The VEOS Project[1]

William Bricken and Geoffrey Coco

Abstract

The Virtual Environment Operating Shell (VEOS) was developed at the Human Interface Technology Laboratory as software infrastructure for the lab's research into virtual reality. VEOS was designed from scratch to provide a comprehensive and unified management facility to support generation of, interaction with, and maintenance of virtual environments. VEOS emphasizes rapid prototyping, heterogeneous distributed computing, and portability. This article presents the architecture of VEOS in the context of the generic functionality of VR systems, the nature of presence, and the varieties of semantics within a virtual environment. We then discuss the design, philosophy and implementation of VEOS in depth. Within the Kernel, the shared database transformations are pattern-directed, communications are asynchronous, and the programmer's interface is LISP. An entity-based metaphor extends object-oriented programming to systems-oriented programming. Entities provide first-class environments and biological programming constructs such as `perceive`, `react`, and `persist`. The organization, structure, and programming of entities is discussed in detail. The article concludes with a description of the applications which have contributed to the iterative refinement of the VEOS project

[1] The HITL software development program is a group effort to create a satisfying software infrastructure for the entire lab. The following discussion is based upon a conceptual VR system architecture developed by Bricken over the last decade and grounded in an implementation by Coco over the last three years (Bricken, 1991a). VEOS supports parallel development of lab applications, technology demonstrations, thesis projects, and software interaction tools (Coco, 1993). Dav Lion, Colin Bricken, Andy MacDonald, Marc Cygnus, Dan Pirone, Max Minkoff, Brian Karr, Daniel Henry, Fran Taylor and several others have made significant contributions to the VEOS project. The VEOS project was supported by members of the HITL Industrial Consortium.

1. Introduction

Computer technology has only recently become advanced enough to solve the problems it creates with its own interface. One solution, *virtual reality* (VR), immediately raises fundamental issues in both semantics and epistemology.

Broadly, virtual reality is that aspect of reality which people construct from information, a reality which is potentially orthogonal to the reality of mass. Within computer science, VR refers to interaction with computer generated spatial environments, environments constructed to include and immerse those who enter them.

VR affords non-symbolic experience within a symbolic environment.

Since people evolve in a spatial environment, our knowledge skills are anchored to interactions within spatial environments. VR design techniques, such as scientific visualization, map digital information onto spatial concepts. When our senses are immersed in stimuli from the virtual world, our minds construct a closure to create the experience of inclusion. *Participant inclusion* is the defining characteristic of VR.[2] Inclusion is measured by the degree of *presence* a participant experiences in a virtual environment.

We currently use computers as symbol processors, interacting with them through a layer of symbolic mediation. The computer user, just like the reader of books, must provide cognitive effort to convert the screen's representations into the user's meanings. VR systems, in contrast, seek to provide interface tools which support natural behavior as input and direct perceptual recognition of output. The idea is to access digital data in the form most easy for our comprehension; this generally implies using representations that look and feel like the thing they represent. A physical pendulum, for example, might be represented by an accurate three dimensional digital model of a pendulum which supports direct spatial interaction and dynamically behaves as would an actual pendulum.

Immersive environments redefine the relationship between experience and representation, in effect rendering the syntax-semantics barrier transparent. Reading, writing, and arithmetic are hidden from the computer interface, replaced by direct, non-symbolic environmental experience.

Before we can explore the deeper issues of experience in virtual environments, we must develop an infrastructure of hardware and software to support "tricking the senses"[3] into believing that representation is reality. The VEOS project was designed to provide a rapid prototyping infrastructure for exploring virtual environments. In contrast to basic research in computer science, this project attempted to synthesize known

[2] Participation within information is often called *immersion*.
[3] The description of VR as techiques which *trick* the senses embodies a cultural value: somehow belief in digital simulation is not as legitimate as belief in physical reality. The VR paradigm shift directly challenges this view. The human mind's ability to attribute equal credibility to Nature, television, words, dreams and computer-generated environments is a *feature*, not a bug.

techniques into a unique functionality, to redefine the concept of interface by providing interaction with environments rather than with symbolic codes.

This chapter presents some of the operating systems techniques and software tools which guide the early development of virtual reality systems at the University of Washington Human Interface Technology Lab. We first describe the structure of a VR system. This structure is actually the design basis of the Virtual Environment Operating Shell (VEOS) developed at HITL. Next, the goals of the VEOS project are presented and the two central components of VEOS, the Kernel and FERN, are described. The chapter concludes with a description of entity-based programming and of the applications developed at HITL which use VEOS. As is characteristic of VR projects, this chapter contains multiple perspectives, approaching description of VEOS as a computational architecture, as a biological/environmental modeling theory, as an integrated software prototype, as a systems-oriented programming language, as an exploration of innovative techniques and as a practical tool.

2. Component Technologies

Computer-based VR consists of a suite of four interrelated technologies:

Behavior Transducers:	hardware interface devices
Inclusive Computation:	software infrastructure
Intentional Psychology:	interaction techniques and biological constraints
Experiential Design:	functionally aesthetic environments

Behavior transducers map physically natural behavior onto digital streams. *Natural behavior* in its simplest form is what two-year-olds do: point, grab, issue single word commands, look around, toddle around. Transducers work in both directions, from physical behavior to digital information (sensors such as position trackers and voice recognition) and from digital drivers to subjective experience (displays such as stereographic monitors and motion platforms).

Inclusive computation provides tools for construction of, management of, and interaction with inclusive digital environments. Inclusive software techniques include pattern-matching, coordination languages, spatial parallelism, distributed resource management, autonomous processes, inconsistency maintenance, behavioral entities and active environments.

Intentional psychology seeks to integrate information, cognition and behavior. It explores structured environments that incorporate expectation as well as action, that reflect imagination as well as formal specifications. It defines the interface between the digital world and ourselves: our sensations, our perceptions, our cognition, and our intentions. Intentional psychology incorporates physiological models, performance metrics, situated learning, multiple intelligences, sensory cross-mapping, transfer effects, participant uniqueness, satisficing solutions, and choice-centered computation.

Experiential design seeks to unify inclusion and intention, to make the virtual world feel good. The central design issue is to create particular inclusive environments out of the infinite potentia, environments which are fun and functional for a participant. From the perspective of a participant, there is no interface, rather there is a world to create (M. Bricken, 1991). The conceptual tools for experiential design may include wands, embedded narrative, adaptive refinement, individual customization, interactive construction, multiple concurrent interpretations, artificial life, and personal, mezzo and public spaces.

Taxonomies of the component technologies and functionalities of VR systems have only recently begun to develop (Naimark, 1991; Zeltzer, 1992; Robinett, 1992), maturing interest in virtual environments from a pre-taxonomic phenomenon to an incipient science. Ellis (1991) identifies the central importance of the environment itself, deconstructing it into content, geometry, and dynamics.

VR unifies a diversity of current computer research topics, providing a uniform metaphor and an integrating agenda. The physical interface devices of VR are similar to those of the teleoperation and telepresence communities. VR software incorporates real-time operating systems, sensor integration, artificial intelligence, and adaptive control. VR worlds provide extended senses, traversal of scale (size-travel), synesthesia, fluid definition of self, super powers, hyper sensitivities, and meta physics. VR requires innovative mathematical approaches, including visual programming languages, spatial representations of mathematical abstractions, imaginary logics, void-based axiomatics, and experiential computation. The entirely new interface techniques and software methodologies cross many disciplines, creating new alignments between knowledge and activity.

VR provides the cornerstone of a new discipline: **Computer Humanities**.

3. The Structure of a VR System

As a technology matures, the demands on the performance of key components increase. In the case of computer technology, we have passed through massive mainframes to personal computers to powerful personal workstations. A growth in complexity of software tasks has accompanied the growth of hardware capabilities. At the interface, we have gone from punch cards to command lines to windows to life-like simulation. Virtual reality applications present the most difficult software performance expectations to date. VR challenges us to synthesize and integrate our knowledge of sensors, databases, modeling, communications, interface, interactivity, autonomy, human physiology, and cognition -- and to do it in real-time.

VR software attempts to restructure programming tools from the bottom up, in terms of *spatial, organic models.*[4] The primary task of a virtual environment operating system is to make computation transparent, to empower the participant with *natural interaction.* The technical challenge is to create mediation languages which enforce rigorous mathematical computation while supporting intuitive behavior. VR uses spatial interaction as a mediation tool. The prevalent textual interface of command lines and

[4] Later in this chapter, we outline the entity model implemented in VEOS which provides both spatial and organic programming metaphors.

pull-down menus is replaced by physical behavior within an environment. Language is not excluded, since speech is a natural behavior. Tools are not excluded, since we handle physical tools with natural dexterity. The design goal for natural interaction is simply *direct access to meaning*, interaction not filtered by a layer of textual representation. This implies both eliminating the keyboard as an input device, and minimizing the use of text as output.

3.1 Functional Architecture

Figure 1 presents a functional architecture for a generic VR system; Figure 1 is also the architecture of VEOS. The architecture contains three subsystems: transducers, software tools, and computing system. Arrows indicate the direction and type of dataflow.[5] Participants and computer hardware are shaded with multiple boxes to indicate that the architecture supports any number of active participants and any number of hardware resources.[6] Naturally, transducers and tools are also duplicated for multiple participants.

This functional model, in addition to specifying a practical implementation architecture, provides definition for the essential concepts of VR.

The ***behavior and sensory transducing subsystem*** (labeled participant, sensors and display) converts natural behavior into digital information and digital information into physical consequence. Sensors convert our actions into binary-encoded data, extending the physical body into the virtual environment with position tracking, voice recognition, gesture interfaces, keyboards and joysticks, midi instruments and bioactivity measurement devices. Displays provide sensory stimuli generated from digital models and tightly coupled to personal expectations, extending the virtual environment into the realm of experience with wide-angle stereo screens, surround projection shells, head-mounted displays, spatial sound generators, motion platforms, olefactory displays, and tactile feedback devices.

[5] In actual implementations, the operating system is involved with all transactions. Figure 1 illustrates direct dataflow paths, hiding the fact that all paths are mediated by the underlying hardware.

[6] Existing serial computers are not designed for multiple concurrent participants or for efficient distributed processing. One of the widest gaps between design and implementation of VR systems is efficient integration of multiple subsystems. VEOS is not a solution to the integration problem, nor does the project focus on basic research toward a solution. Real-time performance in VEOS degrades with more than about ten distributed platforms. We have experimented with only up to six interactive participants.

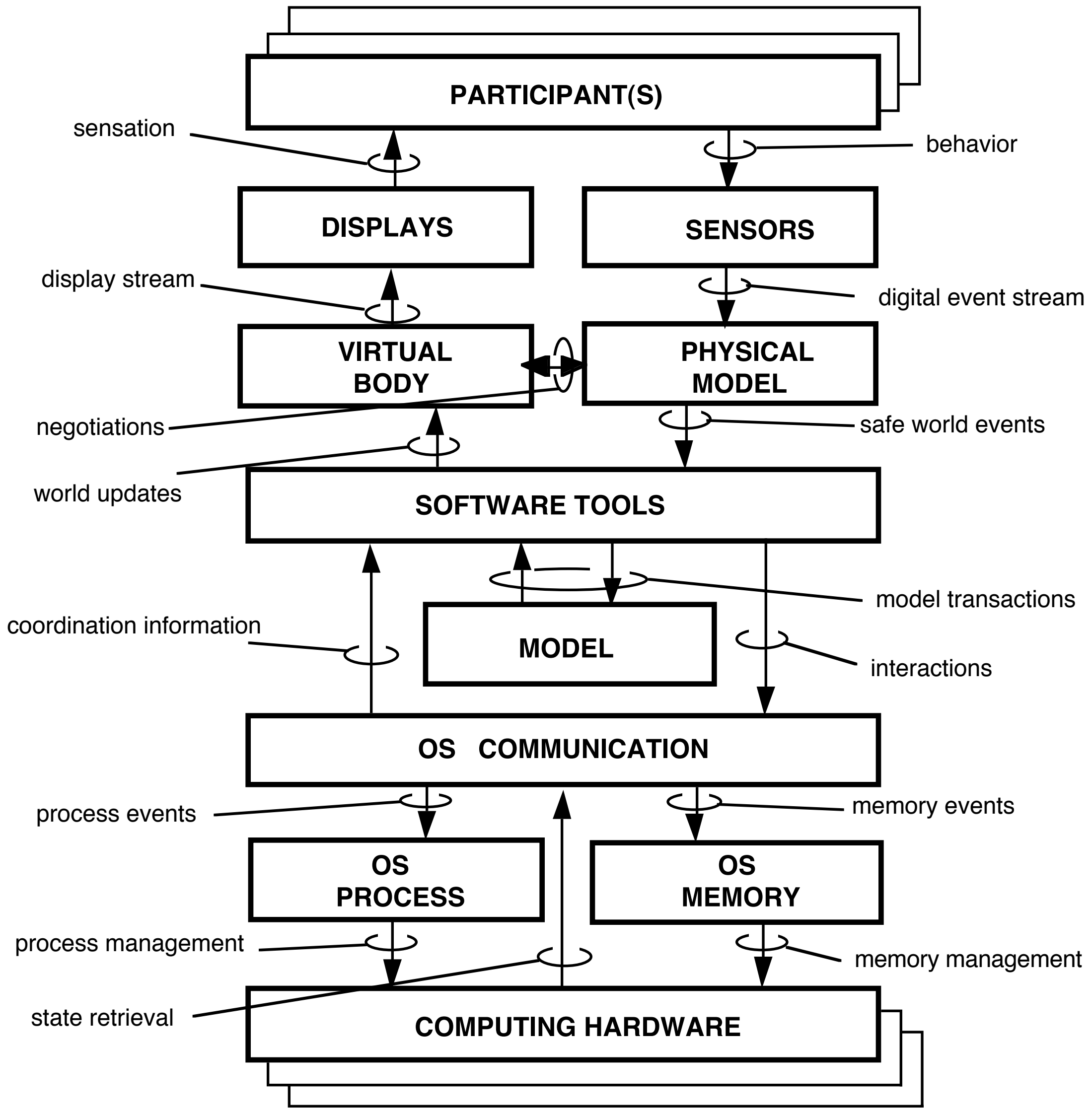

Figure 1. VEOS System Architecture.

