communication employed, a system might fail to scale. Perhaps the most important scaling criterion, then, is how the application itself is designed.

For example, a distributed VE system using unreliable multicast might scale better than one using reliable multicast. This happens because reliable multicast protocols using positive acknowledgments suffer from the “ack implosion” problem—even though the data is sent by multicast, the acknowledgments are sent using point-to-point messages from each receiving host, leading to an increased number of messages as the number of peers increases. In reliable multicast protocols using negative acknowledgments, or nacks, receivers send nacks when they discover that packets have been lost. This approach has more potential for scaling, but it is an ongoing research problem to find plausible solutions that scale well.

Applications can send less information, for example by employing compression or using dead-reckoning techniques, where kinematic equations of the objects may be used to locally compute successive positions and orientations. DIS-based systems, for example, employ dead-reckoning techniques.6

Another technique is to receive less information. In an “optimal” application, peers only receive exactly the data of interest to them. If the data is partitioned in some way, each partition can be sent on a separate network connection. Then hosts need only process messages from those connections of interest to them. In fine-grained partitioning, a process could, for example, register interest in specific objects or even aspects of objects, and then receive updates to just that set of objects. In a coarse-grained partitioning, data could be separated in disjoint sets, for example into completely separated worlds, or partitioned into geographical areas.7

Whatever the approach, future distributed VE systems will require hundreds or thousands of multicast addresses that can be simultaneously connected.11 In comparison, current network interfaces have a relatively low limit on the number of multicast addresses to which they can connect simultaneously. In the case of IP multicast, it is unclear whether thousands of multicast addresses per application are feasible. Despite an extended address space, routers might not be able to handle the required amount of state.

Hierarchical encoding techniques provide a plausible approach, in which several levels of resolution are sent; the highest levels are discarded at gateways with limited bandwidth. However, this method might be hard to carry out when the degree of resolution requires knowledge of application semantics. For example, a peer might want low resolution for objects further away in virtual space and high resolution for objects that are close. But being far away in virtual space might have no correlation with being far away in “networking” space.

Many of the scaling issues in distributed VEs depend on how the application is designed. Increasing network capacity and processing power make distributed VE systems possible, but to support hundreds or thousands of simultaneous users requires software solutions to decrease the number of messages.

**Distribution in DIVE**

DIVE addresses scaling considerations by using a negative acknowledgment reliable multicast protocol, partial replication, and a simple dead-reckoning module. The distribution model is based on peer-to-peer distribution, multicast protocols, and coarse-grained partitioning of data. Peers connected to the same multicast group interact by making concurrent accesses to replicated data (entities) and by sending messages. Messages addressed to a group are relayed by multicast pro-
tocols to each peer.

Earlier versions of DIVE use a full replication model where peers communicate by a reliable multicast protocol using positive acknowledgments. This approach worked fine for 10 peers in one simultaneous session on a local area network, but failed beyond this limit. (Note, however, that several worlds can run in parallel as well.) Current versions use partial replication and a reliable multicast protocol based on a negative acknowledgment scheme. This approach is based on the analysis presented earlier: (1) the number of messages and end-to-end latency are minimized, and (2) the application itself is designed for large-scale internetworking.

Let me now describe how these issues are addressed in recent versions of DIVE.

Networking and multicast protocols

We pursued an idea originally in SRM\textsuperscript{13} to reduce the amount of message passing and thereby minimize network load and increase scalability. The method uses multicasting heavily, makes communication object-based, and bases reliability on a negative acknowledgment request/response scheme.

In our approach, communication objects are equivalent to DIVE entities. A sending protocol peer need not store messages until their arrival has been confirmed by all recipients. Instead, it may request the latest object replica from its local application by a callback.

If a peer detects a missing update message or just requests an object, it simply requests the object on the associated multicast address. By round-trip-time estimation and a timeout algorithm, the closest peer with the latest version of the object responds also by multicast to inhibit other similar replies. In this way, the network is not flooded by replies; in the optimal case, only the closest peer replies.

Reliable service needs support from the application in the form of callbacks. Therefore, our transport service (Sid) is implemented in user space and is an integral part of the DIVE software. However, the protocol has been used for other purposes, such as the jetfile file system for gigabit networks.\textsuperscript{11}

DIVE software model redesign

The new software model uses partial replication. Instead of replicating entities to each site, requests are sent over the network. This leads to a question related to many cache-consistency problems in distributed systems: by what criterion should objects be fetched (and cached) at the local peer?

Since several approaches exist on this issue, the semantics of object replication is not part of the core DIVE system. Typically, objects are requested based on visual range, such as all objects seen by an actor, or collisions, such as detected by a collision manager controlling a geographical region.

When a DIVE peer intends to join a world, it contacts a name server in order to retrieve a multicast address and thereby join a session. An initial world description might exist on a remote system, accessible by a universal resource locator (URL). However, only the first member of a session loads the initial description—all subsequent members request world state from an already connected peer. The new peer requests initial world state over the multicast address and receives an up-to-date replica from the closest peer with the latest version of the world. This brief initial world state contains bounding box approximations that serve as input on which entities can be requested.

Since partial replication means that not all state is replicated at each peer, data can be lost if peers leave the group. In a fault-tolerant application, therefore, the partial replication scheme might not be adequate. In this case, a world server can keep track of all events in a group and log changes to persistent storage.

To keep the number of messages low, DIVE also implements a simple dead-reckoning module based on linear and angular velocity. These work fine for regular motion, such as vehicle models, but such a simple model cannot correctly define general human motion.

