The VEOS Project[1]

William Bricken and Geoffrey Coco

Abstract

The Virtual Environment Operating Shell (VEOS) was developed at University of Washington's Human Interface Technology Laboratory as software infrastructure for the lab's research in virtual environments. 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. We 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 software.

---

[1] The VEOS project was a group effort to create a satisfying software infrastructure for research into VR. 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 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

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. Virtual environment 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 attempt to redefine the relationship between experience and representation, in effect rendering the syntax-semantics barrier transparent. The idea is to hide reading, writing, and arithmetic from the computer interface, replacing them by direct, non-symbolic environmental experience.

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

This article presents some of the operating systems techniques and software tools which guided the early development of virtual environment systems at the University of Washington Human Interface Technology Lab. We first describe existing virtual environment systems. Next, the goals of the VEOS project are presented and the two central components of VEOS, the Kernel and FERN, are described. The article 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 article 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. 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 maintenance of virtual environments. It provides an infrastructure for implementation and an extensible environment for prototyping distributed virtual environment 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.[2]

Other systems built at HITL enhance the performance and functionality of the VEOS core. *Mercury* is a participant system which optimizes interactive performance. *UIM* 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.

2.1 Virtual Environment Software Systems

Virtual environment (VE) 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 VE 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 virtual environment systems. Instead, the first generation of virtual environment systems have been *assembled* from many relevant component technologies available in published academic research and in newer commercial products.[3]

Taxonomies of the component technologies and functionalities of VE 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.

The challenge, then, for the design and implementation of VE 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

---

[2] Only the VEOS Kernel and FERN are discussed in this chapter. For a deeper discussion of the programming, operating system and performance issues associated with VEOS, see Coco (1993).

[3] 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).

games, etc.). We describe several related software technologies that have contributed to the decisions made within the VEOS project.

As yet, relatively few turnkey VE 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 virtual environment design systems.

Architectures for virtual environment 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 VE 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.[4]

More comprehensive overviews have been published for VE research directions (Bishop *et al*. 1992), for software (Zyda *et al*, 1993, Zyda *et al*, in press), 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).

Virtual environment development systems can be classified as *tool kits* or as *integrated software systems*. Tool kits commonly consist of a set of functions to be assembled into virtual events by a programmer. The most common language for tool kits is C; often the functionality is at a fairly low computational level, such as packaged device drivers. Integrated systems, in contrast, are intended for non-programmers and for domain experts such as graphic artists or stage directors. Integrated systems hide the low level computational details by providing more intuitive visual and default options for constructing events.

Of course some tool kits, such as 3D modeling software packages, have aspects of integrated systems. Similarly, some integrated systems require forms of scripting (i.e. symbolic programming) at one point or another. For comparison, VEOS is a hybrid. It provides the LISP language as a tool kit with generic functionality, and several specialized LISP functions for scripting entity behavior. VEOS is also highly integrated, since the entire virtual environment, the supporting network of computers, and all connected devices can be brought up by issuing a single instruction.

2.1.1 Toolkits

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

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 VE 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.

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 VE software functionality and links tightly to AutoCAD.

2.1.2 Integrated Systems

When the VEOS project began in 1990, VPL Research, Inc. manufactured RB2™, the first commercially available integrated VE 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 VE 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 VE simulation system, NPSNET (Zyda *et al*, 1992), has been developed at the Naval Postgraduate School.

2.2 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 VE 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 their compositions can represent common programming domains) (Coco, 1993). Industrial toolkits were not included in VEOS in order to avoid commercial dependencies. VEOS was intended to be distributed and copied freely for non-commercial usage.

As a research vehicle, VEOS emphasized functionality at the expense of performance. Premature optimization is a common source of difficulty in software research. So our efforts were directed first toward demonstrating that a thing could be done at all, then toward demonstrating how well we could do it. Since a research prototype must prepare for the future, VEOS was 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.

Naturally, the VEOS project passed through several phases over its three years of development. VEOS 0.6 first demonstrated multiple participants in 1991. VEOS 2.2 improved upon the conceptual design, but quickly became inefficient (relative to a 30 frame per second update rate) when the number of active nodes grew beyond a dozen (Coco & Lion, 1992). VEOS 3.0 emphasized performance.

VR is characterized by a rapid generation of applications ideas; it is the potential of VR that people find exciting. However, most complex VE 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. Since a participant in the virtual world ideally is free to engage in almost any behavior, VEOS is reactive, permitting easily constructed maps from participant behavior to virtual effect.

TABLE I: VEOS Design Approach

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

When more than one person inhabits a virtual world, the perspective of each participant is different. Different participants may see different views of the same graphical database or hear different sounds or feel different motion feedback. 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).