The behavior transducing subsystem consists of these three components:

The participant. VR systems are designed to integrate the human participant into the computational process. The participant interprets the virtual world perceptually and generates actions physically, providing human transduction of imagination into behavior.

Sensors (input devices) convert both the natural behavior of the participant and measurements of events occurring in the physical world into digital streams. They transduce physical measurement into patterned representation.

Displays (output devices) convert the digital model expressed as display stream instructions into subjective sensory information perceived as sensation by the participant. They physically manifest representation.

The ***virtual toolkit subsystem*** (the physical model, virtual body, software tools and model) coordinates display and computational hardware, software functions and resources, and world models. It provides a wide range of software tools for construction of and interaction with digital environments, including movement and viewpoint control; object inhabitation; boundary integrity; editors of objects, spaces and abstractions; display, resource and time management; coordination of multiple concurrent participants; and history and statistics accumulation.

The virtual toolkit subsystem consists of four software components:

The **physical model** maps digital input onto a realistic model of the participant and of the physical environment the participant is in. This model is responsible for screening erroneous input data and for assuring that the semantic intent of the input is appropriately mapped into the world database.

The **virtual body** customizes effects in the virtual environment (expressed as digital world events) to the subjective display perspective of the participant.[7] The virtual body is tightly coupled to the physical model of the participant in order to enhance the sensation of presence. Differences between physical input and virtual output, such as lag, contradiction, and error, can be negotiated between these two components of the body model without interacting with the world model. The physical model and the virtual body comprise a *participant system* (Minkoff, 1993).

Virtual world **software tools** program and control the virtual world, and provide techniques for navigation, manipulation, construction, editing, and other forms of participatory interaction. All transactions between the model and the system resources are managed by the tool layer.

The virtual world **model** is a database which stores world state and the static and dynamic attributes of objects within the virtual environment. Software tools access and assert database information through model transactions. During runtime, the database undergoes constant change due to parallel transactions, self-simplification, canonicalization, search-by-sort processes, process demons, and function evaluations. The database is better viewed as a turbulent fluid than as a stable crystal.

The ***computational subsystem*** (the operating system and hardware) customizes the VR software to a particular machine architecture. Since machine level architecture dictates

[7] A graphics rendering pipeline, for example, transforms the world coordinate system into the viewpoint coordinate system of the participant. Since renderers act as the virtual eye of the participant, they are part of the participant system rather than part of the operating system.

computational capacity and operating system architecture dictates computational efficiency, this subsystem is particularly important for ensuring real-time performance, including update rates, complexity and size of worlds, and responsiveness to participant behavior.

The computational subsystem consists of these components:

> The operating system **communications** management (messages and networking) coordinates resources with computation. The intense interactivity of virtual worlds, the plethora of external devices, and the distributed resources of multiple participants combine to place unusual demands on communication models.
>
> The operating system **memory** management (paging and allocation) coordinates data storage and retrieval. Virtual worlds require massive databases, concurrent transactions, multimedia datatypes, and partitioned dataspaces.
>
> The operating system **process** management (threads and tasks) coordinates computational demands. Parallelism and distributed processing are prerequisite to VR systems.
>
> The computational **hardware** provides digital processing specified by the operating system. Machine architectures can provide course and fine grain parallelism, homogeneous and heterogeneous distributed networks, and specialized circuitry for real-time performance.

Operating systems also manage input and output transactions from physical sensors and displays. Some data transactions (such as head position sensing used for viewpoint control) benefit from having minimal interaction with the virtual world. Real-time performance can be enhanced by specialized software which directly links the input signal to the output response.[8]

3.2 Presence

Presence is the impression of being within the virtual environment. It is the suspension of disbelief which permits us to share the digital manifestation of fantasy. It is a reunion with our physical body while visiting our imagination.

The traditional *user interface* is defined by the boundary between the physical participant and the system behavior transducers. In a conventional computer system, the behavior transducers are the monitor and the keyboard. They are conceptualized as

[8] The Mercury Project at HITL, for example, implements a participant system which decouples the performance of the behavior transducing subsystem from that of the virtual world through distributed processing. Even when complexity slows the internal updates to the world database, the participant is still delivered a consistently high frame rate.

specific tools. The user is an interrupt. In contrast, *participant inclusion* is defined by the boundary between the software model of the participant and the virtual environment. Ideally the transducers are invisible, the participant feels like a local, autonomous agent with a rendered form within an information environment.

The degree of presence achieved by the virtual world can be measured by the ease of the subjective shift on the part of the participant from attention to interface to attention to inclusion. For example, a standard mouse-driven cursor feels more like an extension of self than does a pull-down menu because the cursor is always active (it does not require a selection to activate), it is one-to-one with the movement of our hand, and it does not have textual mediation. The cursor has more presence. When we attach a force-feedback system to the cursor, so that we can feel the edges of windows, presence is significantly enhanced, the cursor feels more like we are holding it.

An *interface* is a boundary which both separates and connects. Traditional interface separates us from direct experience while connecting us to a representation of information (the semantics-syntax barrier). The keyboard connects us to a computational environment by separating concept from action, by sifting our intention through a symbolic filter.

Interface provides access to computation by first objectifying processes and then displaying the objective encodement. Displays, whether command line, window or desktop, present tokens which we must interpret through reading. Current multimedia video, sound and animation provide images we can watch and interact with within the two dimensional space of the monitor. VR provides three dimensional interaction we can experience.

Conventionally we speak of the "software interface" as if the locale of human-computer interaction were somehow within the software domain. The human interface, the boundary which both separates and connects us, is our skin. *Our bodies are our interface.* VR inclusion accepts the entirety of our bodily interface, internalizing interactivity within an environmental context.

The architectural diagram in Figure 2 is composed of three nested inclusions (physical, digital, virtual). The most external is physical reality, the participant's physical body on one edge, the computational physical hardware on the other. All the other components of a VR system (software, language, virtual world) are contained within the physical. Physical reality *pervades* virtual reality.[9] For example, we continue to experience physical gravity while flying around a virtual environment.

[9] The apparent dominance of physical reality is dependent on how we situate our senses. That is to say, physical reality is dominant only until we close our eyes. Situated perception is strongly enhanced by media such as radio, cinema and television, which invite a refocusing into a virtual world. The objective view of reality was reinforced last century by print media which presents information in an objectified, external form. Immersive media undermine the dominance of the physical simply by providing a different *place* to situate perception.

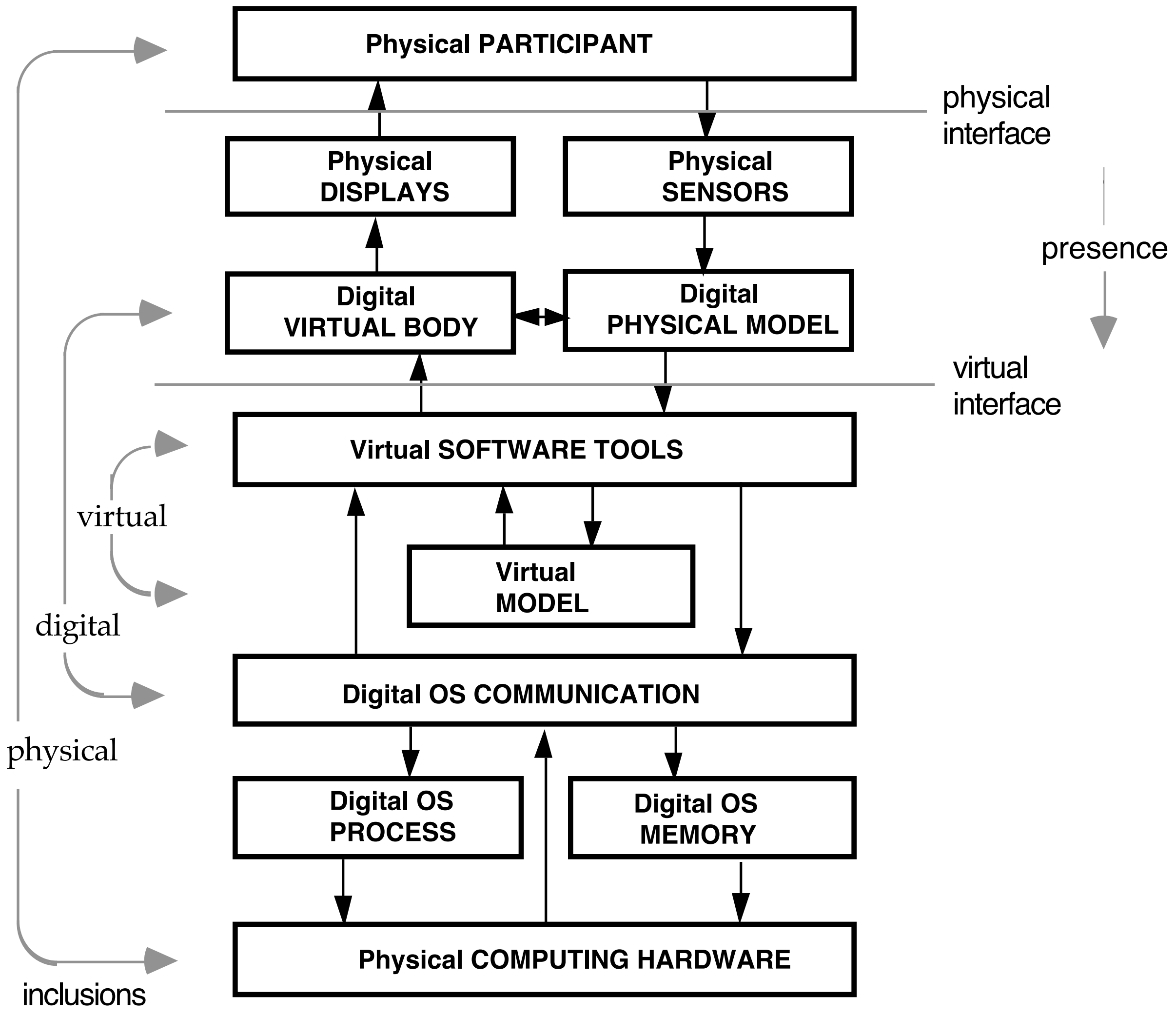

Figure 2: Presence and Inclusion

One layer in from the physical edges of the architecture are the software computational systems. A participant interfaces with behavior transducers which generate digital streams. The hardware interfaces with systems software which implements digital computations. Software, the *digital reality*, is contained within physical reality, and in turn, pervades virtual reality.

The innermost components of the architecture, the virtual world tools and model, form the virtual reality itself.[10] Virtual software tools differ from programming software tools in that the virtual tools provide a non-symbolic look-and-feel. Virtual reality seemlessly mixes a computational model of the participant with an anthropomorphized

[10] To be manifest, VR also requires a participant.

model of information. In order to achieve this mixing, both physical and digital must pervade the virtual.

Humans have the ability to focus attention on physicality, using our bodies, and on virtuality, using our minds. In the VR architecture, the participant can focus on the physical/digital interface (watching the physical display) and on the digital/virtual interface (watching the virtual world). Although the digital is necessary for both focal points, VR systems make digital mediation transparent by placing the physical in direct correspondence with the virtual.

As an analogy, consider a visit to an orbiting space station. We leave the physically familiar Earth, transit through a domain which is not conducive to human inhabitation (empty space), to arrive at an artificial domain (the space station) which is similar enough to Earth to permit inhabitation. Although the space station exists in empty space, it still supports a limited subset of natural behavior. In this analogy the Earth is, of course, physical reality. Empty space is digital reality, the space station is virtual reality. A virtual environment operating system functions to provide an inhabitable zone in the depths of symbolic space. Like the space station, virtual reality is pervaded by essentially alien territory, by binary encodings transacted as voltage potentials through microscopic gates. Early space stations on the digital frontier were spartan, the natural behavior of early infonauts (i.e. programmers) was limited to interpretation of punch cards and hex dumps. Tomorrow's digital space stations will provide human comfort by shielding us completely from the emptiness of syntactic forms.

