CHAPTER 19

Internet-Based Virtual Environments

Charles E. Hughes, J. Michael Moshell, Dean Reed
University of Central Florida

22 May 2000

Hosting virtual worlds on the Internet introduces many problems not inherent in private networks. These include security, reliability, speed and differing agendas of participants. The purpose of this chapter is to review the history and present status of Internet-based virtual environments, and to discuss a number of efforts designed to meet perceived future needs.

1 Internet Protocols

1.1 The basics

This section sets the context for the rest of the chapter. It is by no means a thorough journey into the world of Internet protocols. Those wishing such a presentation are referred to Stevens (1994).

Several conceptually simple properties make the Internet work. Its richness comes from building new protocols and paradigms on top of this well-conceived foundation.

First, for a computer (host) to participate as one of the many nodes in the worldwide network known as the Internet, it must be assigned a unique Internet Protocol (IP) address. An IP address (old form) is a 32-bit number, usually presented as a "dotted-decimal notation," which includes the decimal equivalents of the four bytes comprising the address, separated by periods. The great popularity of the World Wide Web has resulted in the need for a richer set of addresses. This is being met by a new standard (IPng) that extends addresses to 128 bits, represented by 8 sixteen-bit segments, separated by colons, rather than periods.

IP addresses are assigned statically to hosts that constantly reside on the Internet, or dynamically, from a collection of locally available addresses, whenever a transient host joins the Internet through some server.

Communication on the Internet occurs between two hosts, using their respective IP addresses. Long messages are broken up into shorter packets, source and destination addresses are appended, and the data is effectively and efficiently routed from source to destination.

Basic IP communication is then extended with a very thin layer to create UDP (the User Datagram Protocol). The only thing that UDP adds is the notion of ports and some minimal error checking. Ports are the logical channels through which peer applications communicate. A port is represented by a 16-bit address, with low-numbered (well-known) ports being reserved for standard kinds of interactions, e.g., file transfer (21), time-of-day (13), or web services (80). Packets associated with UDP communication include the port numbers of the sender and receiver, in addition to the IP addresses of these participants as required in any IP packet.

UDP is employed for very simple kinds of interactions. For example, to inquire about the time of day, a user need only send a message (any data will do) to port 13 of a designated host. The sender must establish a local port through which this communication occurs. The recipient will then send a response back to the sender, using the port through which the sender started this interchange. The response is, of course, the time of day at the recipient's site.

UDP is inherently unreliable. There is no acknowledgement of the receipt of a message, unless the protocol of the service requires a return message, e.g., as in a time service. Furthermore, messages, even when received, may not be in the order in which they were sent. This latter property is a direct consequence of the fact that IP routing can be dynamic, allowing the system to effectively manage load balancing and adjust for lost links in the net. The low overhead of UDP makes it desirable for cases where speed is paramount, and where some loss or reordering of data is acceptable. Examples of this are streaming audio or video, and a global clock. In fact, streaming audio or video would be greatly compromised by an obsession with reliable receipt; can you imagine the player producing a great silence while it waits for one missed note?

In support of shared virtual worlds, virtual environments often use UDP when messages need to be communicated quickly, and the effects of lost data are minimal. Examples of protocols built in this way are often called "self healing." A standard example of this is in the communication of small movements of a character in a virtual world. Sending the new position of an object every time it changes, keeps every one's view up-to-date. Loss of a packet of positional information is easily compensated when later position packets arrive in a timely manner.

TCP (the Transmission Control Protocol) ,or TCP/IP as it is usually known ,–builds on top of UDP. This protocol adds, among other things, a sequence number to each message. The receiver uses the sequence number to determine if packets are missed or received out of order. Its obligation is to acknowledge receipt of messages. The sender must retransmit missed packets. The receiver must reorder out of sequence messages.

TCP also extends UDP by being a connection-based protocol. This means that TCP establishes contact, and keeps the connection open until some two-way communication task is completed. In contrast, UDP is used for quick exchanges. Writers often equate UDP to letter sending ; you may or may not get a response, but you at least let the receiver know your name (port) and address (IP address). TCP equates to phone conversations ; you establish a connection, participate in a dialogue, and then close the connection when one of the parties deems it is time to do so.

TCP is preferred over UDP in virtual worlds when losses of messages can create critical inconsistencies in the views of participants. For instance, if I pick up an object, and the message does not get to you, you may get frustrated trying to pick up this same object when it is no longer available.

In summary, the Internet moves information in packets of manageable size, using the following tools: IP addresses; message routing; ports; and optional reliability. These features serve as the building blocks to support all communication between participants in Internet-based virtual worlds (VWs).

1.2 HTTP protocol

The HyperText Transfer Protocol (HTTP) is built on top of TCP. This service, running on port 80 of a host, just listens to see if anyone wants to acquire hypertext files from this particular site. When a request comes in, the message within that request specifies what file is to be sent back. The initiating computer is then responsible to interpret the returned file, e.g., by displaying text and/or images in a browser's window, and by requesting additional data, as required in the page's hypertext description. This simple protocol is at the heart of the World Wide Web (WWW).

Note that HTTP is built on top of TCP, not UDP. This is because reliability is critical here; a document with phrases out of order or missing makes no sense.

Many Internet-based VEs use HTTP to help participants find and join currently available virtual worlds. Dynamically created hypertext can provide up-to-date information on existing worlds, and include the links needed to connect to these worlds. In addition, the advent of Java applets even makes it possible to dynamically provide participants with the most current programs they need to interact with these worlds. Users need never worry about having outdated VE software cluttering their machines, since these virtual world engines will be delivered on demand, and destroyed when no longer in use.

1.3 RMI, tuplespace and Jini

UDP, TCP and even HTTP are often too low level for the needs of developers of VEs. It's not that we cannot build what is needed ; all Internet protocols are built on top of UDP. It's that we will often be forced to work at the wrong level of abstraction, reinventing interaction protocols.

Many protocols have been developed over the years to support the kind of distributed collaboration required by shared virtual worlds. We will focus on just a few, all of which are currently Java-based, but not Java-dependent. We are ignoring CORBA, the mother of all such protocols, not because it's irrelevant, but only because it is not used in any of the systems we are discussing here.

RMI stands for Remote Method Invocation. For those readers familiar with remote procedure calls (RPC), this is a modern version of that old scheme. RMI is built on TCP, extending it to allow communication between remote objects (software objects distributed across a network) using the standard object-oriented (OO) paradigm of messages being sent to objects that respond in ways appropriate to their class identity. In effect this carries the OO message-passing paradigm into the world of distributed computing, even supporting the transmission of software objects as raw data, followed by the subsequent reconstitution of these objects at the receiver.

RMI is a natural tool for VE developers. With its availability, VE systems can send messages by reference or by value directly to remote objects, without having their designers develop arcane, hard-to-understand message passing schemes that turn out to be homegrown versions of RMI. In other words, VE developers are left to concentrate on issues central to their domains of interest, knowing that the existing infrastructure supports object-oriented message passing, even in the context of distributed objects.

