On System Architectures for Virtual Environments

Dieter Schmalstieg, Michael Gervautz
Institute of Computer Graphics
University of Technology Vienna, Austria
email: dieter|gervautz@cg.tuwien.ac.at

Abstract. In the last years, the number of virtual reality applications has dramatically increased. Despite the huge effort that went into development, little foundation has been laid for system architectures that embody clear concepts and software engineering methodologies. We analyse a number of existing systems that illustrate what we believe are core concepts. A taxonomy is developed that allows to characterize a virtual environment system, and we attempt to formulate the state of the art in the field, proposing a design that we think incorporates the key issues that have been identified.

1. Introduction

It has been pointed out [Appi92][Snow94] that that traditional symbolic user interfaces devices limit the amount of information exchange between user and machine. We interact with the real world through highly developed skills such as our visual system. Providing an interface that involves these skills rather than artificially created interaction techniques has the potential of dramatically increasing the usability of the medium computer.

It appears that some key technologies have recently become cost-effective, and so virtual reality is extremely popular inside and outside the scientific community. To avoid confusion and expectations that come from market hype, researchers have introduced the term *virtual environments* (VE). Virtual environments should be interactive, three-dimensional simulations. They require a number of techniques to be incorporated into a single software system:

- **Multi-sensory Interaction**: The user must be able to interact with the environment in real-time. Although graphics alone are a major factor in VEs, the desired high *level of immersion* is boosted by integrating devices that interact with the other human senses, both for input and output. Output includes visual stereoscopy, 3D spatial audio, and haptic display. Input includes hand and body tracking.
- **Real-time 3-D rendering**: Sensual response from the system must be immediate. Even small delay (in the order of 0.1sec) or inadequate smoothness of the system's output easily destroys the feeling of immersion and therefore the goal of the virtual environment. This places very high demand on the rendering hardware that is by no means adequate for high-quality real-time rendering yet.
- **Simulation**: To step beyond interactive graphics demos that show off the vendor's hardware capabilities, serious simulation engines are necessary that allow to fill the virtual world with meaningful content and create productive applications. The selected simulation model determines how the core architecture of the VE software is structured, and should therefore be treated most carefully.

Virtual Environments		
Animation		Interaction
Rendering	Simulation	Multi-Sensory I/O

Fig. 1: Virtual environment building blocks

These concepts are the major building blocks of virtual environment software [Fig. 1]. Most people associate the term “Virtual Reality” with exotic I/O devices such as head-mounted displays and data gloves. As can be seen, this is just a very small part on the lowest level of the architecture.

Terminology

Unfortunately, the terminology used to describe virtual environments is not well-established. The same expression is used with different meaning, and equivalent concepts are obscured by multiple terms. Not all ambiguities can be resolved.

One of the most abused term is *object*. Every unit of discussion may be called so. We therefore use the term “object” only in the strict object-oriented sense, denoting an encapsulated union of attributes and functionality. The units that “live” in a VE are sometimes called objects, sometimes entities, artefacts or actors. For our discussion, we choose artefact to describe passive compound data-structures, and entities to describe active objects, driven by built-in behavior or by the user. *Participant* is also used to describe users and/or applications, which may or may not correspond to an entity.

Virtual environment (VE) is another expression frequently used ambiguously. Often the term is used in interchangeably with *virtual world* (VW). An *application* may be equivalent to VE and/or VW, or a VE/VW may be composed of multiple applications.

Distributed virtual environments can mean that the computational load is split among several processors. Multi-user systems are always distributed in the sense that every user must have at least a processor that does the rendering. We use term *distributed computation* not to describe support for multiple users but rather functional decomposition and concurrent execution of application-specific code. With *concurrent computation* we mean the simultaneous execution of one particular piece of code by different processors, so that pretermined update information need not be sent over a band-limited network.

Client-server models are often choosen as a network topology for distributed models. Unfortunately, depending on the author the meaning of “client” and “server” is exchangeable. In database transaction systems, the “big” computer is the server that stores all the data, and the “small” PC requests it. In the X-Windows system, the “small” X-Terminal is the server, and the “big” host that runs the application is the client. The same confusion arises in virtual environment network models.