Another way to view the architecture of a VR system is in terms of meaning, of semantics (Figure 3). A VR system combines two mappings, from physical to digital and from digital to virtual. When a participant points a physical finger, for example, the digital database registers an encoding of pointing. *Physical semantics* is defined by the map between behavior and digital representation. Next, the "pointing" digit stream can be defined to fly the participant's perspective in the virtual environment. *Virtual semantics* is defined by the map between digital representation and perceived effect in the virtual environment. Finally, *natural semantics* is achieved by eliminating our interaction with the intermediate digital syntax. In the example, physical pointing is felt to “cause” virtual flying.

By creating a closed loop between physical behavior and virtual effect, the concepts of digital input and output are essentially eliminated from perception. When natural physical behavior results in natural virtual consequences, without apparent digital mediation, we achieve presence in a new kind of reality, virtual reality. When I knock over my glass, its contents spill. The linkage is direct, natural, and non-symbolic. When I type into my keyboard, I must translate thoughts and feelings through the narrow channel of letters and words. The innovative aspect of VR is to provide, for the first time, natural semantics within a symbolic environment. I can literally spill the image of water from the representation of a glass, and I can do so by the same sweep of my hand.

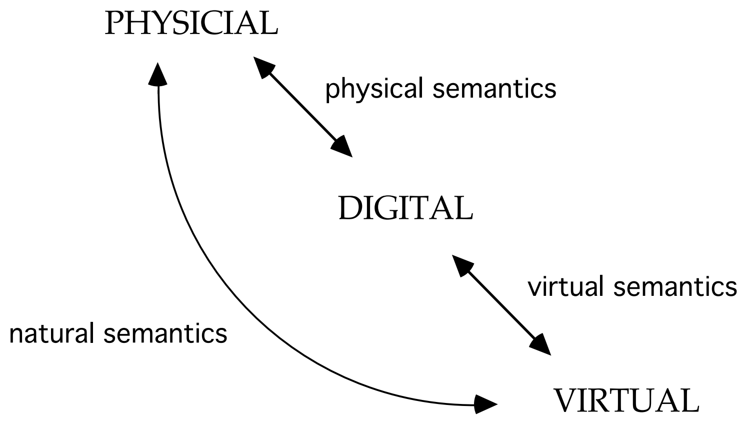

Figure 3: Types of Semantics

Natural semantics affords a surprising transformation. By passing through digital syntax twice, we can finesse the constraints of physical reality.[11] Through presence, we can map physical sensations onto imaginary capacities. We can point to fly. Double-crossing the semantics/syntax barrier allows us to *experience imagination.*

Natural semantics can be very different from physical semantics because the virtual body can be any digital form and can enact any codable functionality. The virtual world is a *physical simulation* only when it is severely constrained. We add collision detection constraints to simulate solidity; we add inertial constraints to simulate Newtonian motion. The virtual world itself, without constraint, is one of potential. Indeed, this is the motivation for visiting VR: although pervaded by both the physical and the digital, the virtual is *larger in possibility* than both.[12]

The idea of a natural semantics that can render representation irrelevant (at least to the interface) deeply impacts the intellectual bases of our culture by questioning the nature of knowledge and representation and by providing a route to unify the humanities and the sciences. The formal theory of VR requires a reconciliation of digital representation with human experience, a reconstruction of the idea of meaning.

4. The Virtual Environment Operating Shell (VEOS)

The Virtual Environment Operating Shell (VEOS) is a software suite operating within a distributed UNIX environment that provides a tightly integrated computing model for data, processes, and communication. VEOS was designed from scratch to provide a comprehensive and unified management facility for generation of, interaction with, and

[11] Crossing *twice* is a mathematical necessity (Spencer-Brown, 1969).
[12] A thing that is larger than its container is the essensce of an imaginary configuration, exactly the properties one might expect from the virtual.

maintenance of virtual environments. It provides an infrastructure for implementation and an extensible environment for prototyping distributed VR applications.

VEOS is platform independent, and has been extensively tested on the DEC 5000, Sun 4, and Silicon Graphics VGX and Indigo platforms. The programmer's interface to VEOS is XLISP 2.1, written for public domain by David Betz. XLISP provides programmable control of all aspects of the operating shell. The underlying C implementation is also completely accessible.

Within VEOS, the *Kernel* manages processes, memory, and communication on a single hardware processor, called a *node*. *FERN* manages abstract task decomposition on each node and distributed computing across networks of nodes. FERN also provides basic functions for entity-based modeling. *SensorLib* provides a library of device drivers. The *Imager* provides graphic output.[13]

Other systems built at HITL enhance the performance and functionality of the VEOS core. *Mercury* is a participant system which optimizes interactive performance. *UM* is the generalized mapper which provides a simple graph-based interface for constructing arbitrary relations between input signals, state information, and output. The *Wand* is a hand-held interactivity device which allows the participant to identify, move, and change the attributes of virtual objects.

We first provide an overview of related work and the design philosophy for the VEOS architecture. Then we present the central components of VEOS: the Kernel, FERN, and entities. The chapter closes with a description of some applications built using VEOS.[14] In contrast to previous sections which discuss interface and architectural theory, this section addresses issues of software design and implementation.

4.1 VR Software Systems

Virtual reality software rests upon a firm foundation built by the computer industry and by academic research over the last several decades. However the actual demands of a VR system (real-time distributed multimedia multiparticipant multisensory environments) provide such unique performance requirements that little research exists to date that is directly relevant to whole VR systems. Instead, the first generation of VR systems have been *assembled* from many relevant component technologies available in published academic research and in newer commercial products.[15]

---

[13] Only the VEOS Kernel and FERN are discussed in this chapter.

[14] For a deeper discussion of the programming and operating system issues associated with VEOS, see Coco (1993).

[15] Aside from one or two pioneering systems built in the sixties (Sutherland, 1965; Furness, 1969), complete VR systems did not become accessible to the general public until June 6, 1989, when both VPL and Autodesk displayed their systems at two concurrent trade shows. Research at NASA Ames (Fisher, 1986) seeded both of these commercial prototypes. At the time, the University of North Carolina was the only academic site of VR research (Brooks, 1986).

The challenge, then, for the design and implementation of VR software is to select and integrate appropriate technologies across several areas of computational research (dynamic databases, real-time operating systems, three-dimensional modeling, real-time graphics, multisensory input and display devices, fly-through simulators, video games, etc.). We describe several related software technologies that have contributed to the decisions made within the VEOS project.

As yet, relatively few turnkey VR systems exist, and of those most are entertainment applications. Notable examples are the multiparticipant interactive games such as LucasArt's Habitat™, W Industries arcade game system, Battletech video arcades, and Network Spector™ for home computers. Virtus Walkthrough™ is one of the first VR design systems.

Architectures for virtual reality systems have been studied recently by several commercial (Blanchard *et al*, 1990; VPL, 1991; Grimsdale, 1991; Appino *et al*, 1992) and university groups (Zeltzer, 1989; Bricken, 1990; Green *et al*, 1991; Pezely *et al*, 1992; Zyda *et al*, 1992; West *et al*, 1992; Kazman, 1993; Grossweiler *et al*, 1993).

Other than HITL at the University of Washington, significant research programs that have developed entire VR systems exist at University of North Carolina at Chapel Hill (Holloway, Fuchs & Robinett, 1992), MIT (Zeltzer *et al*, 1989), University of Illinois at Chicago (Cruz-Neira *et al*, 1992), University of Central Florida (Blau *et al*, 1992), Columbia University (Feiner *et al*, 1992), NASA Ames (Wenzel *et al*, 1990; Fisher *et al*, 1991), and within many large corporations such as Boeing, Lockheed, IBM, Sun, Ford, and AT&T.[16]

More comprehensive overviews have been published for VR research directions (Bishop *et al.* 1992), for software (Zyda *et al*, 1993), for system architectures (Appino *et al*, 1992), for operating systems (Coco, 1993), and for participant systems (Minkoff 1993). HITL has collected an extensive bibliography on virtual interface technology (Emerson 1993).

VR development systems can be grouped into *tool kits* for programmers and *integrated software* for novice to expert computer users. Of course some kits, such as 3D modeling software packages, have aspects of integrated systems. Similarly, some integrated systems require forms of scripting (i.e. programming) at one point or another.

4.1.1 Toolkits

The MR Toolkit was developed by academic researchers at the University of Alberta for building virtual environments and other 3D user interfaces (Green *et al*, 1991). The toolkit takes the form of subroutine libraries which provide common VR services such as tracking, geometry management, process and data distribution, performance analysis, and interaction. The MR Toolkit meets several of the design goals of VEOS, such as modularity, portability and support for distributed computing. MR, however, does not strongly emphasize rapid prototyping; MR programmers use the compiled languages C, C++, and FORTRAN.

[16] Of course the details of most corporate VR efforts are trade secrets.

Researchers at the University of North Carolina at Chapel Hill have created a similar toolkit called VLib. VLib is a suite of libraries that handle tracking, rigid geometry transformations and 3D rendering. Like MR, VLib is a programmer's library of C or C++ routines which address the low level functionality required to support high level interfaces.

Sense8, a small company based in Northern California, produces an extensive C language software library called WorldToolKit™ which can be purchased with 3D rendering and texture acceleration hardware. This library supplies functions for sensor input, world interaction and navigation, editing object attributes, dynamics, and rendering. The single loop simulation model used in WorldToolKit is a standard approach which sequentially reads sensors, updates the world, and generates output graphics. This accumulates latencies linearly, in effect forcing the performance of the virtual body into a codependency with a potentially complex surrounding environment.

Silicon Graphics, an industry leader in high-end 3D graphics hardware, has recently released the *Performer* software library which augments the graphics language GL. Performer was designed specifically for interactive graphics and VR applications on SGI platforms. Autodesk, a leading CAD company which began VR product research in 1988, has recently released the Cyberspace Developer's Kit, a C++ object library which provides complete VR software functionality and links tightly to AutoCAD.

4.1.2 Integrated Systems

When the VEOS project began in 1990, VPL Research, Inc. manufactured RB2™, the first commercially available integrated VR system (Blanchard *et al*, 1990; VPL, 1991). At the time, RB2 supported a composite software suite which coordinated 3D modeling on a Macintosh, real-time stereo image generation on two Silicon Graphics workstations, head and hand tracking using proprietary devices, dynamics and interaction on the Macintosh, and runtime communication over Ethernet. The system processing speed, or throughput, of the Macintosh created a severe bottleneck for this system. VEOS architects had considerable design experience with the VPL system; its pioneering presence in the marketplace helped define many design issues which later systems would improve.

Division, a British company, manufactures VR stations and software. Division's ProVision™ VR station is based on a transputer ring and through the aid of a remote PC controller runs dVS, a director/actors process model (Grimsdale, 1991). Each participant resides on one station; stations are networked for multiparticipant environments. Although the dVS model of process and data distribution is a strong design for transputers, it is not evident that the same approaches apply to workstation LANs, the target for the VEOS project.

Perhaps the most significant distributed immersive simulation system today is the military multiparticipant tank combat simulator, SIMNET (Blau *et al*, 1992). Another advanced military VR simulation system, NPSNET (Zyda *et al*, 1992), has been developed at the Naval Postgraduate School.

4.2 VEOS Design Philosophy

The negotiation between theory and implementation is often delicate. Theory pays little attention to the practical limitations imposed by specific machine architectures and by cost-effective computation. Implementation often must abandon rigor and formality in favor of making it work. In sailing the digital ocean, theory provides the steerage, implementation provides the wind.

The characteristics of the virtual world impose several design considerations and performance requirements on a VR system. The design of VEOS reflects multiple objectives, many practical constraints, and some compromises (Bricken, 1992a).

The dominant design decision for VEOS was to provide broad and flexible capabilities. The mathematical ideals include simplicity (a small number of independent primitives), integration (all primitives are composable), and expressability (primitives and compositions represent all programming domains) (Coco, 1993). Commercial toolkits were not included in VEOS in order to avoid commercial dependencies. VEOS was intended for distribution under royality free license for non-commercial usage.

As a research vehicle, VEOS emphasizes functionality at the expense of performance. Premature optimization is a common source of difficulty in software research. So our efforts are directed first toward demonstrating that a thing can be done at all, then toward demonstrating how well we can do it. Since a research prototype must prepare for the future, VEOS is designed to be as generic as possible; it places very little mechanism in the way of exploring diverse and unexpected design options. It is possible to easily replicate procedural, declarative, functional and object-oriented programming styles within the VEOS framework.