Tuplespace is based on the conceptual framework of a globally shared memory (shared among a very large group of computers, potentially including all computers in the world) conceived in 1985 and refined in Gelernter (1991). His simple idea is to allow cooperating processes to write messages onto a global message board. These messages can then be read or consumed (read and then destroyed) by other processes, based on some pattern that matches the form of the original message. The term ‘tuple’ is a mathematical term for messages that are comprised of ordered lists of components. One can think of a tuple as a row in a relational database, except that no preordained schemes need to be used. Matching is then a form of "selection" from among those tuples that would "naturally" be organized into the same relational table. The absence of a matching tuple can block the requestor, thus providing coordination as well as communication.

Tuplespace was a theoretically interesting, but not heavily used concept up until very recently. With the advent of Java and its potential to make ubiquitous computing a reality, Sun and IBM, the two big Java players, were each seeking a vision for how data should reside in the massive network of the Internet. Both came to the conclusion that tuplespace is a very appealing option. IBM has developed an API called TSpaces (Wyckoff, McLaughry, Lehman & Ford, 1998), while Sun has developed JavaSpaces (Freeman, Hupfer & Arnold, 1999). At present, both of these are layered on top of RMI.

Tuplespace, like RMI, offers many benefits for VE developers. It raises the level of abstraction to allow one to think about communication, not with peers, but rather with a shared blackboard. This shared space can serve to communicate state information, and to coordinate critical activities, all without participating nodes needing to have any explicit information about the locale or even presence of other participants. Multiple tuplespaces can be used to categorize data, to provide redundancy or even to maintain a history of all interactions that ever occurred in a virtual community. Such a history can be an archeological treasure, especially if data is maintained in XML, the extensible markup language in which data is marked with domain-specific tags.

Sun has further enhanced the desirability of JavaSpaces by having made it one of the core technologies in its Jini vision of the network as the computer (Edwards & Joy, 1999). The VE community’s interest in Jini can easily be seen from Sun's own statement that "Jini technology provides simple mechanisms which enable devices to plug together to form an impromptu community—a community put together without any planning, installation, or human intervention. Each device provides services that other devices in the community may use. These devices provide their own interfaces, which ensures reliability and compatibility." If you replace the word device by citizen in a shared virtual community, the relevance to VEs is immediately apparent.

The goal of Jini is to make network computing painless – the service provider and service requestors take care of their own connectivity without human intervention. The protocols by which services are registered and found are key to the Jini vision. The registering of a service is initiated when a provider broadcasts a message seeking to discover a service broker. Jini objects that act as service brokers recognize these messages and report back their willingness to act as intermediaries between the service provider and its potential customers. After a bit of negotiation, the service provider sends information to one or more brokers describing the kind of service and its special attributes. So, for example, an instantiation of a virtual world might announce its availability, indicating that it is a virtual world with attributes of being "social" and focused on "Celtic music." Those wishing to participate in a virtual world can broadcast a message intended to find a broker who is acting as the intermediary for some virtual world service. This participant can specify required attributes, like "social." Any broker who "hears" the request will respond with an indication that it can negotiate on the user's behalf. A successful negotiation results in access to the service, perhaps employing RMI (the service provides a handle to a remote object) and/or the local instantiation of software objects on the client's machine.

Clearly, Jini and other systems that support directory services provide one part of the solution to helping users find virtual worlds that meet their needs. The very existence of such brokers highlights the needs of VE designers to pay close attention to the kind of audience they intend to serve. Some of those considerations are discussed in more detail in the next section.

2. Characteristics of Internet VEs

2.1 Lack of a dedicated network

The fact that Internet VEs must operate in the wide-open spaces of the Internet has profound effects on their design, and often on their inherent limitations. Three of the primary factors that the Internet influences are speed, reliability and security. In fact, these three factors often interact with each other in very negative ways.

Consider speed first. Traffic on the Internet doesn't just slow down message passing; it introduces unpredictability. Experiments in human factors have shown that variations in delays can often be more disconcerting to users than predictably slow performance. This issue of unpredictability becomes more profound as the number and geographical diversity of participants increases, unless the VE is designed from the outset with scalability in mind. Two systems that we will study, NetEffects (Das, Singh, Mitchell, Kumar & McGee, 1997) and Bang (Bjarnason, 1999), address this problem in effective, but radically different ways.

Reliability often directly conflicts with speed. For example, using TCP over UDP improves reliability at a cost in speed. One of the VEs we will study, DeepMatrix (Reitmayr, Carrol, Reitemeyer & Wagner, 1999), uses UDP where unreliability is tolerable and TCP where reliability is critical.

VE designers often ignore security, although it is critical to the acceptance of some systems (e.g., when they are employed in schools or business settings). Again, as with reliability, security often runs counter to speed. One particularly easy way to provide security is through the encapsulation mechanism of objects. This implies the use of RMI, or some other remote object passing system, adding another layer with its attendant delays to the network protocol. The use of this mechanism can be seen in VRMINet (Reed, Hughes & Moshell, 1999).

2.2 Device heterogeneity

The Internet playground was designed to support diversity among participating computers. This platform independence has many sides to it. Some of the most significant variables are the computation power, display capabilities, rendering speed, operating system and media support available in the machine being used.

Independence from specific machine architectures makes the Internet more accessible, but it can create a burden on VE designers. For example, to reduce network traffic we often place computational burdens on all participating nodes, whether they are high-end workstations or low-end PCs. A simple example of this is in the use of "dead reckoning" within ExploreNet (Hughes & Moshell, 1997) and NetEffects. The idea behind dead reckoning, a concept borrowed from the military shared virtual environment Simnet (Calvin et al., 1993), is that detailed movement of objects does not need to be communicated. Approximate positions can be computed locally, so long as each participating node knows the destination or direction of movement of an object, and the speed and perhaps acceleration at which it is moving. The upside of this is a reduction in network traffic. The downside is that all nodes interested in this object's movement must now consume computational power, including that needed to avoid unintended object interactions (e.g., a character going through, rather than around a wall).

In general there's no easy answer to the dilemma of supporting a wide range of machine architectures, but Java platform independence does provide a seductive middle ground, one that is being adopted by many of the newer systems appearing today. This comes with the usual performance penalties one faces when choosing complete machine independence, although that is less of a consideration with "just-in-time" (JIT) compilers.

Supporting a wide variety of display capabilities can lead designers to target their products for either the lowest common denominator, or towards only high-end users. Some systems, like ExploreNet, defer the decision to world builders. In this approach, the size of scene background images determines the amount of resolution needed on the displays of participants. Again, there is no "silver bullet" that solves this problem, although the device independence of Java is certainly a great help.

Device independence, however, does not resolve the issue of the diversity of rendering speeds. In many experiments we have run, the rendering component has turned out to be the bottleneck, more so than computational or network speed. The solution to this is generally out of the hands of VE designers, and more appropriately the concern of device and rendering engine designers. However, VE developers often feel forced into making tradeoffs between procedural graphical scene description APIs like Java3D (Sowizral, Rushforth & Deering, 1997), and declarative description languages like VRML (Web3D Consortium [Web3D], 2000). A compromise is to allow VRML to be used for the external description and Java3D for the internal representation. Bang employs such a strategy.