Quality Requirements

The market interest in virtual reality techniques is high and technology is evolving and maturing rapidly. Thus virtual environment research should rather focus on software design to provide a clean separation of tasks and layers. Quality requirements for such a software are similar to those for conventional operating systems, among them are:

- **Simplicity** of use for non-skilled users: It should not only be easy for a naive user to *use* the virtual environment, *creating* new worlds or entities should be just as simple. In particular, changing or extending the world should not require any programming effort (or at most using a very simple scripting language).

- The skilled programmer, however, should have **access to the system on various levels** (scripting for prototyping, system calls for efficient implementation). A well-structured *application programmer's interface* is necessary.
- A **library of ready-made modules** should be supplied. These modules are not part of the system software, but rather building blocks that can be used to implement often-needed functions that the programmer otherwise would have to create himself. For example, not every virtual environment may need collision detection, but if a world designer decides to use it, it should be at hand.
- **Rapid prototyping** to explore new concepts: Virtual environment applications are a very new development, and little is known about the factors that determine the quality of such software. Experimentation is necessary to explore new presentation metaphors, interaction styles etc. A quick development cycle is crucial.
- **Scalability**: the system should become more powerful as faster hardware becomes available. This is non-trivial because fixed data-flow path in the application easily create bottle-necks. The same application should provide high quality on high-end systems but should still be useful on low-cost platforms.
- **Platform independence** and portability: Creating virtual environments is a major design effort. The content of the environment should not be bound to a specific hardware setup. System upgrades should not destroy existing application code.
- Support for **multiple users** and **multi-tasking** (more than one application at a time).
- **Performance**: Most virtual environments suffer from bad performance that hinders or destroys immersion. Unfortunately, clean, layered software components interfer with high performance requirements, because of the overhead involved.

2. Related Work

In this section we attempt to give an overview of virtual environment systems. For the sake of brevity, we limit ourselves to those systems that contribute important software architecture concepts.

BOLIO

An early system with a focus on animation rather than interaction is BOLIO [Zelt89]. BOLIO allows for the rapid construction of animated virtual worlds. In order not to compromise flexibility, no a priory object structure is given. BOLIO supplies geometric primitives; these can be arbitrarily composited using constraints. The system also supplies a set of procedural tools such as inverse kinematics, six-pod locomotion, collision detection or a device driver for a data glove. Using these building blocks, arbitrary environments may be constructed.

The application is single-threaded and monolithic, and so the communication among the entities can be implemented using procedure calls. The animation is executed in a single loop that calculates new position and orientation of every artifact, enforces constraints, and then performs rendering. Dynamic simulation is executed on a per-entity base. The system is often referred to as having initiated some major direction of research in virtual environments.

However, some recent technological advances such as object-oriented modeling were not available at the time.

MR

Being one of the earlier systems, (MR) Minimal Reality is a toolkit for VEs. Shaw et. al. [Shaw92] [Shaw93] first introduced the *decoupled simulation model*. This term describes the functional decomposition of the system, allowing components to execute independent of one another, interating at maximum speed. Thus components cannot slow down each other; for example, the frame rate for rendering should be higher than the update rate of a physically based simulation. MR uses four components: Presentation (rendering), interaction (handling of I/O devices), a geometric model database (that converts data into a form amenable for output), and a computation (simulation) task. These components can be distributed over multiple processors on a network.

RB2

RB2 (Reality Built for Two) [Blan90] appears to be the first widely recognized multi-participant VE. It is a commercial system combining a modelling tool on a Mac with a rendering process on one or more SGI workstations. A server process on a Mac handles I/O devices and broadcasts the scene updates to the rendering hardware. A visual programming system allows convenient specification of behavior. The system is capable of handling changes to behavior constraints and interaction mechanisms at runtime.

VUE

VUE (Veridical User Interface) [Appi92][Code92] is composed of a processes running a VE application, devices and a central dialogue manager. Each device is controlled by a server process. Style and content of the VE are clearly separated. The device servers and processes in the VE application communicate with the dialogue manager, which coordinates concurrent events.

Interaction techniques are represented as dialogues. A dialogue is hierarchically structured into subdialogues. This technique allows dynamic remapping of I/O devices, and interaction techniques by replacing subdialogues in the hierarchy. Further flexibility is achieved by distributing application processes and device servers over multiple processors and dynamically reconfiguring processes.