TABLE I: VEOS Practical Design Decisions

Research prototype, 5-10 years ahead of the marketplace
Functional rather than efficient
Rapidly reconfigurable
Synthesis of known software technologies
Incorporates commercially available software when possible

TABLE II: VEOS Functionality

General computing model
Interactive rapid prototyping
Coordination between distributed, heterogeneous resources
Parallel decomposition of worlds (modularity)
Multiple participants

Biological/environmental modeling

Naturally, the VEOS project has passed through several phases over its three years of development. VEOS 2.2 has the desired conceptual structure, but quickly becomes inefficient (relative to a 30 frame per second update rate) when the number of active nodes grows beyond a dozen (Coco & Lion, 1992). Our most recent work, VEOS 3.0, emphasizes performance.

VR is characterized by a rapid generation of applications ideas; it is the potential of VR that people find exciting. However, complex VR systems take too much time to reconfigure. VEOS was designed for rapid prototyping. The VEOS interface is interactive, so that a programmer can enter a new command or world state at the terminal, and on the next frame update the virtual world display will change. VR systems must avoid hardwired configurations, because a participant in the virtual world is free to engage in almost any behavior. For this reason, VEOS is reactive, it permits the world to respond immediately to the participant (and to the programmer).

The broad-bandwidth display and the multisensory interaction of VR systems create severe demands for sensor integration. Visual, audio, tactile, and kinesthetic displays require the VR database to handle multiple data formats and massive data transactions. Position sensors, voice recognition, and high dimensional input devices overload traditional serial input ports. An integrated hardware architecture for VR should incorporate asynchronous communication between dedicated device processors in a distributed computational environment.

When more than one person inhabits a virtual world, the perspective of each participant is different. This can be reflected by different views on the same graphical database. But in the virtual world, multiple participants can have divergent models embodied in divergent databases as well. Each participant can occupy a unique, personalized world, sharing the public database partition and not sharing private database partitions.

With the concept of entities, VEOS extends programming metaphors to include first-class environments, biological models, and systems-oriented programming. A programming metaphor is a way to think about and organize symbolic computation. The biological/environmental metaphor introduced in VEOS originates from the artificial life community (Langton, 1988; Meyer & Wilson, 1991; Varela & Bourgine, 1992); it is a preliminary step toward providing a programming language for modeling autonomous systems within an inclusive environment (Varela, 1979; Maturana & Varela, 1987).

4.3 The VEOS Kernel

The VEOS Kernel is a significant effort to provide transparent low-level database, process, and communications management for arbitrary sensor suites, software resources, and virtual world designs. The Kernel facilitates the VR paradigm shift by taking care of operating system details without restricting the functionality of the virtual world. The Kernel is implemented as three tightly integrated components:

SHELL manages node initialization, linkages, and the LISP interface.
TALK manages internode communications.
NANCY manages the distributed pattern-driven database.

The fundamental unit of organization in the Kernel is the *node*. Each node corresponds to exactly one UNIX process. Nodes map to UNIX processors which ideally map directly to workstation processors.

Nodes running the VEOS Kernel provide a substrate for distributed computing. Collections of nodes form a distributed system which is managed by a fourth component of the VEOS system, FERN. FERN manages sets of uniprocessors (for example, local area networks of workstations) as pools of nodes.

The VEOS programming model is based on entities. An *entity* is a coupled collection of data, functionality and resources, which is programmed using a biological/environmental metaphor. Each entity within the virtual world is modular and self-contained, each entity can function independently and autonomously.

In VEOS, everything is an entity (the environment, the participant, hardware devices, software programs, and all objects within the virtual world). Entities provide database modularity, localization of scoping, and task decomposition. All entities are organizationally identical. Only their structure, their internal detail, differs. This means that a designer needs only one metaphor, the entity, for developing all aspects of the world. Changing the graphical image, or the behavioral rules, or even the attached sensors, is a modular activity. We based the entity concept on distributed object models (Jul *et al*, 1988).

Entities are multiplexed processes on a single node. As well as managing nodes, FERN also manages sets of entities, providing a model of lightweight processing and data partitioning. From the perspective of entity-based programming, the VEOS Kernel is a transparent set of management utilities.

The SHELL is the administrator of the VEOS Kernel. It dispatches initializations, handles interrupts, manages memory, and performs general housekeeping. There is one SHELL program for each node in the distributed computing system. The programmer interface to the SHELL is the LISP programming language, augmented with specialized Kernel functions for database and communications management. LISP permits user configurability of the VEOS environment and all associated functions. LISP can also be seen as a rapid prototyping extension to the native VEOS services.

TALK provides internode communication, relying on common UNIX operating system calls for message passing. It connects UNIX processes which are distributed over networks of workstations into a virtual multi-processor. TALK is the sole mechanism for internode communication. Message passing is the only kind of entity communication supported by TALK, but, depending on context, this mechanism can be configured to behave like shared memory, direct linkage, function evaluation and other communication regimes.

TALK uses two simple asynchronous point-to-point message passing primitives, *send* and *receive*. It uses the LISP functions *throw* and *catch* for process sharing on a single node. Messages are transmitted asynchronously and reliably, whether or not the receiving node is waiting. The sending node can transmit a message and then continue processing. The programmer, however, can elect to block the sending node until a reply, or handshake, is received from the message destination. Similarly, the receiving node can be programmed to accept messages at its own discretion, asynchronously and non-blocking, or it can be programmed to react in a coupled, synchronous mode.

An important aspect of VEOS is consistency of data format and programming metaphor. The structure of messages handled by TALK is the same as the structure of the data handled by the database. The VEOS database uses a Linda-like communication model which partitions communication between processes from the computational threads within a process (Gelertner & Carriero, 1992). Database transactions are expressed in a pattern-directed language.

4.3.1 Pattern-Directed Data Transactions

NANCY, the database transaction manager, provides a content addressable database accessible through pattern-matching. NANCY is a variant of the Linda *parallel database model* to manage the coordination of interprocess communication (Arango *et al*, 1990). In Linda-like languages, communication and processing are independent, relieving the programmer from having to choreograph interaction between multiple processes. Linda implementations can be used in conjunction with many other sequential programming languages as a mechanism for interprocess communication and generic task decomposition (Gelertner, 1990; Cogent, 1990; Torque, 1992). A Linda database supports local, asynchronous parallel processes, a desirable quality for complex, concurrent, interactive systems. NANCY does not support parallel transactions, but the FERN entity manager (Section 4.4) which uses NANCY does.

The Linda approach separates programming into two essentially orthogonal components, *computation* and *coordination*. Computation is a singular activity, consisting of one process executing a sequence of instructions one step at a time. Coordination creates an ensemble of these singular processes by establishing a communication model between them. Programming the virtual world is then conceptualized as defining "a collection of asynchronous activities that communicate" (Gelertner & Carriero, 1992).

NANCY adopts a uniform data structure, as do all Linda-like approaches. In Linda, the data structure is a *tuple*, a finite ordered collection of atomic elements of any type. Tuples are a very simple and general mathematical structure. VEOS extends the concept of a tuple by allowing nested tuples, which we call *grouples*.

A *tuple database* consists of a set of tuples. Since VEOS permits nested tuples, the database itself is a single grouple. The additional level of expressability provided by nested tuples is constrained to have a particular meaning in VEOS. Basically, the nesting structure is mapped onto logical and functional rules, so that the control structure of a program can be expressed simply by the depth of nesting of particular grouples.[17] Nesting implements the concept of containment, so that the contents of a grouple can be interpreted as a set of items, a *grouplespace*.

Grouples provide a consistent and general format for program specification, inter-entity communication and database management. As the VEOS database manager, NANCY performs all grouple manipulations, including creation, destruction, insertion, and copying of grouples. NANCY provides the access functions *put*, *get* and *copy* for interaction with grouplespace. These access functions take patterns as arguments, so that sets of similar grouples can be retrieved with a single call.

Structurally, the database consists of a collection of fragments of information, labeled with unique syntactic identifiers. Collections of related data (such as all of the current properties of Cube-3, for example) can be rapidly assembled by invoking a parallel pattern match on the syntactic label which identifies the sought after relation. In the example, matching all fragments containing the label "Cube-3" creates a grouple with the characteristics of Cube-3. The approach of fragmented data structures permits dynamic, interactive construction of arbitrary grouple collections through real-time pattern-matching. Requesting "all-blue-things" creates a transient complex grouple consisting of all the things in the current environment that are blue. The blue-things grouple is implemented by a dynamic database thread of things with the attribute "color = blue". A blue-things *entity* can be created by passing these attributes to the function for constructing entities.

Performance of the access functions is improved in VEOS by *association matching*. When a process performs a *get* operation, it can block, waiting for a particular kind of grouple to arrive in its perceptual space (the local grouplespace environment). When a matching grouple is *put* into the grouplespace, usually by a different entity, the waiting process gets the grouple and continues. This is implemented through daemons, or react procedures.

Putting and getting data by pattern-matching implements a Match-and-Substitute capability which can be interpreted as the substitution of equals for equals within an algebraic mathematical model. These techniques are borrowed from work in artificial intelligence, and are called *rewrite systems* (Dershowitz 1990).

4.3.2 Languages

Rewrite systems include expert systems, declarative languages, and blackboard systems. Although this grouping ignores differences in implementation and programming semantics, there is an important similarity. These systems are variations on the theme of inference or computation over rule-based or equational representations. Declarative languages such as FP, Prolog, lambda calculus,

[17] Integration of logical control structure with database functionality is not yet implemented.

Mathematica and constraint-based languages all traverse a space of possible outcomes by successively matching variables with values and substituting the constrained value. These languages each display the same trademark attribute: their control structure is *implicit* in the structure of a program's logical dependencies.

The VEOS architects elected to implement a rewrite approach, permitting declarative experimentation with inference and meta-inference control structures. Program control structure is expressed in LISP. As well, this model was also strongly influenced by the language Mathematica (Wolfram 1988).

LISP encourages prototyping partly because it is an interpreted language, making it quite easy to modify a working program without repeated takedowns and laborious recompilation. Using only a small handful of primitives, LISP is fully expressive, and its syntax is relatively trivial to comprehend. But perhaps the most compelling aspect of LISP for the VEOS project is its program-data equivalence. In other words, program fragments can be manipulated as data and data can be interpreted as executable programs. Program-data equivalence provides an excellent substrate for the active message model (vonEicken *et al,* 1992). LISP expressions can be encapsulated and passed as messages to other entities and then evaluated in the context of the receiving entity by the awaiting LISP interpreter.

In terms of availability, LISP has been implemented in many contexts: as a production grade development system (FranzLisp, Inc.), as a proprietary internal data format (AutoLisp from AutoDesk, Inc.), as a native hardware architecture (Symbolics, Inc.), and most relevantly as XLISP, a public domain interpreter (Betz 1992). Upon close inspection, the XLISP implementation is finely-tuned, fully extendible, and extremely portable; it therefore became the clear choice for the VEOS application programmer's interface (API).

4.4 FERN: Distributed Entity Management

The initial two years of the VEOS project focused on database management and Kernel processing services. The third year (1992) saw the development of FERN, the management module for distributed nodes and for lightweight processes on each node.[18] With its features of systems orientation, biological modeling and active environments, FERN extends the VEOS Kernel infrastructure to form the entity-based programming model. We first discuss related work which influenced the development of FERN, then we describe entities in detail.

4.4.1 Distributed Computation

Multi-processor computing is a growing trend (Spector, 1982; Li & Hudak, 1989; Kung *et al,* 1991). VR systems are inherently multi-computer systems, due primarily to the large number of concurrent input devices which do not integrate well in real-time over

[18] As of mid 1993, we have yet to install higher level functions, such as inference engines and learning nets, in entities.

serial ports. The VEOS architects chose to de-emphasize short-term performance issues of distributed computing, trusting that network-based systems would continue to improve. We chose instead to focus on conceptual issues of semantics and protocols.

The operating systems community has devoted great effort toward providing seamless extensions for distributed virtual memory and multiprocessor shared memory. Distributed shared memory implementations are inherently platform specific since they require support from the operating systems kernel and hardware primitives. Although this approach is too low level for the needs of VEOS, many of the same issues resurface at the application level, particularly protocols for coherence.

IVY (Li & Hudak, 1989) was the first successful implementation of distributed virtual memory in the spirit of classical virtual memory. IVY showed that through careful implementation, the same paging mechanisms used in a uniprocessor virtual memory system can be extended across a local area network.

The significance of IVY was twofold. First, it is well known that virtual memory implementations are afforded by the tendency for programs to demonstrate locality of reference. Locality of reference compensates for lost performance due to disk latency. In IVY, locality of reference compensates for network latency as well. In an IVY program, the increase in total physical memory created by adding more nodes sometimes permits a superlinear speedup over sequential execution. Second, IVY demonstrates the performance and semantic implications of various memory coherence schemes. These coherence protocols, which assure that distributed processes do not develop inconsistent memory structures, are particularly applicable to distributed grouplespace implementations.

MUNIN and MIDWAY (Carter *et al*, 1992; Bershad *et al*, 1992) represent deeper explorations into distributed shared memory coherence protocols. Both systems extended their interface languages to support programmer control over the coherence protocols. In MUNIN, programmers always use *release consistency* but can fine-tune the implementation strategy depending on additional knowledge about the program's memory access behavior. In MIDWAY, on the other hand, the programmer could choose from a set of well-defined coherence protocols of varying strength. The protocols ranged from the strongest, *sequential consistency*, which is equivalent to the degenerate distributed case of one uniprocessor, to the weakest, *entry consistency*, which makes the most assumptions about usage patterns in order to achieve efficiency. Each of these protocols, when used strictly, yield correct deterministic behavior.