Operating system issues have led some VE designers to abandon diversity and focus on specific platforms (e.g., Microsoft VWorlds group). Again, Java is viewed by many as the savior, especially with its Java Media Framework, which supports a common protocol for existing and projected media objects, and the performance improvements one gets through its HotSpot.technology.

2.3 Community size and diversity

The Internet community is both large and diverse. Its size factors are ones of geographic separation as well as sheer numbers of potential participants. NetEffects addresses this problem through the use a three-tiered network system, in which participants are connected to a community server, which is in turn connected to a master server. Load balancing and relevance clustering are then used to keep the system responsive over time.

Diversity is an extremely challenging issue. On one level, this means that VEs should be designed with an international audience in mind. An orthogonal concern is accessibility (e.g., for those without sight or hearing). Apple, IBM, Microsoft and Sun, among others, provide APIs and guidelines to address these issues, but there is little actual employment of these standards within VE systems. Accessibility represents one of the most important research issues in the design of human computer interfaces (Stephanidis, 1997).

3. Standalone Shared VEs

VE systems developed before 1997 tended to be all-or-nothing, stand-alone packages. Here, the designers determined what features you would get, and how they would be provided. We will describe in detail two representatives of this class of systems, ExploreNet and NetEffect.

3.1 ExploreNet

ExploreNet was developed by two of the authors (Hughes and Moshell) at the University of Central Florida in 1994 based on the Virtual Academy model. Major revisions were made in 1996 and 1997 as part of an experiment in the Department of Defense Dependents Schools. The premier world hosted in ExploreNet is called Autoland. Eighth grade students at Maitland Middle School in Maitland, Florida authored this, based on the requirements of third grade teachers in Wuerzburg, Germany. The world was then used to help third graders understand social science concepts related to the acquisition, transportation and refinement of mineral resources. The software is available from the project web site (Hughes & Moshell, 2000) as unsupported freeware. Unlike the other environments discussed here, ExploreNet hosts two-dimensional worlds, designed specifically to support the constructionist model of learning. Chapter 52 in this book provides more information on how ExploreNet addresses educational issues.

A world in ExploreNet consists of a number of scenes. These scenes are connected through portals, although some characters can have behaviors that allow them to teleport. An avatar is always located in some scene. Props start out in a "home" scene, but can be conveyed from place to place as possessions of, or attachments to, other objects. Props can also propagate, producing fully functional clones. Figure 1 shows a scene from AutoLand. Here, two avatars, Sandy and John (he's the truck driver) are in the same scene, Pittsburgh. There are three well-labeled portals to other scenes. A truck is attached to John, and a pickaxe is, in turn, attached to the truck. Sandy possesses a sifter. These props move from scene to scene with the corresponding avatars. Communication is through cartoon-like speech lines. Normal communication is visible to all users whose avatars are in the same scenes. Additionally, an avatar can page a single recipient, or yell so all can hear.

The ExploreNet environment was developed in Smalltalk. The choice of Smalltalk led to a clean object-oriented design. Unfortunately, Smalltalk did not provide appropriate network and media interfaces, and so ExploreNet's developers had to build their own. Consequently, ExploreNet does not provide a rich set of mechanisms to address issues of security, nor does it support communication beyond simple textual conversations among participants. Later versions of Smalltalk, like IBM's VisualAge Smalltalk, have addressed these issues, but ExploreNet's future development has been migrated to Java.

ExploreNet employs a client/server architecture, but with a twist. The ExploreNet software supports three modes of operation: standalone; server; and client. If a user selects standalone, no sharing occurs – we'll ignore this case. If the user selects server operation, he must then choose a world to offer to potential clients. This user also is a client; in essence, the server spawns a local client who is automatically connected to this world. When a user chooses the client mode, ExploreNet searches the net (using well known or user selected sites) to find all available world servers. UDP is used here, since the servers need only report back the worlds they are offering. The client then filters the list of offered worlds to include only those for which descriptions are locally available (pre-installed on the client's machine). The worlds that meet these filtering constraints are then offered to the user. When a user selects a world, a TCP/IP connection is made to the associated server. This connection remains open until the user drops out or the server closes the world.

The primary features of ExploreNet's protocols that we will discuss are:

- Affordances;
- ground truth;
- global consistency;
- client drop-out;
- historical record.

ExploreNet worlds consist of objects and their potential *affordances*. The central objects of a world are its scenes and avatars. Scenes, in turn, possess connecting portals and props. All objects have associated image sets, behavior scripts and attributes, and can acquire additional images, behaviors and attributes from their *prototypes*. The prototyping system allows, for instance, a character named *Fred* to acquire part or all of its behaviors from a prototype called *Mentor* and another called *Human*. *Fred* can, in turn, be a prototype for other characters.

Attributes are specified as names (e.g., *hasChildren*) and associated weights (e.g., *2*). Scripts are written in a simple, homegrown, message-based language, with many primitive behaviors from which to build. Each behavior and portal has conditions that must be met in order for it to be active. These conditions can be based on the attributes of any objects in the world, and/or the identities of the object, the initiator of the action or even the relatives (prototypes) of any of these characters. The affordances of an object are its presently active behaviors. In addition to selectable behaviors, a world and any of its objects can have automatic behaviors that are triggered any time their conditions are met. These simple mechanisms of conditional and scheduled behavior activation, and conditional portals, along with means to trigger events randomly, make ExploreNet worlds appear to exhibit behaviors that are unpredictable and often interesting.

The evolutionary nature of ExploreNet worlds makes them interesting and useful for problem solving, as well as for quests and social interactions. Unfortunately, the system's primitive support for media, its non-standard scripting language, and its lack of integration with web browsers limits ExploreNet's general appeal and usefulness.

Over and above its use of a client/server model, ExploreNet employs several protocols worth discussing. In particular, each client maintains *ground truth* on the avatar(s) it controls. This means that critical decisions about an avatar's state are deferred to the client controlling that avatar, not to the server. However, the server always maintains ground truth on props. This approach is a compromise between the server-centric design seen in typical MUDs and the purely distributed approach taken in military distributed simulations.

Messages sent by clients are always to the server who provides *global consistency* to the ordering of events. In fact, clients are unaware of the identity of each other. This approach allows the server to impose an order on the receipt of messages by all clients (recall that TCP/IP is used here). The potential downside of making the server work too hard is alleviated by ExploreNet's protocol, which communicates a complex behavior as a simple message, not as a series of individual primitive actions. It is further alleviated by the fact that navigation is communicated as a destination, with the receiving clients responsible for path planning and actual articulated movement.

A world server expects regular communication from all its clients, and can use the lack of such communication to detect *client drop out*. The clients are correspondingly expected to send brief "I'm alive" messages, even when they are otherwise passive. If a server observes that a client has been quiet for too long, the client is sent a "wake up" message. Whenever a client fails to respond to three consecutive wakes up messages, the client's connection is closed and its avatars are made available to new clients. Similarly, a client expects regular communication from its server. If this fails to happen, the client can recast itself as a server, offering the world, in its current state, to other clients. Of course, that can lead to a world's replication, if a server goes down and more than one client chooses to continue the world.