VB2

Virtuality Builder 2 (VB2) [Gobe93] models an application as a group of processes using asynchronous messages to communicate. A central application process is responsible for the VE and runs the simulation. It also deals with the messages sent by the processes that manage input devices. Interaction paradigms include direct manipulation, gestures and simulated tools.

Dynamic system state is expressed in active variables that can monitor their value as it changes over time. Demons are activated each time a variable changes its value and can be used for procedural simulations such as inverse kinematics. Hierarchical constraints can be defines on the variables, and for each simulation frame a constraint solver enforces the constraints.

DVS

DVS [Ghee94] is a commercially available system. Entities in DVS are called actors and are implemented as an independent processes. Special purpose processes are dedicated to deal with user I/O, rendering, or collision detection. DVS maintains a shared database, and actors keep a local copy of relevant data. A central director propagates updates that one actor makes locally to all other actors that have expressed interested in the change.

SIMNET, NPSNET, and VERN

SIMNET (Simulator network) [Calv93], NPSNET (Naval Postgraduate School Network) [Mace94][Zyda92], and VERN (Virtual Environment Realtime Network) [Blau92], are a family of large-scale combat-training oriented simulations. The aim of these systems is to support a large number of participants; successful demonstrations of around 300 simultaneous players have been reported.

Each local host has a full copy of the database (combat terrain), and the entities communicate over a wide-area network using the standardized protocol DIS (distributed interactive simulation). Updates are sent over the network using IP-mulicast. The message system is limited to information relevant to the training application (packets for position change, granade launch etc.). Dead reckoning is used to locally simulate the remote participants' behavior, updates are sent only if a predefined error treshhold is exceeded. However, participants may join the running simulation at any time, and in the absence of a central server, everyone has to broadcast full state information in regular intervals to inform the new participant. A single participants' software is decomposed into functional units executing in concurrent processes, such as separate rendering and networking processes.

SIMNET was developed to allow special simulator machines running on custom hardware to interconnect. NPSNET is the "low-cost" version of SIMNET, running on standard SGI workstations. VERN is a successor to SIMNET that realises a more rigorous object-oriented design implemented in C++. Otherwise, the architectural concepts are largely the same.

ALICE & DIVER

ALICE [Paus93][Goss93] is a rapid-prototyping system for virtual reality. The system is not intended at the creation of a large and consistent virtual world, but rather at the quick implementation or modification of single simulations by both skilled and novice users. The goal is to bring the VE technology to non-computer scientists that can do research in their domain utilizing the potential of VEs.

ALICE allows the definition of entities and coding of simulation and interaction. Scripting is done in the interpreted object-oriented language Python, that is very simple to learn. Rendering is decoupled from simulation by using a separate rendering process, DIVER. Beyond that, distribution or support for multiple users is not implemented.

DIVE

The DIVE (Distributed Interactive Virtual Environment) system is based on a distributed database containing artifacts. ASrtifacts are passive, but a finite state machine may be attached to an object to give it behavior. The database is partitioned into worlds.

Each participant has a replica of the database. A participant is either a human or an application. Participants may enter or leave the world dynamically. Participants are modelled as processes that communicate by making concurrent updates to the shared database and by sending messages to each other. Internally, each process runs multiple lightweight threads to achieve a functional decomposition. A participant may enter and leave DIVE dynamically. Participants are present in exactly one world, but may switch worlds dynamically.

BRICKNET

BRICKNET [Sing94] allows the creation of virtual environments comprised of objects that embody their graphical, behavioral and network properties. The worlds are defined in the object-oriented interpretive language Starship. Objects can be added, deleted and modified at runtime.

BRICKNET consists of a network of servers that allow clients to connect. Clients cannot change their server, but they can share information across servers. In particular, they can lease out objects to other clients. Clients communicate by messages routed by the servers. A world object acts as a container for other objects (Solids), connected users, and laws for the world. A clients runs one world, but more than one user can be it that world at a time. Because artifacts can be shared, the content of multiple worlds need not be completely disjunct.