4.4.2 Lightweight Processes

The VEOS implementation also needed to incorporate some concept of *threads*, cooperating tasks each specified by a sequential program. Threads can be implemented at the user level and often share single address spaces for clearer data sharing semantics and better context-switch performance. Threads can run in parallel on multiple processors or they can be multiplexed preemptively on one processor, thus allowing *n* threads to execute on *m* processors, an essential facility for arbitrary configurations of VEOS entities and available hardware cpus.

This generic process capability is widely used and has been thoroughly studied and optimized. However, thread implementations normally have system dependencies such as the assembly language of the host cpu, and the operating system kernel interface. Inherent platform specificity combined with the observation that generic threads may be too strong a mechanism for VEOS requirements suggest other lightweight process strategies.

The driving performance issue for VR systems is frame update rate. In many application domains, including all forms of signal processing, this problem is represented in general by a discrete operation (or computation) which should occur repeatedly with a certain frequency. Sometimes, multiple operations are required simultaneously but at different frequencies. The problem of scheduling these discrete operations with the proper interleaving and frequency can be solved with a *cyclic executive* algorithm. The cyclic executive model is the *de facto* process model for many small real-time systems.

The cyclic executive control structure was incorporated into VEOS for two reasons. It provided a process model that can be implemented in a single process, making it highly general and portable. It also directly addressed the cyclic and repetitious nature of the majority of VR computation. This cyclic concept in VEOS is called *frames*.

The design of VEOS was strongly influenced by object-oriented programming. In Smalltalk (Goldberg 1980), all data and process is discretized into objects. All parameter passing and transfer of control is achieved through messages and methods. VEOS incorporates the Smalltalk ideals of modular processes and hierarchical code derivation (classes), but does not to enforce the object-oriented metaphor throughout all aspects of the programming environment. More influential was EMERALD (Jul *et al* 1988). The EMERALD system demonstrates that a distributed object system is practical and can achieve good performance through the mechanisms of object mobility and compiler support for tight integration of the runtime model with the programming language. EMERALD implements intelligent system features like location-transparent object communication and automatic object movement for communication or load optimization. As well, EMERALD permits programmer knowledge of object location for fine-tuning applications. EMERALD was especially influential during the later stages of the VEOS project, when it became more apparent how to decompose the computational tasks of VR into entities. In keeping with the ideal of platform independence, however, VEOS steered away from some EMERALD features such as a compiler and tight integration with the network technology.

4.5 Entities

An *entity* is a collection of resources which exhibits behavior within an environment. The entity-based model of programming has a long history, growing from formal modeling of complex systems, object-oriented programming, concurrent autonomous processing and artificial life. Agents, Actors, and Guides all have similarities to entities (Agha, 1988; Oren *et al*, 1990).

Entities act as autonomous systems, providing a natural metaphor for responsive, situational computation. When a single entity resides on a single node, the entity is a stand-alone executable program that is equipped with the VEOS functionalities of data management, process management, and inter-entity communication. In a virtual environment composed of entities, any single entity can cease to function (if, for example, the node supporting that entity crashes) without effecting the rest of the environment.

Entities provide a uniform, singular metaphor and design philosophy for the organization of both physical (hardware) and virtual (software) resources in VEOS. Uniformity means that we can use the same editing, debugging, and interaction tools for modifying each entity.

The biological/environmental metaphor for programming entities provides functions that define perception, action and motivation within a dynamic environment. *Perceive* functions determine which environmental transactions an entity has access to. *React* functions determine how an entity responds to environmental changes. *Persist* functions determine an entity's repetitive or goal-directed behavior.

The organization of each entity is based on a mathematical model of inclusion, permitting entities to serve as both objects and environments. Entities which *contain* other entities serve as their environment; the environmental component of each entity contains the global laws and knowledge of its contents. From a programming context, entities provide an integrated approach to variable scoping and to evaluation contexts. From a modeling point-of-view, entities provide modularity and uniformity within a convenient biological metaphor. But most importantly, from a VR perspective, entities provide first-class environments, *inclusions*, which permit modeling object/environment interactions in a principled manner.

Synchronization of entity processes (particularly for display) is achieved through *frames*. A frame is a cycle of computation for an entity. Updates to the environment are propagated by an entity as discrete actions. Each behavioral output takes a local tick in local time. Since different entities will have different workloads, each usually has a different frame rate. As well, the frame rate of processes internal to an entity is decoupled from the rate of activity an entity exhibits within an environment. Thus, entities can respond to environmental perturbances (reacting) while carrying out more complex internal calculations (persisting).

To the programmer, each entity can be conceptualized to be a separate process. Actual entity processing is transparently multiplexed over available physical processors, or nodes. The entity process is non-preemptive; it is intended to perform only short discrete tasks, yielding quickly and voluntarily to other entities sharing the same node.

Entities can function independently, as worlds in themselves, or they can be combined into complex worlds with other interacting entities. Because entities can access computational resources, an entity can use other software modules available within the containing operating system. An entity could, for instance, initiate and call a statistical analysis package to analyze the content of its memory for recurrent patterns. The capability of entities to link to other systems software make VEOS particularly appealing as a software testing and integration environment.

4.5.1 Systems-Oriented Programming

In object-oriented programming, an object consists of static data and responsive functions, called methods or behaviors. Objects encapsulate functionality and can be organized hierarchically, so that programming and bookkeeping effort is minimized. In contrast, entities are objects which include interface and computational resources, extending the object metaphor to a systems metaphor. The basic prototype entity includes VEOS itself, so that every entity is running VEOS and can be treated as if it were an independent operating environment. VEOS could thus be considered to be an implementation of *systems-oriented programming*.

Entities differ from objects in these ways:

- *Environment:* Each entity functions concurrently as both object and environment. The environmental component of an entity coordinates process sharing, control and communication between entities contained in the environment. The root or global entity is the virtual universe, since it contains all other entities.

- *System:* Each entity can be autonomous, managing its own resources and supporting its own operation without dependence on other entities or systems. Entities can be mutually independent and organizationally closed.

- *Participation:* Entities can serve as virtual bodies. The attributes and behaviors of an inhabited entity can be determined dynamically by the physical activity of the human participant at runtime.

In object-oriented systems, object attributes and inheritance hierarchies commonly must be constructed by the programmer in advance. Efficiency in object-oriented systems usually requires compiling objects. This means that the programmer must know in advance all the objects in the environment and all their potential interactions. In effect, the programmer must be omniscient. Virtual worlds are simply too complex for such monolithic programming. Although object-oriented approaches provide modularity and conceptual organization, in large scale applications they can result in complex property and method variants, generating hundreds of object classes and forming a complex inheritance web. For many applications, a principled inheritance hierarchy is not available, forcing the programmer to limit the conceptualization of the world. In other cases, the computational interaction between objects is context dependent, requiring attribute structures which have not been preprogrammed.

Since entities are interactive, their attributes, attribute values, relationships, inheritances and functionality can all be generated dynamically at runtime. Structures across entities can be identified in real-time based on arbitrary patterns, such as partial matches, unbound attribute values (i.e. abstract objects), ranges of attribute values, similarities, and analogies. Computational parallelism is provided by a fragmented database which provides opportunistic partial evaluation of transactions, regardless of transaction ownership. For coordination, time itself is abstracted out of computation, and is maintained symbolically in data structures.

Although world models composed of collections of objects provide conceptual parallelism (each object is independent of other objects), programming with objects paradoxically enforces sequential modeling, since messages from one object are invariably expected to trigger methods in other objects. Objects are independent only to the extent that they do not interact, but interaction is the primary activity in a virtual world. The essential issue is *determinism*: current object-oriented methodologies expect the programmer to conceptualize interaction in its entirety, between all objects across all possibilities. In contrast, entities support strong parallelism. Entities can enter and leave a virtual environment independently, simply by sending the change to the environment entity which contains them. An autonomous entity is only *perturbed* by interactions; the programmer is responsible for defining subjective behavior locally rather than objective interaction globally. For predictability, entities rely on *equifinality*: although the final result is predictable, the paths to these results are indeterminant.

Dynamic programming of entity behavior can be used by programmers for debugging, by participants for construction and interaction, and by entities for autonomous self-modification. Since the representation of data, function, and message is uniform, entities can pass functional code into the processes of other entities, providing the possibility of genetic and self-adaptive programming styles.

4.5.2 Entity Organization

Each entity has the following components:

- A *unique name*. Entities use unique names to communicate with each other. Naming is location transparent, so that names act as paths to an entity's database partition.

- A *private partition* of the global database. The entity database consists of three subpartitions. The *external* partition contains the entity's environmental observations. The *boundary* partition contains an entity's attributes and its observable form. The *internal* partition contains recorded transactions and internal structure.

- Any number of *processes*. Conceptually, these processes operate in parallel within the context of the entity, as the entity's internal activities. Collectively, they define the entity's autonomous behavior.

- Any number of *interactions*. Entities call upon each others' relational data structures to perform communication and joint tasks. Interactions are expressed as perceptions accompanied potentially by both external reactions and internal model building.

The functional architecture of each entity is illustrated in Figure 4 (Minkoff, 1992). FERN manages the distributed database and the distributed processes within VEOS, providing location transparency and automated coordination between entities. FERN performs three internal functions for each entity:

Communication: FERN manages transactions between an entity and its containing environment (which is another entity) by channeling and filtering accessible global information. TALK, the communication module, facilitates inter-node communication.

Information: Each entity maintains a database of personal attributes, attributes and behaviors of other perceived entities, and attributes of contained entities. The database partitions use the pattern language of NANCY, another basic module, for access.

Behavior: Each entity has two functional loops that process data from the environment and from the entity's own internal states. These processes are LISP programs.

4.5.2.1 Internal Resources

The data used by an entity's processes is stored in five resource areas (Figure 4): hardware (device streams which provide or accept digital information), memory (local storage and workspace) and the three database partitions (external, boundary and internal). These internal resources are both the sources and the sinks for the data created and processed by the entity.

The three database partitions store the entity's information about self and world.[19] Figure 5 illustrates the dual object/environment structure of entities.

The **boundary** partition contains data about the self that is meant to be communicated within the containing environment and thus shared with as many other entities in that environment as are interested. The boundary is an entity's self-presentation to the world. The boundary partition is both readable and writable. An entity reads a boundary (of self or others) to get current state information. An entity writes to its own boundary to change its perceivable state.

The **external** partition contains information about other entities that the self entity perceives. The external is an entity's perception of the world. An entity can set its own perceptual filters to include or exclude information about the world that is transacted in its external. The external is readable only, since it represents externally generated and thus independent information about the world.

---

[19] This tripartite model of data organization is based on spatial rather than textual syntax. The shift is from *labels* which point to objects to *containers* which distinguish spaces. Containers differentiate an outside, an inside, and a boundary between them. Higher dimensional representation is essential for a mathematical treatment of virtual environments (Bricken and Gullichsen, 1989; Bricken, 1991b; Bricken, 1992b). Text, written in one-dimensional lines, is too weak a representational structure to express environmental concepts; words simply lack an inside.

The **internal** partition consists of data in the boundary partitions of contained entities. This partition permits an entity to serve as an environment for other entities. The internal is readable only, since it serves as a filter and a communication channel between contained entities.

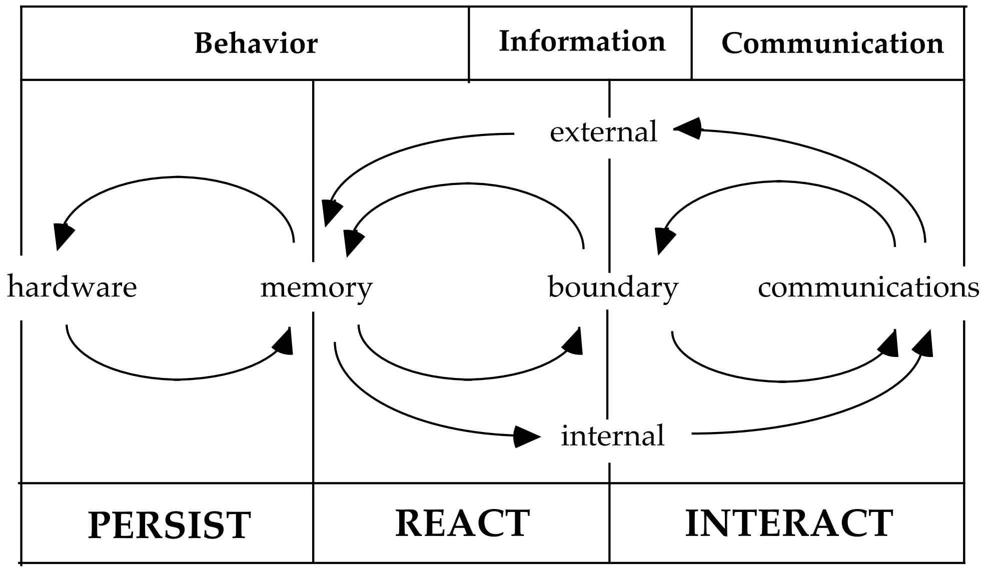

Figure 4: Functionality, Resources and Processes in an Entity

The other two resources contain data about the entity that is never passed to the rest of the world. These connect the entity to the physical world of computational hardware.