As a world evolves, each client, including the server's attached client, keeps an *historical record* of all messages that have been received. Clients are also free to take snapshots of this history at any time. The complete history, or any snapshot can be used for retrospective analysis or to restart the world at any point in its evolution. In fact, retrospective analysis can be done from the perspective of any avatar, even with the ability to change the viewpoint from one avatar to another in mid story. Histories also allow late arrivers to a world to see how the world evolved to its current state.

3.2 NetEffects

NetEffects was developed by researchers at the National University of Singapore. The premier world hosted on NetEffects is called HistoryCity (Kent Ridge Digital Labs [KRDL], 1999). This world provides a shared community for 7 to 11 year old children in Singapore to learn about the history and culture of their city/state. HistoryCity is a virtual Singapore of 1870, consisting of 24 communities complete with historical buildings, costumes, and objects. The software is presently available for download at the Kent Ridge Digital Labs at KRDL, 1999.

A world in NetEffect consists of a collection of communities. These communities have a hierarchy of venues, typically consisting of an outdoor area, (e.g., a town square) and a number of buildings. The buildings, in turn, contain rooms. Avatars can move within a venue and from venue to venue, based on geographical relationships. Visibility among objects is limited to those in the same venue. Movement can also occur between communities, but there are some limitations on this movement, as we will discuss later. Text communication is within a venue. Speech communication is a private conversation between any two avatars, whether or not they reside in the same venue or even the same community. Agents are objects in a venue that can provide content, for instance in the form of stories, jokes or sound effects, or services like acting as brokers for the exchange of possessions. Figure 2 is a snapshot of the Commercial Square venue in HistoryCity, inhabited by lots of citizens, and containing many buildings into which these citizens can move.

According to its authors, NetEffect is implemented largely in Java, communicating via TCP/IP. NetEffect employs a client/peer-servers/master-server architecture. When a user enters a NetEffect world, he does so through a master server. Based on prior history (e.g., where this user was when he last left the world), the master server selects a peer server that is handling communication for this community, passing its address back to the user's client machine. The client then connects to the community through the assigned peer server. The peer server provides the client with a community description, from which it constructs the 3D world. Users can navigate and interact within this community, or move into other communities through visible portals.

The primary features of NetEffect's protocols that we will discuss are:

- three-tiered networking;
- need to know updating;
- group dead-reckoning;
- dynamic load balancing.

As noted above, NetEffect employs a *three-tiered networking* strategy. The center of the network is a single main server. As part of its start-up, the main server starts a number of peer server nodes, usually one per community. It then passes each of these peer servers a community allocation table, so that each knows its responsibility. Having all servers aware of the location of every community also enables inter-community communication.

The master server is also responsible for maintaining the personal databases of all the world's citizens. Since users always start at the master server, this provides a simple mechanism by which these users can resume wherever they left off in the world.

The periphery of the network consists of the Client nodes on which users reside. Generally, a client node only talks to the peer node that supports the user's current community. When a user moves from one community to another, the client stops communicating with one peer server, and starts communicating with another, unless of course both communities reside on the same peer server. The mechanism’s operation is somewhat analogous to the way cellular telephone systems work.

The master server accepts TCP/IP connections on two ports, one for peer servers and the other for clients. A peer server keeps its connections open to the master service for as long as it is on-line, whereas a client drops its connection once it learns the identity (IP address, port number) of its peer server. Each peer server also accepts connections on two ports, one for other peer servers and the other for clients. Connections made by peer servers are persistent so long as the other server is on-line. Connections made by clients are broken when the client leaves or migrates to a community served by another peer. To migrate to another peer server, the client gets the IP address and port number of the new peer server from its current server. This is possible since every peer server knows about all the others. In fact, migration is only possible if the new peer reports that it can handle the additional load.

While this three-tiered network approach is an effective means of supporting NetEffect's goal to scale up to many users, congestion in a single community, and thus on a single peer server, is still possible. One might expect a situation in which all updates must be broadcast to all residents of a community. To alleviate this problem, NetEffect employs *need-to-know updating*. For instance, a user inside a room does not need to hear conversations or see movements anywhere but within this room. When the user leaves the room, information can quickly be passed that reflects the state of objects in the corridor into which the user just moved.

NetEffect also employs a technique called g*roup dead reckoning*. The idea here is to prevent a continuous flow of positional messages as a user moves throughout a community. ExploreNet uses a destination-based scheme to avoid this, whereas NetEffect uses an interval-based scheme. Users in NetEffect send their positions to their server every *C* seconds, but only if they have moved. The peer server then accumulates this positional data from each user and broadcasts a composite packet to all group members every *P* seconds. *U* and *P* are parameters with which an administrator can experiment. In the case of HistoryCity, they are set to 1 and 1.5 seconds, respectively. The notion of a group is also flexible. The group could be all users in a community, or all users in a venue (e.g., room). HistoryCity uses the former, but NetEffect supports both protocols.

The final technique used to support NetEffect's scalability goal is *dynamic load balancing*. The scheme is rather simple, but effective. The master server periodically checks to see if the world load is balanced across the peer servers. A number of strategies can be employed here, from detailed traffic analysis to just counting the number of attached clients per server. When the master server recognizes an imbalance, it devises a new community allocation across peer servers, and initiates the process of reassignment. During this time, all interactions by clients are stopped, so one hopes that this event is infrequent.

4. Shared VRML Worlds

Starting in 1997, VE developers switched from creating standalone monolithic systems to ones based on plug-ins and Java-enabled browsers. Much of the motivation for this change came form the emergence of the 3D interactive web standard VRML97, and its External Authoring Interface (EAI) specifications which allow 3D objects to be manipulated by external applications that are not written in the VRML language (Web3D, 2000).

VRML is based on the mature and powerful scene graph architecture defined in the Silicon Graphics Performer and Inventor technologies. VRML's shortcomings for large virtual worlds are its relatively low performance, and the lack of a large body of ongoing development in substrate tools such as rendering engines, browsers, optimizers and utility libraries. Low performance manifests itself in numerous ways such as simplistic lighting models, unreliable texture handling and rapid performance degradation as the world's polygon count increases. Performance degradation occurs because there is no effective continuous level-of-detail or load module control mechanism.

Despite VRML's drawbacks, its provision of an Open API rendering layer and its easy access to Java's rich networking capabilities afforded independent developers the tools needed to implement distributed virtual world systems. Further development of the Living Worlds specification (Web3d, 2000) provided guidance to anyone who wished to create shared VRML worlds. We will present detailed descriptions of two such systems, DeepMatrix and VRMINet.

4.1 DeepMatrix
DeepMatrix was developed in 1998 by a collaborative team from Vienna University of Technology, Geometrek in Nyon, Switzerland and Arizona State University (Reitmayr et al, 1999). This system focuses on scalability and is optimized for low bandwidth environments. It has no particular premier world, so we've presented an image in Figure 3 of a representative shared world in which participants can compete in chess.