VEOS

VEOS (Virtual Environment Operating Shell) [Bric90][Bric94] has a very rigorous structure. It is capable of supporting multiple processors and multiple users with a very large amount of flexibility. A virtual environment is composed of entities. An entity is not only an object participating in the environment, but can also be an I/O-device, or a composition of objects and devices. Composition is hierarchical, so ultimately, an entity is equal to a world. A software structure for the creation of entities is provided, enabling the entity to access a shared database as well as sending and interpreting of messages. Every entity is mapped onto a process. The only means of a process to communicate with another process is by asynchronous messages. Thus it is relatively simple to distribute the communication among several processors for load balancing.

XLISP is used both as a scripting language and as a message format. As an interpreted language, XLISP is very suitable for dynamic configuration of the environment; it is even possible to send program code from one entity to another and make the receiver “learn”.

AVIARY

AVIARY [Snow94] represents the environment as a collection of communicating autonomous objects executing concurrently. Some objects represent artefacts in the virtual environment, other object provide functional abstraction such as collision detection, I/O handling etc. A user object will combine information from a number of input objects, and interpret them.

A world is a container object managing its contained objects. Attributes can also be defined on a per-world base, for example mass for physically based simulation. Objects communicate by asynchronous message passing. They can broadcast messages to other messages in the world via the world object. Messages contain attribute type information, so the content of the message can be interpreted by the receiver. A name server helps objects find one another on the net, and also registers message types so two objects that understand the same message can communicate, even if they were implemented completely independent. However, it is unclear how objects negotiate the semantics of a particular message.

An object may either be implemented as a heavyweight process or as a lightweight object managed by an object server. This saves resources and allows load balancing. Objects can enter and exit the environment at any time, and new object types can be created dynamically. A world manager acts as a broker of available services.

VRML

The Virtual Reality Modeling Language (VRML) [Ball95] is a very recent development still in exploratory stage. It is inspired by the World Wide Web (WWW). WWW servers provide a world-wide distributed information structure. Using the uniform resource locators (URL), a user can access information without being concerned about the physical location.

VRML attempts to extend this idea to virtual reality, thus forming a three-dimensional "cyberspace" [Gibs83]. Server register a part of the cyberspace, similar to a human buying real estate. The *cyberspace protocol* maps 3-D locations to servers, and the world information is transferred to the client (user) for browsing and exploring. Hyperlinks can be used to connect to other servers and to access conventional (WWW) information such as 2-D images. Using VRML, virtual environments become completely platform-independent.

The concept of a markup language that defines context rather than appearance is extented to 3-D scene descriptions in VRML. "Real" 3-D models can be transmitted using a standard 3-D file format such as SGI's Inventor. Markup commands define very high level description such as "a tree", the actual appearance of these artefacts is determined by the local client who consults its internal database. This is necessary because of the limited bandwidth available on long-distance Internet connections. A scripting language is planned to be able to describe and transmit objects with internal behavior.

3. Discussion

Level of distribution

Virtual environment software requires real-time simulation and rendering and is therefore extremely demanding in the computational resources. It is therefore logical to employ some kind of concurrent computation model to resolve the computational bottle-neck; however, system complexity increases as well. Some models of distribution may be distinguished:

- **Single-threaded**: polling of user interface devices, simulation and rendering are all performed in a single loop. Practically all systems have abandoned this model that is simple to implement, but not powerful enough for advanced applications.
- **Single-user, multi-threaded**: This model has been called the decoupled simulation model. The key idea here is to assign parts of the application to dedicated threads (e.g. simulation, rendering) that execute concurrently and synchronize by interprocess communication (shared memory, low-level network protocols). Each dedicated thread maintains its own loop, so the update rates of the threads are independent. This allows maximum performance for each subsystem, which is crucial for demanding applications. For example, the rendering loop must always deliver a minimum of about 15 frames in order not to destroy immersion, however, a physically based simulation may run at a much slower update speed.
- **Multi-user**: Recent systems allow several human participants to be present simultaneously and to interact with each other. The entertainment industry is very fond of this concept, and it is also very promising for computer-supported cooperative work. Since every user has to have a console of his own, a local area network is necessary. The network load may quickly become a problem, because large amounts of data have to be

transmitted in real time. Other non-trivial issues are distribution protection and consistency.