The **memory** contains internal data that is not directly communicated to other entities. Memory provides permanent storage of entity experiences and temporary storage of entity computational processes. Internal storage can be managed by NANCY, by LISP, or by the programmer using C.

The **hardware** resource contains data which is generated or provided by external devices. A position tracker, for example, generates both location and orientation information which would be written into this resource. A disc drive may store data such as a behavioral history, written by the entity for later analysis. An inhabited entity would write data to a hardware renderer to create viewable images.

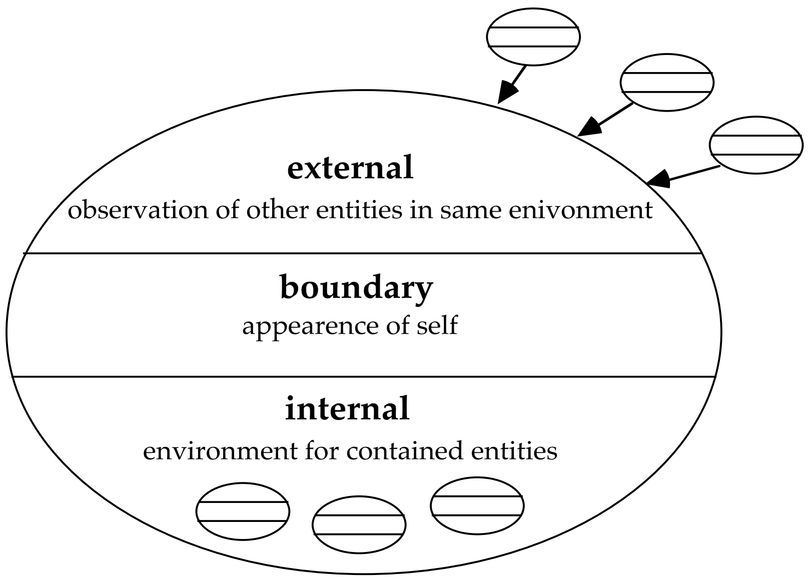
external
observation of other entities in same enivonment
boundary
appearence of self
internal
environment for contained entities

Figure 5: Entities as both Object and Environment

4.5.2.2 Internal Processes

Internal processes are those operations which define an entity's behavior. Behavior can be private (local to the entity) or public (observable by other entities sharing the same environment). There are three types of behavioral processes: each entity has two separate processing regimes (React and Persist), while communications is controlled by a third process (Interact). By decoupling local computation from environmental reactivity, entities can react to stimuli in a time-critical manner while processing complex responses as computational resources permit.

The **Interact** process handles all communication with the other entities and with the environment. The environmental component of each entity keeps track of all contained entities. It accepts updated boundaries from each entity and stores them in the internal data space partition. The environmental process also updates each contained entity's external partition with the current state of the world, in accordance with that entity's perceptual filters. Interaction is usually achieved by sending messages which trigger behavioral methods.[20]

The **React** process addresses pressing environmental inputs, such as collisions with other entities. It reads sensed data and immediately responds by posting actions to the environment. This cycle handles all real-time interactions and all reactions which do not require additional computation or local storage. React processes only occur as new updates to the boundary and external partitions are made.

[20] Technically, in a biological/environmental paradigm, behavior is under autonomous control of the entity and is *not necessarily* triggered by external messages from other entities.

The **Persist** process is independent of any activity external to the entity. The Persist loop is controlled by resources local to the specific entity, and is not responsive in real-time. Persist computations typically require local memory, function evaluation, and inference over local data. Persist functions can copy data from the shared database and perform local computations in order to generate information, but there are no time constraints asserted on returning the results.

The Persist mechanism implements a form of cooperative multitasking. To date, the responsibility of keeping the computational load of Persist processes balanced with available computational resources is left to the programmer. To ensure that multitasking simulates parallelism, the programmer is encouraged to limit the number of active Persist processes, and to construct them so that each is relatively fast, is atomic, and never blocks.

4.5.3 Coherence

FERN provides a simple coherence mechanism for shared grouplespaces that is based on the inter-node message flow control facility. At the end of each frame, FERN takes an inventory of the boundary partitions of each entity on the node, and attempts to propagate the changes to the sibling entities of each of the entities in that environment. Some of these siblings may be maintained by the local node, in which case the propagation is relatively trivial. For local propagation, FERN simply copies the boundary attributes of one entity into the externals of other entities. For remote sibling entities, the grouplespace changes are sent to the nodes on which those entities reside where they are incorporated into the siblings' externals.

Because of mismatched frame rates between nodes, change propagation utilizes a flow-control mechanism. If the logical stream to the remote node is not full, some changes can be sent to that node. If the stream is full, the changes are cached until the stream is not full again. If an entity makes further changes to its boundary while there is still a cached change waiting from that entity, the intermediate value is overwritten. The new change replaces the previous one and continues to wait for the stream to clear. As the remote nodes digest previous change messages, the stream clears and new changes are propagated.

This coherence protocol guarantees the two things. First, if an entity makes a single change to its boundary, the change will reach all subscribing sibling entities. Second, the last change an entity makes to its boundary will reach its siblings. This protocol does not guarantee the intermediate changes because FERN cannot control how many changes an entity makes to its boundary each frame, while it must limit the stack of work that it creates for interacting nodes.

To tie all the FERN features together, Figure 6 provides a graphical overview of the FERN programming model (Coco 1993).

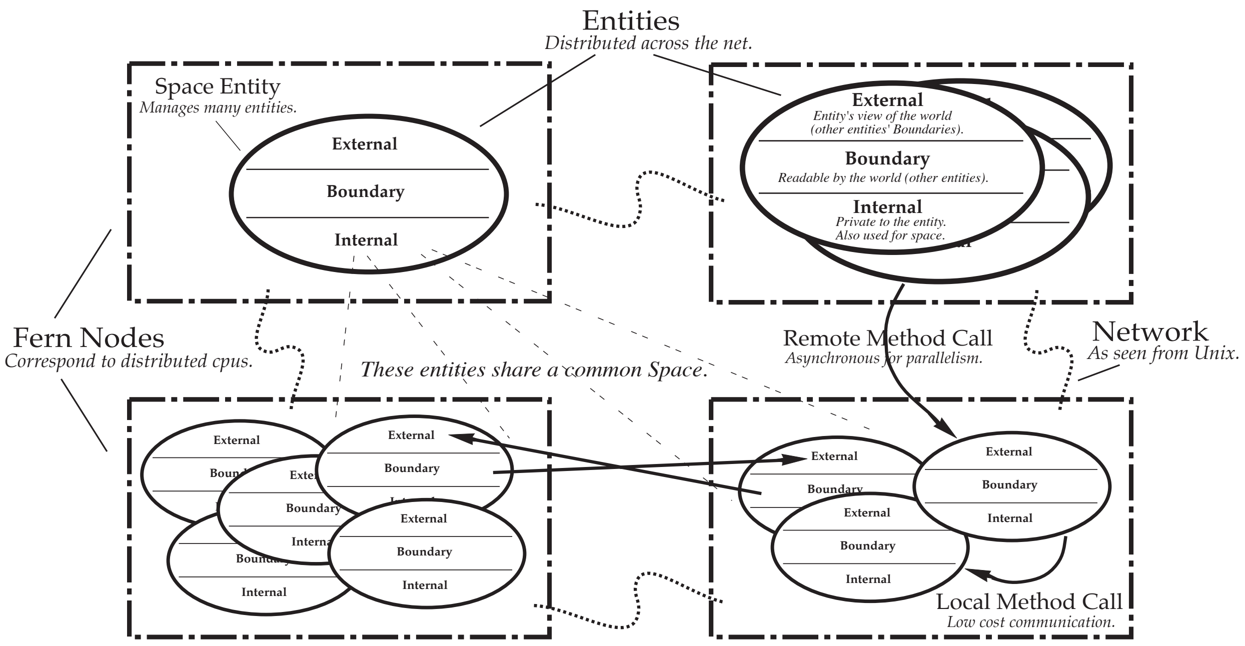
Entities
Distributed across the net.
Space Entity
Manages many entities.
External
Boundary
Internal
Entity's view of the world (other entities' Boundaries).
Readable by the world (other entities).
Private to the entity. Also used for space.
Fern Nodes
Correspond to distributed cpus.
These entities share a common Space.
Remote Method Call
Asynchronous for parallelism.
Network
As seen from Unix.
Local Method Call
Low cost communication.

Figure 6: FERN Topology and Entity Structure

4.5.4 Programming Entities

Complete documentation on the VEOS project, examples and source code are available in the VEOS 3.0 release (VEOS, 1993). Since VEOS supports many programming styles, and since it incorporates techniques from operating systems, database and communication theory, object-oriented programming, theorem proving, artificial intelligence, and interactive interface, it is not possible to present here a complete programming guide. Rather, we will discuss the unique function calls available for entities that support the biological/environmental programming metaphor.[21]

An entity is defined by the LISP function `(fern-entity...)`. This function bundles a collection of other LISP functions which specify all of the entity's initial constructs, forming the entity's capabilities and behavioral disposition. These initializing commands establish the memory allocation, process initialization, and potential activities of an entity.

`(fern-entity...)` actually defines a *class* of entities; each time the function is called, an instance of the entity class is created. Instances are created by `(fern-new-entity <fern-entity-definition>)`. The entity definition itself is a first class citizen that

[21] FERN functions in this section are indicated by `courier font`. Since these are LISP functions, they include the standard LISP parentheses. Angle brackets enclose argument names. An ellipsis within the parentheses indicates arguments which are unspecified. Function names in the text are written in complete words; during actual programming, these function names are abbreviated.

can be loaded unevaluated, bound to a symbol, stored in the grouplespace, or sent as a message. Within an entity definition, the code can include other entity definitions, providing an inheritance mechanism.

The functions normally included within `(fern-entity...)` define the following characteristics:

- **attributes:** `(fern-put-boundary-attribute...)` Properties which are associated with state values are constructed within the entity's boundary resource. Examples of common attributes are listed in Table III.

- **workspace:** `(fern-put-local...)` Local memory and private workspace resources are reserved within a local partition of the database.

- **behavior:** `(fern-define-method...)` Methods which define an entity's response to the messages it receives are defined as functions which are evaluated within the local context.

- **processes:** `(fern-persist...)` Persistent processes within an entity are defined and initialized. An entity can engage in many processes which timeshare an entity's computational process resources.

- **perceptions:** `(fern-perceive...)` When specific changes occur in an entity's environment, the entity is immediately notified, modeling a perceptual capability. An entity can only access external data which it can perceive.

- **peripherals:** `(sensor-init...)` Connections to any physical sensors or input devices used by the entity are established and initialized.

- **functionality:** `(define <function-name>...)` Any particular functions required to achieve the above characteristics are defined within the entity's local context.

As well as defining entities, FERN includes functions for initializing the computational environment `(fern-init...)`, changing the platforms which form the processor pool `(fern-merge-pool...)`, running the FERN process on each node `(fern-run...)`, and providing debugging, timing, and connectivity information. FERN also provides management facilities for all functions `(fern-close)(fern-detach-pool...)(fern-dispose-entity...)(fern-undefine-method...)(fern-unperceive)`.

TABLE III: Common Attributes of Entities

Name	
	concise (human readable)
	verbose (human readable)
	system-wide

Spatial
- location in three dimensions
- orientation in three dimensions
- scale

Visual
- picture-description
- color
- visibility
- opacity
- wireframe
- texture-description
- texture-map
- texture-scale

Aural
- sound-description
- loudness
- audibility
- sound-source
- doppler
- rolloff
- midi-description
- midi-note (pitch, velocity, sustain)

Dynamic
- mass
- velocity
- acceleration

Object/environment relationships are created by `(fern-enter <space-id>)`. The contained entity sends this registration message to the containing entity. Entities within a space can access only the perceivable aspects of other entities *in the same space.* That entities can both act as spaces and enter other spaces suggests a hierarchical nature to spaces. However, any hierarchy significance must be implemented by the application. Spaces as such are primarily a dataspace partitioning mechanism.

Entities can select and filter what they perceive in a space with `(fern-perceive <attribute-of-interest>)`. These filters constrain and optimize search over the shared dataspace. For example, should an entity wish to perceive changes in the color of other entities in its environment, the following code would be included in the entity's definition: `(fern-perceive "color")`. This code will automatically optimize the shared dataspace for access to color changes by the interested entity, posting those changes directly in the external partition of the entity.