DeepMatrix is written in Java, runs as an applet within a Java-enables browser, employs a client/server model, and follows but does not completely implement the Living Worlds specification. All code is "semi-open" (Geometrek, 2000), with much of it having been previously implemented in VNet, the first publicly available implementation of the Living Worlds specification (White, 1997). The VRML plug-in provides basic navigation and display of the 3D world. The chat window, list of participants, and the "Ghost mode" and "Disconnect" buttons complete the user interface. Each of these widgets is a standard visual component available in the Java Abstract Windowing Toolkit (AWT). With its simple textual chatting area, this system provides support for games like chess and many other forms of social interaction.

The primary features of DeepMatrix's protocols that we will discuss are:
- loading, rendering, navigation;
- room objects;
- real-time communication.

DeepMatrix does not handle the *loading, rendering* or *navigation* of a world's visual components. Rather, VRML code that defines the visual aspects of DeepMatrix worlds is transported using the HTTP protocol. DeepMatrix just passes information about the location of VRML code for its rooms and client objects by transmitting URLs pointing to VRML files. The VRML plug-in uses HTTP to acquire the files, creates a scene graph from the VRML description and manages all rendering and navigation. DeepMatrix only requires that some simple changes be made to the VRML file so it can learn about gates (portals to other worlds), objects and their behaviors, and so there is a "center" point in the world relative to which it can sense and communicate avatar movement to other clients. This requirement to sense movement is a consequence of the fact that DeepMatrix does not provide its own navigator. DeepMatrix also supports two optional fields, inMotionBehaviour and notInMotionBehaviour, in avatar prototypes. These specify behaviors that are triggered automatically when the avatar is moving or standing still. Their use facilitates articulated motion when the avatar is moving and fidgeting behaviors when it is at rest.

Scalability is addressed on a DeepMatrix system's server by defining *Room* objects. A room defines a partition of the entire world, and clients receive messages only from the Room that they are in. This partitioning of a single large VE or several interconnected VEs dramatically reduces the overall messages that a client machine will receive. Partitioning in this manner can also reduce server load since separate machines can be responsible for different rooms.

The transport layer of DeepMatrix was originally a combination of the UDP and TCP/IP protocols. In these earlier versions, the streamlined UDP protocol was used for *real-time communications* (e.g., state updates of the avatars). Since reporting avatar state changes is the largest single source of network traffic, DeepMatrix's original use of UDP increased response time and used less bandwidth than a pure TCP/IP session. A TCP stream connection was implemented for control messages (e.g., logging in and logging out), managing of Avatar objects and passing of chat messages.

DeepMatrix 1.1 incorporates a major change to the transport layer in that it no longer uses UDP, relying instead on a single TCP stream. The authors state that this change was necessitated by differences in Java implementations within different browsers. Message filtering now removes "stale" messages (ones whose values are superceded by newer updates), in contrast to the UDP approach that can sometimes flood a network with very small incremental updates.

4.2 VRMINet

VRMINet is a system developed in 1999 at the University of Central Florida (Reed et al., 1999). The name "VRMINet" is based on VRML + RMI (Remote Method Invocation), and also reminds one of the occasional use of "vrmin" as a humorous and somewhat pejorative term for geeks who have access to no professionally supported graphics systems. However, the choice of VRML in VRMINet was not driven by lack of alternatives, but by a desire to adhere to a component-based technology, wherein the descriptive language of VRML can be retained while the rendering component might later be replaced.

The premier world hosted in VRMINet is "Caracol Time Travel." This world was authored by an interdisciplinary group of 18 undergraduate students. The team spent two semesters designing and constructing an interactive drama-based educational role-playing game. The game is fashioned after an actual Central American archaeology project being carried out in Belize, under the supervision of Diane and Arlen Chase (Caracol, 1999). The game was designed to teach basic concepts of archaeology and facts about the life of the ancient Maya, to a target audience of middle school students.

A world in VRMINet is a shareable VRML world with Java implemented behaviors. Figure 4 shows a scene in the Caracol world with three avatars (Ana, Pedro and Dr. Smith). Participants are in a cave found at the site, studying a three-dimensional map of the ancient city. In addition to the avatars, this scene is populated with many artifacts that have autonomic behaviors triggered by actions and/or states of the participants. One of the most complex behaviors in the Caracol world involves a flight by a Toucan, with all the avatars being dragged along for the ride from modern times at the archaeological dig into classic Maya times.

VRMINet is written entirely in Java. Software is available for educational use at Reed et al. (1999). The server is a Java application, whereas each participant merely references an html page that contains the client applet. Thus, the standard HTTP web protocol is used to find a world. A user logs in, selecting a role to play in the world. When login is successful, the VRML scene and a Java applet dashboard are displayed. This dashboard allows many forms of interaction, but is not needed for basic navigation, and thus can be minimized at any time.

Initially, the user is controlling the position of the avatar that was selected, and seeing the world through its eyes. If the "out of body" button is clicked, this stations' viewpoint is detached from its avatar and can be pulled back to get an over-the-shoulder view, or maneuvered in such a way as to frame any desired view of the scene. The same button toggles the viewpoint back into the avatar.

An "Online users" window provides the user with a list of the other humans in the world. The available sound menus contain, respectively, sound effects (or utterances such as "hey!"), and short canned speeches (if desired). An "available gestures" menu selects behaviors your avatar knows how to carry out. A "free bodies" window lets you take into your possession any available objects in the scene.

The primary features of VRMINet's protocols that we will discuss are:

- behavior authoring;
- client-server architecture;
- distributed leases;
- remote method invocation;
- performance.

Selecting a free object (which is not an avatar) in the scene either triggers an automatic default behavior or opens a menu, listing the optional behaviors you can specify for this object. This control is very similar to that provided by ExploreNet. However, *behavior authoring* in VRMINet is very flexible, allowing any combination of VRML routes, VRML Script nodes and Java methods to define behaviors. The component nature of this system even allows other scripting languages to be employed. In fact, JPython (a Java friendly variant of Python) has been successfully (actually effortlessly) used in several experimental worlds.

VRMINet employs strict *client-server architecture*. While this is necessitated by Java's security model, it also provides the message-ordering benefits we have discussed before while describing ExploreNet. Thus, all clients receive messages in the order that the server handles them and all clients have the opportunity to process these messages in the same order.

Ground truth in VRMINet is implemented through the concept of *distributed leases*. Java 2 (JDK 1.2) introduced the idea of leased resources in a Java network environment. The notion is similar to a landlord and tenant lease agreement. The landlord (server) grants the tenant (client) a lease on his property (behavior) if it is available. At some point in the future, the lease expires and the tenant/landlord must renegotiate the lease or the property (behavior) is no longer in the hands of the client. The client must vacate the property (stop executing the behavior) and regain the lease before taking possession of the property again. There are no guarantees that the lease will be renewed once it has expired, so the tenant should be ready to give up ownership of the property at any time.