Sample session

Figure 7 illustrates a sample DIVE session,

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig7.png}
\caption{Traffic received by one peer in a DIVE session with three participants. Elapsed time is shown on the $x$ axis, while the received rate in Kbytes per second appears on the $y$ axis.}
\end{figure}
showing the incoming traffic for one peer in a medium-sized world with three actors. The example typifies DIVE sessions—a number of participants join a world and then perform actions within it. The measurements are made after parsing the incoming messages. The world, also shown in Figure 1, contains 750 entities corresponding to approximately 200 Kbytes. When the actors move, they transmit 70 bytes of coordinate transformation updates every 50 ms.

In Figure 7, a first peak in incoming traffic appears at \( T = 0 \), which corresponds to the fetching of the initial world description. At \( T = 7 \) and \( T = 16 \), the embodiments of the two other actors are received. After an initial idle period, the actors start moving at \( T = 20 \), visible as gradually increased traffic.

Despite its simplicity, the sample session illustrates some key points. For one thing, the initial world description is comparably large. In relation, actor movements generate small amounts of traffic. Interestingly, partial replication can help bring down the initial bandwidth requirements, while fine-grained partitioning could decrease the continuous actor traffic, which increases linearly as the number of peers increases.

However, the world shown in Figure 1 does not contain textures; a textured world would be much larger. The initial peak at \( T = 0 \) would also be higher and narrower if the measurements were made before parsing, essentially limited only by network bandwidth. Further, more complex actor motion, such as general human motion, generates more continuous traffic.

**Discussion**

Because of the intuitive ways 3D information can be presented, shared, and interacted with, we believe that large-scale distributed virtual environment systems have great potential as a general interface to the Internet and the Web. This article has shown the power of multiuser interactive VEs and their realization in the DIVE software platform, with special emphasis on interfaces where interaction between humans is essential.

Recently, the VRML\(^6\) effort to extend Web documents with 3D has shown that the Internet community, including hardware and software platforms, is ready for 3D. However, interaction and multiuser aspects, as highlighted here, have only recently been addressed within the VRML discussion and are not a part of VRML 1.0. As such, the VRML approach defines inherently static 3D scenes.

The approach in DIVE has similarities to many DIS-based systems. The DIS protocol\(^5\) itself, however, is designed for military use in simulation-type environments and has little support for the dynamic environments we envision. As long as the DIS protocol does not support the flexible sharing, creating, and modifying of new objects and interaction between users, gestures, and behavior, it will be difficult to use with intensely interactive interfaces and applications. Recent experiments with DIS-based systems have shown that they can support humans, but it remains to be seen whether DIS itself can be extended to support true interactive environments, such as a conference.

Although distributed VEs promise a lot on the interface side, technical limits to large-scale participation still inhibit general use. In addition to and apart from bandwidth considerations, one limit is current multicast network interface limitations and router support. Until those problems have been resolved, our plans to assign multicast addresses on a finer granularity, such as on an entity basis, cannot be realized.

The newer DIVE versions are presently being evaluated. We have proven they work for 20 participants on a wide-area network with network latencies below 200 ms. It is an ongoing activity to push this limit further.

The DIVE effort has a long history in supporting and designing multiuser VEs. We hope that our experiences concerning flexible interaction and multiuser support will serve as input to the current construction of virtual environments and future 3D development of the World Wide Web.

**Acknowledgments**

I would like to thank several people involved in the DIVE project, including Magnus Andersson, Christer Carlsson, Lennart Fahлен, Emmanuel Frécon, and Olov Ståhl. I also thank Peter Sjödin and Stephen Pink, as well as several anonymous referees, for helpful comments.

**References**

Olof Hagsand, the prime architect of the DIVE system, is a researcher at the Computer and Network Architectures group and the Distributed Collaborative Environments group at the Swedish Institute of Computer Science. His present research interests include operating system support for multimedia and distributed virtual reality. He received his MS degree in engineering physics from Uppsala University in 1986 and the PhD degree in computer science at the Department of Teleinformatics at the Royal Institute of Technology in Stockholm in 1995.

Contact Hagsand at the Swedish Institute of Computer Science, Box 1263, 164 28 Kista, Sweden, e-mail olof@sics.se. His home Web page is located at http://www.sics.se/~olof.html.

Examples on the Web

Figure 2 in this article shows a VE conference application with three people simultaneously connected to the DIVE system. The people, represented by embodiments, interact, talk, and move in a shared environment. In this nonimmersive example, the movement of the embodiments is achieved by DIVE/Tcl scripts associated with each actor. The scripts are triggered by simple menus, and no special 3D devices are required.

I wanted to increase understanding of the example and the system as a whole by going beyond the limits of the illustration in Figure 2. Therefore, I've made the conference example available in a more interactive form on the IEEE Computer Society's Web site at http://www.computer.org/. You can find the example by following the menus through Publications to Magazines to IEEE MultiMedia's spring issue and this sidebar. There, Figure 2 is accessible as three Web items, each illustrating the same conference situation in different ways:

- A DIVE world specification. The DIVE world enables full interactivity with the scene and can also be run in multiuser mode. However, this item can only be accessed if you are running the DIVE system—available for a limited set of Unix platforms at http://www.sics.se/dive.
- A VRML 1.0 scene. The VRML scene shows a static 3D version of the conference situation and can be accessed by standard VRML web browsers. Note that this item only shows a static "snapshot" of the example, much like a 3D view of Figure 2, lacking behavior and interaction capabilities. The VRML scene was created from the DIVE tool during a conference session by exporting the scene into VRML format.
- An MPEG movie. The movie shows the conference session when the participants interact, walk, and move. It was recorded during a live DIVE session using the movie recording facility provided by the platform. The MPEG compression, however, was made off line.