4.5.4.1 Processes

All FERN application tasks are implemented as one of three types of entity processes:

- react `(fern-perceive <attribute> :react <react-function>)`

- persist `(fern-persist <persist-function>)`
- interact `(fern-define-method <message-name>...)`
  `(fern-send <entity> <message-name>...)`.

React processes are triggered when entities make changes to the shared grouplespace. Since reactions occur only as a function of perception, they are included with the perceive function. For example, an entity may want to take a specific action whenever another entity changes color:

```
(fern-perceive "color" :react (take-specific-action))
```

Persist processes can be used to perform discrete computations during a frame of time (for example, applying recurrent transformations to some object or viewpoint each frame). Persist processes can also be used in polling for data from devices and other sources external to VEOS. The following simple example reads data from a dataglove and sends it to the relocate-hand method of a renderer entity, updating that data once every rendering frame:

```
(fern-persist '(poll-hand))

(define poll-hand ()
  (let ((data (read-position-of-hand)))
    (if data (fern-send renderer "relocate-hand" data) )))
```

When persist processes involve polling, they often call application specific primitives written in C. The `(read-position-of-hand)` primitive would most likely be written in C since it accesses devices and requires C level constructs for efficient data management.

During a single frame, FERN's cyclic executive evaluates every persist process installed on that node exactly once. For smoother node performance, FERN interleaves the evaluation of persist processes with evaluation of queued asynchronous messages. When a persist process executes, it runs to completion like a procedure call on the node's only program stack.[22]

Interact processes are implemented by object-oriented messages and methods.[23] Like Smalltalk, FERN methods are used to pass data and program control between entities. An entity can invoke the methods of other entities by sending messages. The destination entity can be local or remote to the sending entity and is specified by the destination entity's network-wide unique id. A method is simply a block of code that an entity provides with a well-defined interface. The following method belongs to a renderer entity, and calls the hand update function.

[22] In comparison, preemptive threads each have their own stack where they can leave state information between context switches.
[23] It is appropriate to model interaction *between* entities using the objective, external perspective of object-oriented programming.

```
(fern-define-method "relocate-hand" new-position
  (lambda (new-position) (render-hand new-position)))[24]
```

4.5.4.2 Messages

Messages can be sent between entities by three different techniques, asynchronous `(fern-send...)`, synchronous `(fern-sequential-send...)` and flow controlled `(fern-stream-send...)`.

Asynchronous messages are most common and ensure the smoothest overall performance. An entity gets program control back immediately upon sending the message regardless of when the message is handled by the receiving entity. The following message might be sent to a renderer entity by the hand entity to update its display position:

```
(fern-send renderer "relocate-hand" current-position)
```

When the receiving entity is remote, a message is passed to the Kernel inter-node communication module and sent to the node where the receiving entity resides. When the remote node receives the message, it posts it on the asynchronous message queue. When the receiving entity is local, a message is posted to the local message queue and handled by FERN in the same way as remote messages.

Although asynchronous message delivery is guaranteed, there is no guarantee when the receiving entity will actually execute the associated method code. As such, an asynchronous message is used when timing is not critical for correctness. In cases where timing is critical, there are common idioms for using asynchronous semantics to achieve determininstic transmission.

Synchronous messages assure control of timing by passing process control from the sending to the receiving entity, in effect simulating serial processing in a distributed environment. When an entity sends a synchronous message, it blocks, restarting processing again only when the receiving entity completes its processing of the associated method and returns an exit value to the sending entity.

Although the VEOS communication model is inherently asynchronous, there are two occasions when synchronous messages may be desirable: when the sending entity needs a return value from the receiving entity, or when the sending entity needs to know exactly when the receiving entity completes processing of the associated method. Although both of these occasions can be handled by asynchronous means, the asynchronous approach may be more complicated to implement and may not achieve the lowest latency. The most important factor in choosing whether to use synchronous or asynchronous messages is whether the destination entity is local or remote. In the remote case, synchronous messages will sacrifice local processor utilization because the entire node blocks waiting for the reply, but in doing so the sending entity is assured

[24] `Lambda` is LISP for "this code fragment is a function".

the soonest possible notification of completion. In the local case, a synchronous method call reduces to a function call and achieves the lowest overall overhead.

A third message passing semantic is needed to implement a communications pacing mechanism between entities. Because interacting entities may reside on different nodes with different frame rates, they may each have different response times in transacting methods and messages.

Stream messages implement a flow-control mechanism between entities. In cases where one entity may generate a stream of messages faster than a receiving entity can process them, stream messages provide a pacing mechanism, sending messages only if the stream between the two nodes is not full. Streams ensure that sending entities only send messages as fast as receiving entities can process them. The user can set the size of the stream, indicating how many buffered messages to allow. A larger stream gives better throughput because of the pipelining effect, but also results in bursty performance due to message convoying.

Streams are usually used for transmission of *delta* information, information indicating changes in a particular state value. Polling a position tracker, for example, provides a stream of changes in position. Streams are useful when data items can be dropped without loss of correctness.

4.5.4.3 Examples of FERN Usage

Entering a world: To enter a new environment, an entity notifies the entity which manages that environment (as an internal partition). Subsequent updates to other entities within that environment will automatically include information about the incoming entity.

Follow: By associating an entity's position with the location of another entity (for example, Position-of-A = Position-of-B + offset), an entity will follow another entity. Following is dependent on another entity's behavior, but is completely within the control of the following entity.

Move with joystick: The joystick posts its current values to its boundary. A virtual body using the joystick to move would react to the joystick's boundary, creating an information linkage between the two entities.

Inhabitation: The inhabiting entity uses the inhabited entity's relevant boundary information as its own, thus creating the same view and movements as the inhabited entity.

Portals: An entity sensitive to portals can move through the portal to another location or environment. Upon entering the portal, the entity changes its boundary attributes to the position, orientation and other spatial values defined by the portal.

4.5.4.4 A Simple Programming Example

Finally, we present a complete FERN program to illustrate basic biological/environmental concepts within a functional programming style. When called within a LISP environment, this program creates a *space* entity, which in turn creates two other entities, *tic* and *toc*. All three entities in this simple example exist on one node; the default node is the platform which FERN is initialized on.[25]

```
(define simple-communications-test ()
   (run "space") )
```

In the file "space":

```
(entity-specification
   (new-entity "tic")
   (new-entity "toc"))
```

In the file "tic":

```
(entity-specification
   (enter (copy.source))      ;space is the source
   (put.attribute '(tics 0))
   (perceive 'tocs
      :react '(lambda (ent value) (print "Tic sees: " value)))
   (persist '(let ((new-value (1+ (copy.attribute 'tics))))
                (print "Tic says: " new-value)
                (put.attribute `(tics ,new-value)) )) )
```

In the file "toc":

```
(entity-specification
   (enter (copy.source))
   (put.attribute '(tocs 1000))
   (perceive 'tics
      :react '(lambda (ent value) (print "Toc sees: " value)))
   (persist '(let ((new-value (1- (copy.attribute 'tocs))))
                (print "Toc says: " new-value)
                (put.attribute `(tocs ,new-value)) )) )
```

Tic and *toc* each enter the space which created them, and each establish a single attribute which stores a numerical value. Jointly subscribing to *space* permits each entity to perceive the attributes of the other. *Tic* persists in incrementing its attribute value, prints that current value to the console, and stores the new value. *Toc* persists in

[25] This example is written in LISP and suffers from necessary LISP syntax. The names of actual functions have been changed in the example, to simplify reading of intent. A non-programmer's interface for configuring entities could be based on filling forms, on menu selections, or even on direct interaction within the virtual environment. We have not yet implemented a non-programmer's interface.

decrementing its attribute value. The perceive function of each looks at the current value of the other entity's attribute and prints what it sees to the console of the default platform. `Simple-communications-test` generates asynchronous varieties[26] of the following console output:

```
Tic says 1
Toc says 999
Tic says 2
Toc says 998
Toc sees 2
Tic sees 998
Toc says 997
Tic says 3
Toc says 996
Toc sees ...
```

5. Applications

VEOS was developed iteratively over three years, in the context of prototype development of demonstrations, theses and experiments at HITL. It was constantly under refinement, extension and performance improvement. It has also satisfied the diverse needs of all application projects, fulfilling the primary objective of its creation. Although not strictly academic research, the VEOS project does provide a stable prototype architecture and implementation that works well for many VR applications. We briefly describe several.

Tours: The easiest type of application to build with VEOS is the virtual tour. These applications provide little interactivity, but allow the participant to navigate through an interesting environment. All that need be built is the interesting terrain or environment. These virtual environments often feature autonomous virtual objects that do not significantly interact with the participant.

Examples of tours built in VEOS are:

- an aircraft walkthrough built in conjunction with Boeing corporation,
- the TopoSeattle application where the participant could spatially navigate and teleport to familiar sites in the topographically accurate replica of the Seattle area, and
- the Metro application where the participant could ride the ever-chugging train around a terrain of rolling hills and tunnels.

[26] The sequence of changing tics and tocs remains constant for each entity, but what each entity sees depends upon communication delays in database transactions. What each entitiy tells you that it sees depends upon how the underlying operating system differentially manages processing resources for print statements within `persist` and `perceive` operations.

Physical Simulation: Because physical simulations require very precise control of the computation, they have been a challenging application domain. Coco and Lion (1992) implemented a billiard ball simulation to measure VEOS's performance, in particular to measure the tradeoffs between parallelism and message passing overhead. Most of the entity code for this application was written in LISP, except for ball collision detection and resolution, which was written in C to reduce the overhead of the calculations.

The simulation coupled eighteen entities. Three entities provided an interface to screen based rendering facilities, access to a spaceball six-degree-of-freedom input device, and a command console. The rendering and spaceball entities worked together much like a virtual body. The spaceball entity acted as a virtual hand, using a persist procedure to sample the physical spaceball device and make changes to the 3D model. The imager entity acted as a virtual eye, updating the screen-based view after each model change made by the spaceball entity. The console entity managed the keyboard and windowing system.

Asynchronous to the participant interaction, fifteen separate ball entities continually recomputed their positions. Within each frame, each ball, upon receiving updates from other balls, checked for collisions. When each ball had received an update from every other ball at the end of each frame, it would compute movement updates for the next frame. The ball entities sent their new positions via messages to the imager entity which incorporated the changes into the next display update. The ball entities used asynchronous methods to maximize parallelism within each frame. Balls did not wait for all messages to begin acting upon them. They determined their new position iteratively, driven by incoming messages. Once a ball had processed all messages for one frame, it sent out its updated position to the other balls thus beginning a new frame.

Multiparticipant Interactivity: In the early stages of VEOS development, Coco and Lion designed an application to demonstrate the capabilities of multiparticipant interaction and independent views of the virtual environment. Block World allowed four participants to independently navigate and manipulate moveable objects in a shared virtual space. Each participant viewed a monitor based display, concurrently appearing as different colored blocks on each other's monitor. Block World allowed for interactions such as 'tug-of-war' when two participants attempted to move the same object at the same time. This application provided experience for the conceptual development of FERN.

One recent large scale application provided multiparticipant interaction by playing catch with a virtual ball while supporting inter-participant spatial voice communication. The Catch application incorporated almost every interaction technique currently supported at HITL including head tracking, spatial sound, 3D binocular display, wand navigation, object manipulation, and scripted movement paths.

Of particular note in the Catch application was the emphasis on independent participant perceptions. Participants customized their personal view of a shared virtual environment in terms of color, shape, scale, and texture. Although the game of catch was experienced in a shared space, the structures in that space were substantively different for each participant. Before beginning the game, each player selected the form of their virtual body and the appearance of the surrounding mountains and flora. One participant may see a forest of evergreens, for example, while concurrently the other saw a field of flowers. Participants experienced the Catch environment two at a time, and could compare their experiences at runtime through spatialized voice communication. The spatial filtering of the voice interaction provided each participant with additional cues about the location of the other participant in the divergent world.

Manufacturing: For her graduate thesis, Karen Jones worked with HITL engineer Marc Cygnus to develop a factory simulation application (Jones, 1992). The program incorporated an external interface to the AutoMod simulation package. The resulting virtual environment simulated the production facility of the Derby Cycle bicycle company in Kent, Washington, and provided interactive control over production resources allocation. The Derby Cycle application was implemented using a FERN entity for each dynamic object and one executive entity that ensured synchronized simulation time steps. The application also incorporated the Body module for navigation through the simulation.

Spatial Perception: Coming from an architectural background, Daniel Henry wrote a thesis on comparative human perception in virtual and actual spaces (Henry, 1992). He constructed a virtual model of the Henry Art Gallery on the University of Washington campus. The study involved comparison of subjective perception of size, form, and distance in both the real and virtual gallery. This application used the Body module for navigation through the virtual environment. The results indicated that the perceived size of the virtual space was smaller than the perceived size of the actual space.

Scientific Visualization: Many applications have been built in VEOS for visualizing large or complex data sets. Our first data visualization application was of satellite collected data of the Mars planet surface. This application allowed the participant to navigate on or above the surface of Mars and change the depth ratio to emphasize the contour of the terrain. Another application designed by Marc Cygnus revealed changes in semiconductor junctions over varying voltages. To accomplish this, the application displayed the patterns generated from reflecting varying electromagnetic wave frequencies off the semiconductor.