Most IP-based shared virtual world systems are optimized for scalability and network performance only. As you've seen, these systems implement TCP/IP and/or UDP network communication with socket bases and a rigid protocol. The result is a highly scalable but inflexible system that requires protocol extension on client and server components any time new features are added.

The Caracol project (the driving force behind VRMINet) required extensibility and flexibility, was operating in an environment supporting high network speed, and was willing to sacrifice scalability for the sake of the educational experiment. *Remote Method Invocation* (RMI) was chosen as the transport layer due to its acceptable response time and flexibility. RMI's primary attractiveness came from the fact that it distributes objects among connected computers without the need for protocol extensions or explicit serialization. Serialization is the flattening of an object that makes it easy to transport across a network and then reconstruct at the receiving node.

Much concern over the network *performance* of VRMINet was raised during the initial trials with the Caracol project. Player avatars seemed to suffer from network delay when a large load was placed on the system. Observation later showed that delay in the system is largely related to rendering speed on the machine hosting a given client, and to the ways the networking architecture and browser generated motion events that had to be rendered

Allocating more CPU time to the rendering thread, and running the VRML browser under optimal display speed conditions gained some minor performance boost. Further improvement was realized by decreasing the granularity of message passing for movements. On fast machines, the VRML browser produces a large number of movement events for trivial moves. It was observed that for a move of 1 meter, some fast machines could produce over 50 messages that in turn must be sent across the network, executed by the receiving object and then rendered on the video display card. By reducing the granularity of motion, VRMINet decreases the total number of messages passed on the network and thus reduces the number of rendering calls to the receivers' browsers.

Major degradations in performance were noticed in worlds that were built using extremely complex structures, or overabundant simpler structures. The VRML browser uses a great deal of resources when complex models are displayed, which leaves very few resources for the Java Virtual Machine (JVM) to execute. Continuous garbage collection can cause undesirable performance from the JVM and hence, the VRMINet system. Fortunately, Sun's HotSpot continuous garbage collection technology has made this a non-issue. Nonetheless, best results were obtained from worlds that were optimized by polygon count, model count, and texture complexity. Optimizing on these criteria resulted in performance boosts for the overall system.

5. The Future –Component-based VEs

Modern virtual environments have broken away from the monolithic, all-or-nothing closed systems that we described earlier in this chapter, and have moved towards a component-based architecture. VRML, and its ability to plug-in to web browsers, started this evolution. Strangely, this model (which provides no economic incentive for the construction of efficient plug-ins) may lead to VRML's own demise, or at least its relegation to a role as a scene description language, but not a renderer or navigator.

Properly designed VEs can and should allow different rendering systems to be “swapped” for the default system. For example, a system could support external scene description in pure VRML, but internal representation in

Java3D or Fahrenheit. This design requires that the underlying state objects, which represent the 3D on-screen objects, be loosely coupled with the rendering layer. Another requirement is that the 3D content be specified in file formats that all available rendering layers understand.

Just as with the rendering layer, we are now seeing that the network transport layer can be treated as a pluggable component. This means that raw UDP and TCP/IP might be used in some circumstances, and remote method invocation or even tuplespace in others. The choice will depend upon the requirements of the world, and the characteristics of the network environment.

We will briefly visit two systems that are representative of component-based VEs, VRMINet (a second visit), and Bang.

5.1 VRMINet Revisited

We just categorized VRMINet as an example of a VE that supports shared VRML worlds. That categorization, while correct, understates the component nature of this system. VRMINet really has no dependence on standard VRML rendering – its rendering component could easily be replaced. However, from another perspective, the renderer doesn't care about the network transport layer; it just wants to know about transformations and changes of viewpoint that occur. In fact, the system doesn't even depend on its current authoring tools; we previously pointed out the ease with which JPython was incorporated as a scripting language.

5.2 Bang

Bang (Bjarnason, 1999) is a Java-implemented VE that can read VRML descriptions, but maintains complete independence from VRML plug-ins. In fact, its internal scene graph representation is in Java3D, allowing content to be provided in either VRML or serialized Java3D. Bang is also one of the first VEs to use Sun's Jini to support discovering and sharing of worlds. The persistent state of each world is maintained in a tuplespace, using Sun's JavaSpaces implementation. Basic communication is still done using open source from the DeepMatrix and VNet Living Worlds specification implementations.

Bang's use of JavaSpaces provides both persistency (world objects stay around after their creating processes leave) and scalability. The essence of Bang's approach to scalability is to divide the world into 100m×100m×100m sized sectors that are used for loading and unloading objects and geometry. As a user moves from sector to sector, his client registers its presence in the new sector and unregisters its presence from the old one. Notifications of changes in geometry are only sent to those clients registered in the sector in which the change occurred. This approach is in sharp and obvious contrast to other VEs that manage large spaces through logical scenes or communities and portals, rather than physical regions.

Bang also allows users great flexibility in their choice of sound and speech engines. It merely requires that the chosen components implement the appropriate services defined in Java's Media Framework.

6. A Few More Systems of Note

Any page-limited review of IP-based VEs is destined to leave out many systems worthy of detailed study. We are clearly guilty of this crime of omission. The following are just a few of these systems. Readers are encouraged to pursue further study (and recreation) by following the leads we are about to give.

6.1 Non-Commercial Systems

All but one (NetEffects) of the VEs we have already discussed are free, non-commercial systems. Such software typically comes from universities or small groups of diehard developers. Free software, of course, comes with its risks. Over the last decade, most of the giveaways have died away, leaving their users feeling deserted. The recent trend of open software, in which users have access to the source code, as well as executables, is breathing new life into these systems. For instance, VNet, a student project at the University of Waterloo, has not been updated for several years, yet its code is still alive within DeepMatrix, VRMINet and Bang. Among the many other systems that now provide open source, we will highlight two, Bamboo and Virtual Worlds Platform.

6.1.1 Bamboo.

Bamboo was developed at the Naval Postgraduate School, based on their many years of experience in the development and deployment of VEs (Singhal & Zyda, 1999). The goal of this system is to serve as an extensible environment using a generalization of the plug-in metaphor, whereby each plug-in can declare its interdependencies, triggering the loading of required cohorts. Bamboo is in effect a host for new VE technology and a VE system, in that it provides default components built on top of a very lightweight kernel. The ADAPTIVE Communication Environment (ACE), an open-source framework that implements design patterns for concurrent communication software, provides the networking infrastructure for Bamboo. The communication software tasks provided by ACE include event handling, signal handling, interprocess communication, shared memory management, message routing, dynamic (re)configuration of distributed services, concurrent execution and synchronization. Using ACE, Bamboo can employ unicast, multicast and broadcast message passing to build complex protocols, including HLA, the military "High Level Architecture" for distributed interactive simulation. In this regard, Bamboo is the only one that addresses the needs of virtual worlds that must be physically correct.

6.1.2 Microsoft VWorlds.
As in other areas of Internet computing, Microsoft has a major presence with both VE products and active research. The heart of this research is Microsoft's Virtual Worlds Group, which has developed a number of engaging virtual environments, ranging from the playful ComicChat™ to the experimental VWorlds system (Vellon 99). The latter VE integrates seamlessly with Internet Explorer and includes a suite of wizards for avatar and world creation. It also employs a novel event handler that automatically propagates event notifications to an object's contents (e.g., the passengers in a car), container (e.g., a scene) and peers (e.g., other objects in the scene). Source code, executable binaries and sample worlds are available at the organization's web site (Virtual Worlds Group, 2000).