- **Multi-user, geometrically disperse**: In the large, multi-user environments have a somewhat different quality. Large number of user (several hundred) imply a potential low-bandwidth wide-area network that is quite different too handle than a simple close-coupled LAN. Furthermore, a large number of user cannot easily be organized, so the system must have some kind of self-administration mechanism that allows it to function in the presence of all kind of errors. Users must be able to login and leave the running simulation.
- **Multi-world**: In addition to support for multiple users, multi-world environments implement of "parallel universes", that is, more that one world is simulated at a time. The multiple worlds may have completely different contents, rules, and interaction styles. Usually concurrent worlds are each run on a separate processor or at least threads. Migration of the user from one world to another is implemented in the form of *portals*, the VR equivalent of links in hypertext systems. Migration of entities in general from one world to another may or may not be possible; if entities may migrate they must be able to adept to a possibly different set of rules in the new world, which is in general is a hard problem.

Level of flexibility

A VE system architecture can be characterized by its level of flexibility. Existing systems can be assigned to one of the following groups:

- **Monolithic application**: Early systems were mostly concerned with getting things to work at all. Implementation was done in an ad-hoc way without spending too much time on clean separation of layers or concepts. Run-time framework, application and interaction paradigm were tightly integrated to achieve acceptable performance. Some aspects such as the representation of the user are predetermined by the nature of the application.
- **Toolkits**: A simple way to create a flexible software foundation is the toolkit approach. A number of modules (often a class hierarchy) is provided that provides the programmer with high-level tools (such as an efficient rendering engine, intelligent device-drivers for 3D-input or a network management module) from which to build an application. Still, the structure of the application is up to the programmer, although somewhat biased by what is (easily) possible given the set of tools at hand.
- **Fixed-feature environment**: Especially in the low-cost market, virtual environment come in the form of ready-made applications. The feature set is fixed, although usually there is some kind of scripting mechanism. It is more or less predetermined how the user may manipulate the world. The programmer's task is to create the world by specifying appearance and properties of the entities. Usually there is little influence on the interaction style or the mechanisms used to run the simulation. The entities' behavior is fixed, they are passive because the central simulation can calculate everything for them. Entities are just passive data containers.
- **Programmable environment**: Somewhat more powerful is an architecture that allows new entities to be programmed, specifying all their properties and their dynamic behavior. The entities are active, that is, they contain code that allows them to communicate with one another and to actively participate in the simulation. The entities are active in the sense that they not only consist of data that is being manipulated by the

world simulation or the user, but have a set of user-specified behaviors defining the way the object interacts with other entities in the simulation. It should also be relatively simple to alter the default behavior of the world itself (e.g. introduce gravity).

- **Dynamic simulation environment**: Still more powerful is an environment that allows the creation and manipulation of entities and the configuration of the world *at runtime*. Ideally, the virtual environment is simulated at a server that features not only dynamic creation of entities but also accepts the specification of new entity *types* . The concept of a *world* in such a system is true because the content of the simulation evolves over time. Users must be able to log into this environment and leave it after a time. The data in the environment must be persistent. This is of course the most demanding form of virtual environment, and all kinds of incompatibility problems may arise from the unspecified and unbound nature of such a simulation.

Distribution and flexibility are neither completely orthogonal, nor are they an exhaustive characterization, but they form a taxonomy that describes most aspects of the current systems reasonably well.

State of the art

A virtual environment should support **multiple users**. Therefore it is inherently **distributed**. Conceptually, a virtual environment can be viewed as a **database** shared over a network (LAN or WAN). The actual implementation is very often done by full or partial local **replication**.

The communication system is **message-based**. An object broadcasts local updates via messages, usually with some kind of IPC mechanism, because most systems are distributed and heterogeneous. Additionally, messages that do not contain information but carry some intention (e.g. requests) are sent from one entity to another.

Efficiency is increased by (1) allowing entities to **register interest** in a particular category of message, so that the number of messages an entity receives is decreased; (2) performing some sort of **concurrent computation** to predict the behavior of remote entities (e.g. dead reckoning).

From the environment point of view, applications and users should be treated the same, here they are called **entities**. **Decoupled simulation** can be used to achieve distribution within entities (e.g., separate rendering and communication into lightweight threads).