Education: Chris Byrne led a program at HITL to give local youth the chance to build and experience virtual worlds. The program emphasized the cooperative design process of building virtual environments. These VEOS worlds employed the standard navigation techniques of the wand and many provided interesting interactive features. The implementations include an AIDS awareness game, a Chemistry World and a world which modeled events within an atomic nucleus.

Creative Design: Using the Universal Motivator graph configuration system, Colin Bricken designed several applications for purely creative ends. These environments are characterized by many dynamic virtual objects which display complex behavior based on autonomous behavior and reactivity to participant movements.

6. Conclusion

Operating architectures and systems for real-time virtual environments have been explored in commercial and academic groups over the last five years. One such exploration is the VEOS project, which is beginning its fourth year.

We have learned that the goals of the VEOS project are ambitious; it is difficult for one cohesive system to satisfy demands of conceptual elegance, usability and performance even for limited domains. VEOS attempts to address these opposing top level demands through its hybrid design. In this respect, perhaps the strongest attribute of VEOS is that it promotes modular programming. Modularity has allowed for incremental performance revisions as well as incremental and cooperative tool design. Most importantly, the emphasis on modularity facilitates the process of rapid prototyping that was sought by the initial design.

VR design is inherently a multidisciplinary process and requires expertise in many different areas. Successful VR applications are manifested by the cooperative effort of system programmers implementing and abstracting performance bottlenecks, designers creating involving objects and terrains, dynamics experts implementing realistic behaviors, authors composing story lines, psychological researchers focusing on perceptual understanding, systems architects building automated and reliable infrastructures, and visionaries encouraging the whole process.

Now that the infrastructure of virtual worlds (behavior transducers and coordination software) is better understood, the more significant questions of the design and construction of psychologically appropriate virtual/synthetic experiences will see more attention. Biological/environmental programming of entities can provide one route to aid in the humanization of the computer interface.

7. References

Agha, G. (1988) *Actors: a model of concurrent computation in distributed systems*. MIT Press.

Appino, P.A., Lewis, J.B., Koved, L., Ling, D.T., Rabenhorst, D. & Codella, C. (1992) An architecture for virtual worlds. *Presence*, 1(1), 1-17.

Arango, M., Berndt, D., Carriero, N., Gelertner, D. & Gilmore, D. (1990) Adventures with network linda, *Supercomputing Review*, October 1990, 42-46.

Bershad, B., Zekauskas, M. J. & Swadon, W. A. (1992) The midway distributed shared memory system, School of Computer Science, Carnegie Mellon University.

Betz, D. & Almy, T. (1992) XLISP 2.1 User's Manual.

Bishop, G., Bricken, W., Brooks, F., *et al*. (1992) Research directions in virtual environments: report of an NSF invitational workshop. *Computer Graphics* 26(3), 153-177.

Blanchard, C., Burgess, S., Harvill, Y., Lanier, J., Lasko, A., Oberman, M. & Teitel, M. (1990) Reality built for two: a virtual reality tool. *Proceedings 1990 Symposium on Interactive Graphics*, Snowbird Utah, 35-36.

Blau, B., Hughes, C.E., Moshell, J.M. & Lisle, C. (1992) Networked virtual environments. *Computer Graphics 1992 Symposium on Interactive 3D Graphics*, 157.

Bricken, M. (1991) Virtual worlds: no interface to design. in Benedikt, M. (ed) *Cyberspace first steps*. MIT Press, 363-382.

Bricken, W. & Gullichsen, E. (1989) An introduction to boundary logic with the LOSP deductive engine, *Future Computing Systems* 2(4).

Bricken, W. (1990) Software architecture for virtual reality. *Human Interface Technology Lab Technical Report P-90-4*, University of Washington.

Bricken, W. (1991a) VEOS: preliminary functional architecture, *ACM Siggraph'91 Course Notes, Virtual Interface Technology*, 46-53. Also *Human Interface Technology Lab Technical Report M-90-2*, University of Washington.

Bricken, W. (1991b) A formal foundation for cyberspace. *Proceedings of Virtual Reality '91, The Second Annual Conference on Virtual Reality, Artificial Reality, and Cyberspace*, San Francisco, Meckler.

Bricken, W. (1992a) VEOS design goals. *Human Interface Technology Lab Technical Report M-92-1*, University of Washington.

Bricken, W. (1992b) Spatial representation of elementary algebra, *1992 IEEE Workshop on Visual Languages*, Seattle, IEEE Computer Society Press, 56-62.

Brooks, F. (1986) Walkthrough -- a dynamic graphics system for simulation of virtual buildings. *Proceedings of the 1986 Workshop on Interactive 3D Graphics.* ACM. 271-281.

Bryson, S. & Gerald-Yamasaki, M. (1992) The distributed virtual wind tunnel. *Proceedings of Supercomputing '92*, Minneapolis, Minn.

Carter, J. B., Bennet, J. K. & Zwaenepoel, W. (1992) Implementation and performance of munin, Computer Systems Laboratory, Rice University.

Coco, G. (1993) *The virtual environment operating system: derivation, function and form.* Masters Thesis, School of Engineering, University of Washington.

Coco, G. & Lion, D. (1992) Experiences with asychronous communication models in VEOS, a distributed programming facility for uniprocessor LANs. *Human Interface Technology Lab Technical Report R-93-2*, University of Washington.

Cogent Research, Inc. (1990) Kernel linda specification: version 4.0. Technical Note, Beaverton, Oregon.

Cruz-Neira, C., Sandin, D.J., DeFanti, T., Kenyon, R. & Hart, J. (1992) The cave: audio visual experience automatic virtual environment, *CACM* 35(6), 65-72.

Dershowitz, N. & Jouannaud, J. P. (1990) Chapter 6: rewite systems, *Handbook of Theoretical Computer Science,* Elsevier Science Publishers, 245-320.

Ellis, S.R. (1991) The nature and origin of virtual environments: a bibliographical essay. *Computer Systems in Engineering,* 2(4), 321-347.

Emerson, T. (1993) Selected bibliography on virtual interface technology. *Human Interface Technology Lab Technical Report B-93-2*, University of Washington.

Feiner, S., MacIntyre, B. & Seligmann, D. (1992) Annotating the real world with knowledge-based graphics on a "see-through" head-mounted display. *Proceedings of Graphics Interface '92*, Vancouver Canada, 78-85.

Fisher, S., McGreevy, M., Humphries, J. & Robinett, W. (1986) Virtual environment display system, *ACM Workshop on Interactive 3D Graphics*, Chapel Hill, NC.

Fisher, S., Jacoby, R., Bryson, S., Stone, P., McDowell, I., Bolas, M., Dasaro, D., Wenzel, E. & Coler, C. (1991) The ames virtual environment workstation: implementation issues and requirements. *Human-Machine Interfaces for Teleoperators and Virtual Environments*. NASA 20-24.

Furness, T. (1969) Helmet-mounted displays and their aerospace applications. *National Aerospace Electronics Conference*. Piscataway, NJ: IEEE.

Gelertner, D. & Carriero, N. (1992) Coordination languages and their significance. *Communications of the ACM*, 35(2), 97-107.

Gelertner, D., & Philbin, J. (1990) Spending Your Free Time, *Byte*, May 1990.

Goldberg, A. (1984) *Smalltalk-80*, Xerox Corporation; Addison Wesley.

Green, M., Shaw, C., Liang, J. & Sun, Y. (1991) MR: a toolkit for virtual reality applications. Department of Computer Science, University of Alberta, Edmonton, Canada

Grimsdale, C. (1991) dVS: distributed virtual environment system. Product documentation, Division Ltd. Bristol, UK.

Grossweiler, R., Long, C., Koga, S. & Pausch, R. (1993) DIVER: a distributed virtual environment research platform, Computer Science Department, University of Virginia.

Henry, D. (1992) *Spatial perception in virtual environments: evaluating an architectural application.* Masters Thesis, School of Engineering, University of Washington.

Holloway, R., Fuchs, H. & Robinett, W. (1992) Virtual-worlds research at the University of north carolina at chapel hill, Course #9 Notes: Implementation of Immersive Virtual Environments, *SIGGRAPH'92* Chicago Ill.

Jones, K. (1992) *Manufacturing simulation using virtual reality*. Masters Thesis, School of Engineering, University of Washington.

Jul, E., Levy, H., Hutchinson, N. & Black, A. (1988) Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1), 109-133.

Kazman, R. (1993, to appear) HIDRA: an architecture for highly dynamic physically based multi-agent simulations. *International Journal of Computer Simulation*.

Kung, H. T., Sansom, R., Schlick, S., Steenkiste, P., Arnould, M., Bitz, F.J., Christianson, F., Cooper, E.C., Menzilcioglu, O., Ombres, D. & Zill, B. (1991) Network-based multicomputers: an emerging parallel architecture, *ACM Computer Science*, 664-673.

Langton, C. (1988) *Artificial life: proceedings of an interdisciplinary workshop on the synthesis and simulation of living systems.* Addison-Wesley

Li, K. & Hudak, P. (1989) Memory coherence in shared virtual memory systems, *ACM Transactions on Computer Systems*, 7(4), 321-359.

Maturana, H. & Varela, F. (1987) *The tree of knowledge*. New Science Library.

Meyer, J. & Wilson, S. (1991) *From animals to animats: proceedings of the first international conference on simulation of adaptive behavior.* MIT Press.

Minkoff, M. (1992) The FERN model: an explanation with examples. *Human Interface Technology Lab Technical Report R-92-3*, University of Washington.

Minkoff, M. (1993) *The participant system: providing the interface in virtual reality.* Masters Thesis, School of Engineering, University of Washington.

Naimark, M. (1991) Elements of realspace imaging: a proposed taxonomy. *Proceedings of the SPIE 1457, Stereoscopic Displays and Applications II*. SPIE 169-179

Oren, T., Salomon, G., Kreitman, K. & Don, A. (1990) Guides: characterising the interface. in Laurel, B. (ed) *The art of human-computer interface design.* Addison-Wesley.

Pezely, D.J., Almquist, M.D. & Bricken, W. (1992) Design and implementation of the meta operating system and entity shell. *Human Interface Technology Lab Technical Report R-91-5*, University of Washington.

Robinett, W. (1992) Synthetic experience: a proposed taxonomy. *Presence* 1(2), 229-247.

Robinett, W. & Holloway, R. (1992) Implementation of flying, scaling and grabbing in virtual worlds. *Computer Graphics 1992 Symposium on Interactive 3D graphics*. 189.

Spector, A. Z. (1982) Performing remote operations efficiently on a local computer network, *Communications of the ACM*, 25(4), 246-260.

Spencer-Brown, G. (1969) *Laws of Form.* Bantam.

Sutherland, I. (1965) The ultimate display. *Proceedings of the IFIP Congress*, 502-508.

Torque Systems, Inc. (1992) Tuplex 2.0 software specification. Palo Alto, Calif.

Varela, F. (1979) *Principles of Biological Autonomy*. Elsevier North Holland.

Varela, F. & Bourgine, P. (1992) *Toward a practice of autonomous systems: proceedings of the first european conference on artificial life*. MIT Press.

VEOS 3.0 Release (1993) Human Interface Technology Lab, University of Washington FJ-15, Seattle WA 98195.

von Eicken, T., Culler, D.E., Goldstein, S. C. & Schauser, K. E. (1992) Active messages: a mechanism for integrated communication and computation, *ACM*, 256-266.

VPL (1991) Virtual reality data-flow language and runtime system, body electric manual 3.0. VPL Research, Redwood City, CA.

Wenzel, E., Stone, P., Fisher, S. & Foster, S. (1990) A system for three-dimensional acoustic 'visualization' in a virtual environment workstation. *Proceedings of the First IEEE Conference on Visualization, Visualization '90*. IEEE 329-337.

West, A.J., Howard, T.L.J., Hubbold, R.J., Murta, A.D., Snowdon, D.N. & Butler, D.A. AVIARY - a generic virtual reality interface for real applications. Department of Computer Science, University of Manchester, UK.

Wolfram, S. (1988) *Mathematica: a system for doing mathematics by computer*. Addison-Wesley.

Zeltzer, D., Pieper, S. & Sturman, D. (1989) An integrated graphical simulation platform. *Graphics Interface '89*, Canadian Information Processing Society, 266-274.

Zeltzer, D. (1992) Auonomy, interaction, and presence. *Presence*, 1(1), 127-132.

Zyda, M.J., Akeley, K., Badler, N., Bricken, W., Bryson, S., vanDam, A., Thomas, J. Winget, J., Witkin, A., Wong, E. & Zeltzer, D. (1993) Report on the state-of-the-art in computer technology for the generation of virtual environments, Computer Generation Technology Group, National Academy of Sciences, National Research Council Committee on Virtual Reality Research and Development.

Zyda, M.J., Pratt, D.R., Monahan, J.G. & Wilson, K.P. (1992) NPSNET: constructing a 3D virtual world. *Computer Graphics*, 3, 147.