6.2 Commercial Systems
The commercial world of VEs is a brutal one, where a typical system has a very short life span, occasionally followed by a reincarnation as a very specialized product. Some go on forever in a state of limbo, in the sense that they have a web page, but no content. Two of the survivors in this commercial arena are Active Worlds and Blaxxun.

6.2.1 Active Worlds.
Active Worlds provides a free browser, with additional privileges afforded to those who register for a nominal fee. Its server technology requires payment based on intended usage (personal or commercial) and intended size in inhabitants and virtual land. The main Active Worlds server hosts over 600 well-traveled virtual worlds, including a shopping mall. The granddaddy of these is AlphaWorld with over seven hundred thousand users and almost fifty million objects (sizes mentioned here are current as of February 2000). The AlphaWorld territory is slightly larger than the area of the state of California, and growing (Activeworlds, 2000).

6.2.2 Blaxxun.
Blaxxun also provides a free viewer, which is in fact a shared VRML browser that can be used as your primary VRML plug-in. Server and software development technology can be purchased. The primary world hosted by Blaxxun is called Cybertown. As with AlphaWorlds, there is a large virtual space for users to colonize. The metaphor to a real community is strong, in that citizens can develop unique identities, own homes and other possessions, organize activities, and, in general, go about the business of virtual life (Blaxxun, 2000).

7. Conclusions
This chapter has reviewed a number of shared virtual environments designed to interoperate using standard Internet protocols. We have not attempted to be complete, but rather to cover a number of contrasting systems, highlighting features that make them interesting objects of study.

Those wishing more detail are encouraged to visit the web sites to which we have referred. Of course, as with all things ephemeral, we cannot guarantee that these sites will still be alive and up-to-date if you try to access them years (or even months) from now. So, enjoy yourself, but come with a sense of humor and patience.

Figure 1: Autoland: a scene from an ExploreNet world
Figure 2: HistoryCity: a scene from a NetEffect world
Figure 3: Chess Board: a scene from a DeepMatrix world
Figure 4: Caana in modern-day city of Caracol: a scene from a VRMINet world

8. References

Bjarnason, R. V. (1999). *Bang*. [Online]. Available: http://this.is/bang/ [2000, May 15].

Activeworlds (2000). *ActiveWorlds*. [Online]. Available: http://www.activeworlds.com/ [2000, May 15].

Blaxxun (2000). *Blaxxun Contact*. [Online]. Available: http://www.blaxxun.com/ [2000, May 15].

Calvin, J., A., Dickens, Gaines, B., Metzger, P., Miller, M. & Owen, D. (1993). The SIMNET virtual world architecture. *Proceedings of the IEEE VRAIS'93 Conference*, 450-455.

Caracol (2000). *Caracol Archaeological Project*. [Online]. Available: http://www.caracol.org/ [2000, May 15].

Das, T. K., Singh, G., Mitchell, A., Kumar, P. S. & McGee K. (1997). NetEffect: A Network Architecture for Large-scale Multi-user Virtual Worlds. *VRST '97*, September.

Edwards W. & Joy, B. (1999). *Core JINI*. Upper Saddle River, NJ. Prentice Hall.

Freeman, E., Hupfer, S. & Arnold, K. (1999). *JavaSpaces™ Principles, Patterns and Practice*. Reading, MA. Addison Wesley.

Gelernter, D. (1991). *Mirror Worlds: Or the Day Software Puts the Universe in a Shoebox...How It Will Happen and What It Will Mean*. London. Oxford University Press.

Geometrek (2000). *DeepMatrix*. [Online]. Available: http://www.geometrek.com/ [2000, May 15].

Hughes, C. E. & Moshell, J. M. (2000). *ExploreNet*. [Online]. Available: http://www.cs.ucf.edu/ExploreNet/ [2000, May 15].

Hughes, C. E. & Moshell, J. M. (1997). Shared Virtual Worlds for Education: The ExploreNet Experiment. *ACM Multimedia* 5(2), 145-154.

Kent Ridge Digital Labs (2000). *HistoryCity Singapore Project*. [Online]. Available: http://www.historycity.org.sg/ [2000, May 15].

Reed, D., Hughes, C. E. & Moshell, J. M. (1999). *VRMINet*. [Online]. Available: http://www.creat.cas.ucf.edu/VRMINet [2000, May 15].

Reitmayr, G., Carrol, S., Reitemeyer, A. & Wagner, M. G. (1999). DeepMatrix - An Open Technology Based Virtual Environment System. *Visual Computer* 15 (7/8), 395-412.

Singhal, S. & Zyda, M. (1999). *Networked Virtual Environments: Design and Implementation*. Reading, MA. Addison-Wesley.

Sowizral, H., Rushforth, K. & Deering, M. (1997). *The Java 3D API Specification*. Reading, MA. Addison-Wesley.

Stevens, W. R. (1994). *TCP/IP Illustrated, Volume 1*. Reading, MA. Addison-Wesley.

Vellon, M., Marple, K., Mitchell, D. & Drucker, S. (1999). *The Architecture of a Distributed Virtual Worlds System*, Virtual Worlds Group, Microsoft Research. [Online]. Available: http://www.research.microsoft.com/vwg/papers/oousenix.htm [2000, May 15].

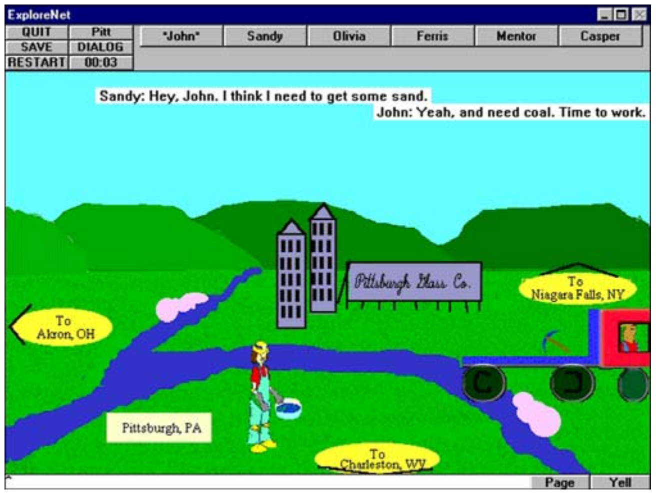
Virtual Worlds Group (2000). *Virtual Worlds Platform*. [Online]. Available: http://www.vworlds.org/ [2000, May 15].

Web3D Consortium (2000). [Online]. Available: http://www.web3d.org/ [2000, May 15].

White, S. F. (1997). *VNet*, Software developed by S. F. White and J. Sonstein at the University of Waterloo, Canada [Online]. Available: http://www.csclub.uwaterloo.ca/~sfwhite/vnet/ [2000, May 15].