The virtual environment is decomposed into **worlds**. A world is decomposed into **artifacts**. Worlds need not be disjoint. Active entities, in particular users, can **migrate** from one world to another, either via **portals** or by leaving the volume of one world and entering another.

Applications can be different from artifacts, but then artifacts are passive. In this case, artifacts can be equipped with some kind of behavioral mechanism, but this complicates the implementation. Often **object-oriented model** is used, so passive artifacts and behavior can be encapsulated, hiding an application inside an artifact.

Some applications are better implemented on a per-world base (such as gravity). In a flexible system, artifacts and worlds are independent, and so their requirements can be contradictory. A proposal is to have the world maintain some state information for contained artifacts [Hubb93], but this is not a universal solution. An alternative solution [Bric94] is to make artifacts programmable and hierarchical, so that artifacts, applications, and worlds are the same.

A powerful system does not come in the form of a toolkit that can be used to implement a particular application, but as a server that is permanently running, similar to a WWW or MUD server. This server (or servers) can easily become performance bottlenecks. However, if the network is peer-to-peer instead, state cannot be kept in a server. The environment is running as long as there are any participants. In regular intervals, full state information has to be broadcast by all entities to inform new participants, which can create a high network load.

In addition to allowing entities to enter and leave the environment at will and migrate between worlds, a system is powerful if it allows **creation and deletion of artifacts at runtime**, and the **definition of new artifacts at runtime**. A system that is built completely in C cannot support this well enough. An **interpreted scripting language** is needed. The power of the scripting language largely determines the power of the environments that can be built with a particular system.

For efficiency reasons, it is convenient to have both **light- and heavyweight objects** at hand. Lightweight objects can be scripted or otherwise attributed with some kind of simple behavior, whereas heavyweight object run in separate processes, driving computationally intensive parts of the system. Dedicated objects such as I/O device drivers are examples. These objects are not artifacts, but it is convenient if they blend with the other simulation objects.

If the architecture is flexible enough, it may support **load balancing**. This is hard, because it is hard to merge automatic load balancing (requiring a uniform communication mechanism) and data path shortcuts implemented for performance reasons, e.g. a direct fed from a head tracker to rendering.

4. Conclusion

We have presented the current state of the art in virtual environment system architectures. Virtual environments are a relatively new field and pose high demands on current hardware and software. Multi-sensory interaction, high-quality 3-D rendering, and flexible dynamic simulation must be combined at real-time rates. Distributed heterogeneous networks are needed to deal with the high computational load and support multiple simultaneous users.

Virtual environments combine techniques from computer graphics, networks, modeling, distributed computation, user interfaces and human-computer interaction. Quality requirements include simplicity both in use and development, rapid prototyping, scalability, platform independence, support for multiple users and applications, and high performance. Early ad-hoc designs are not sufficient for today’s virtual reality applications. These requirements can only be met using a structured approach.

From a discussion of the most influential existing systems, we identity two taxonomies that characterize virtual environments. These are the level of distribution and the level of flexibility. State of the art systems are both highly concurrent and flexible. They support multiple users on a wide-area networks and multiple worlds. Participants can dynamically enter and leave the environment and migrate between worlds. New artifacts and artifact types can be defined at runtime. Arbitrary behavior can be attributed to artifacts.

Performance scalability requires load balancing and local prediction. Decoupled simulation allows concurrent execution of functionally independent modules. The underlying network communication model determines the flow of information through the system and has a large impact on the implementation.

As a conclusion, a system architecture that meets all or most of the issued that are addressed in this paper should be well-equipped for tomorrow’s virtual reality application.