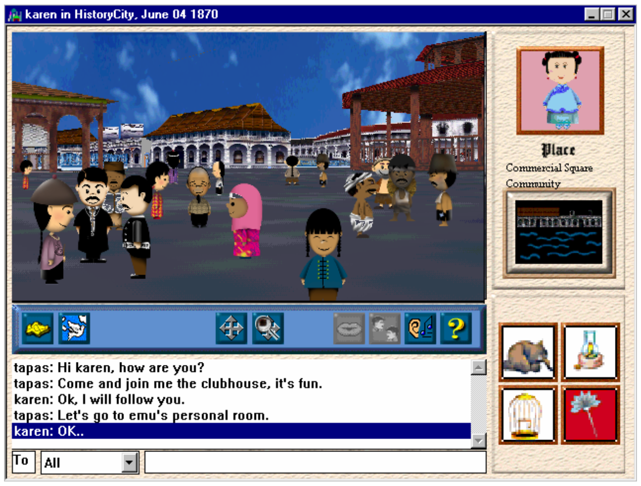
Wyckoff, P., McLaughry, S. W., Lehman, T. J. & Ford, D. A. (1998). TSpaces. *IBM Systems Journal* 37(3).

Figure 1: Autoland: A scene from an ExploreNet world



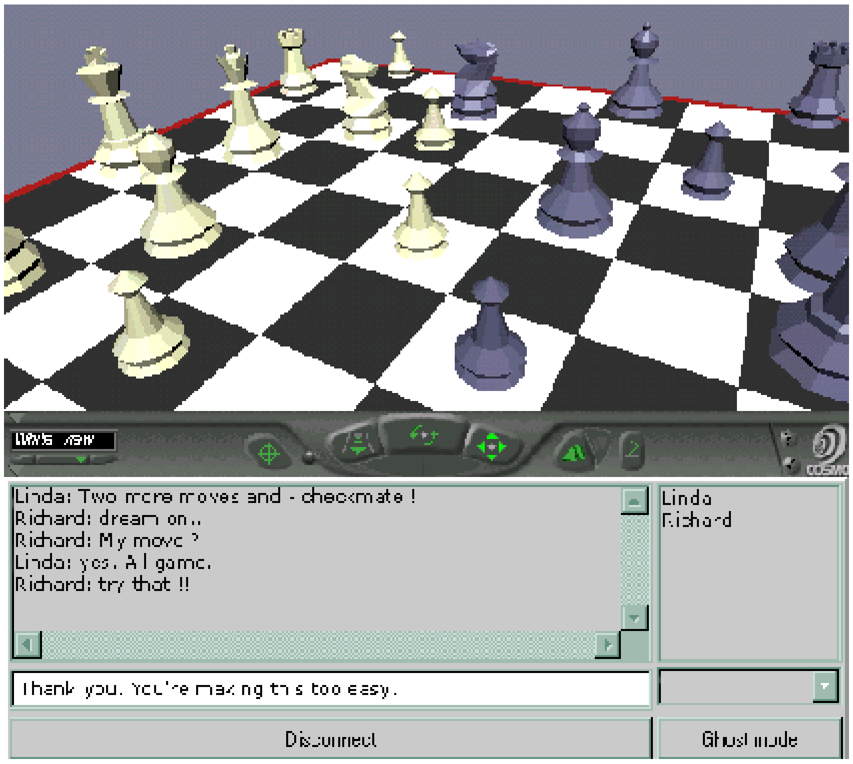
Sandy: Hey, John. I think I need to get some sand.
John: Yeah, and need coal. Time to work.

Figure 2: HistoryCity: A scene from a NetEffect world



tapas: Hi karen, how are you?
tapas: Come and join me the clubhouse, it's fun.
karen: Ok, I will follow you.
tapas: Let's go to emu's personal room.
karen: OK..

Figure 3: Chess Board: A scene from a DeepMatrix world

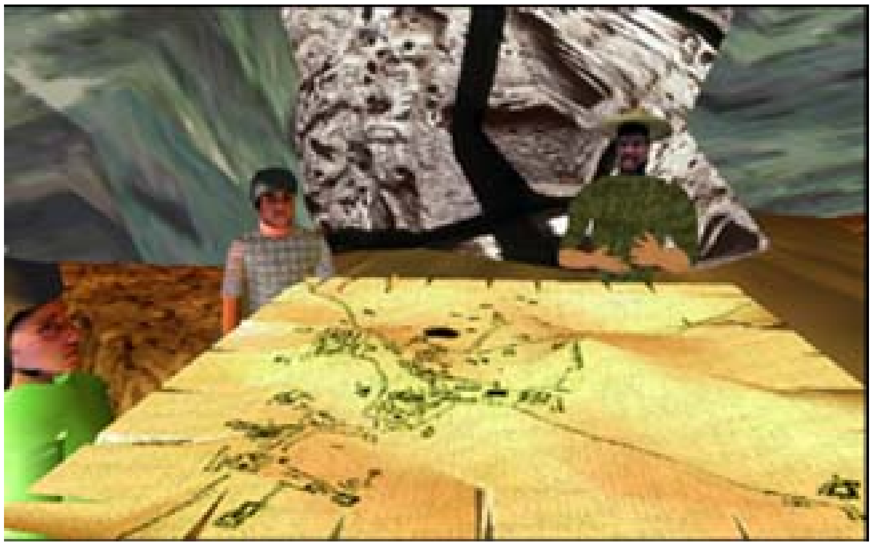


```
Linda: Two more moves and - checkmate !
Richard: dream on..
Richard: My move ?
Linda: yes. AI game.
Richard: try that !!
```

```
Linda
Richard
```

```
Thank you. You're making this too easy.
```

Figure 4: Cave in modern-day Caracol: A scene from a VRMINet world



Keywords

Chapter 19

affordances
API
asynchronous
autonomous behavior
avatar
bandwidth
behavior
browser
client
client-server
communities
component-based software
consistency
Common Object Request Broker Architecture (CORBA)
datagram
dead reckoning
distributed processing
dynamic load balancing
garbage collection
ground truth
Hypertext Transfer Protocol (HTTP)
Internet Protocol (IP)
IP address
Java Virtual Machine (JVM)
JavaSpaces
Jini
leases
load balancing
message routing
navigation
need-to-know
object
object request broker
open software
packet
peer-to-peer
persistent state
plug-in
port
portal
prop
protocol
real-time
relevance clustering
relevance filtering
Remote Method Invocation (RMI)
rendering
scalability
scene
scene graph
server
SIMNET
synchronous

TCP/IP
tiered networking
transient state
Transmission Control Protocol (TCP)
TSpaces
tuplespace
User Datagram Protocol (UDP)
virtual environment (VE)
Virtual Reality Modeling Language (VRML)
virtual world
world
World-Wide Web (WWW)

Other

agent
Distributed Interactive Simulation (DIS)
High-Level Architecture (HLA)
immersion
MUD Object Oriented (MOO)
Multi-User Domain (MUD)
non-player controlled character (NPC)
Virtual Reality (VR)