References

[Appi92] P. A. Appino, J. B. Lewis, L. Koved, D. T: Ling, D. A. Rabenhorst, C. F. Codella: An Architecture for Virtual Worlds. Presence, Vol. 1, No. 1, pp. 1-17 (1992)

[Ball95] G. Ball, A. Parisi, M. Pesce: VRML Draft Specification 1.0. Technical Report, http://www.eit.com/vrml/vrmlspec.html (1995)

[Blan90] C. Blanchard, S. Burgess, Y. Harvill, J. Lanier, A. Lasko, M. Oberman, M. Teitel: Reality Built for Two: A Virtual Reality Tool. SIGGRAPH Symposium on 3D Interactive Graphics, pp. 35-38 (1990)

[Blau92] Blau B., Hughes C. E., Moshell J. M., Lisle C.: Networked virtual environments. Computer Graphics (1992 Symposium on Interactive 3D Graphics) , Vol. 25, No. 2, pp. 157-160 (1992)

[Bric90] W. Bricken: Virtual Environment Operating Shell: Preliminary Functional Architecture. Technical Report TR-HITL-M-90-2 (1990)

[Bric94] W. Bricken, G. Coco: The VEOS Project. Presence, Vol. 3, No. 2, pp. 111-129 (1994)

[Calv93] J. Calvin, A. Dicken, B. Gaines, P. Metzger, D. Miller, D. Owen: The SIMNET virtual world architecture. Proceedings of VRAIS, pp. 450-455 (1993)

[Carl93a] C. Carlsson, O. Hagsand: DIVE - A platform for multi-user virtual environments. Computers and Graphics, Vol. 17, No. 6, pp. 663-669 (1993)

[Carl93b] C. Carlsson, O. Hagsand: DIVE - A multi-user virtual reality system. Proceedings of VRAIS, pp. 394-400 (1993)

[Code92] C. Codella et al.: Interactive Simulation in a Multi-Person Virtual World. Proceedings of SIGCHI, pp. 329-334 (1992)

[Ghee94] S. Ghee, J. Naughton-Green: Programming Virtual Worlds. SIGGRAPH Course, No. 17 (1994)

[Gibs83] W. Gibson: Neuromancer. Novel (1983)

[Gobe93] E. Gobetti, J. Balaguer, D. Thalmann: VB2 - An Architecture For Interaction In Synthetic Worlds. Proceedings UIST, pp. 167-178 (1993)

[Goss93] R. Gossweiler, C. Long, S. Koga, R. Pausch: DIVER: A Distributed Virtual Environment Research Platform. Symposium on Reasearch Frontiers in Virtual Reality (1993)

[Hubb93] R. Hubbold, A. Murta, A. West, T. Howard: Design issues for virtual reality systems. 1st Eurographics Workshop on Virtual Environments (1993)

[Mace94] M. R. Macedonia, M. J. Zyda, D. R. Pratt, P. T. Barham, S. Zeswitz: NPSNET: A Network Software Architecture for Large-Scale Virtual Environment. Presence, Vol. 3, No. 4, pp. 265-287 (1994)

[Paus93] R. Pausch, M. Conway, R. DeLine, R. Gossweiler, S. Miale: Alice and DIVER: A software architecture for building virtual environments. Proceedings of INTERCHI, pp. 13-14 (1993)

[Pesc95] M. Pesce, P. Kennard, A. Parisi: Cyberspace. Technical report, http://vrml.wired.com/concepts/pesce-www.html (1995)

[Ragg95] D. Raggett: Extending WWW to support Platform Independent Virtual Reality. Technical Report, http://vrml.wired.com/concepts/ragett.html (1995)

[Shaw92] C. Shaw, J. Liang, M. Green, Y. Sun: The Decoupled Simulation Model for VR Systems. Proceedings of SIGCHI, pp. 321-328 (1992)

[Shaw93] C. Shaw, M. Green: The MR toolkit peers package and experiment. Proceedings of VRAIS, pp. 463-469 (1993)

[Sing94] G. Singh, L. Serra, W. Png, Hern Ng: BrickNet: A Software Toolkit for Network-Based Virtual Worlds. Presence, Vol. 3, No. 1, pp. 19-34 (1994)

[Snow94] D. Snowdon, A. West: AVIARY: Design Issues for Future Large-Scale Virtual Environments. Presence, Vol. 3, No. 4, pp. 288-308 (1994)

[Zelt89] D. Zeltzer, S. Pieper, D. J. Sturman: An Integrated Graphical Simulation Plattform. Proc. Graphics Interface ‘89, pp. 266-274 (1989)

[Zyda92] M. Zyda, D. Pratt, J. Monahan, K. Wilson: NPSNET: Constructing a 3D Virtual World. SIGGRAPH Symposium on 3D Interactive Graphics, pp. 147 (1992)