2.3 The VEOS Functional Architecture

Figure 1 presents the functional 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

[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.

participants and any number of hardware resources.[6] Naturally, transducers and tools are also duplicated for multiple participants.

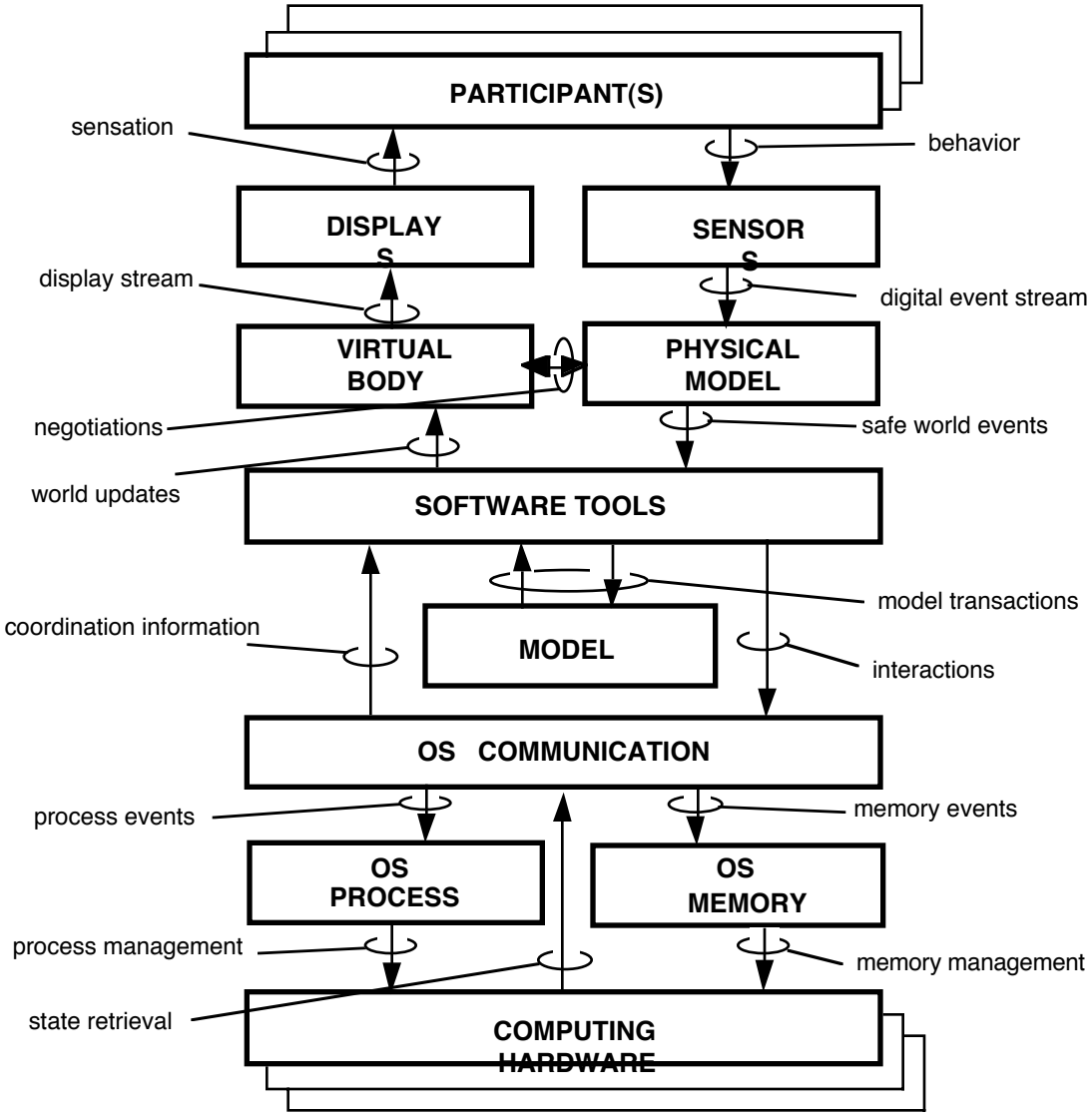

Figure 1. VEOS System Architecture.

[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.

2.4 The VEOS Kernel

The VEOS Kernel is a serious 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 design and implementation of virtual environments by taking care of operating system details. 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.

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.

2.4.1 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 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.[7] Nesting implements the concept of containment, so that the contents of a grouple can be interpreted as a set of items, a *grouplespace*.

---

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

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).

2.4.2 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, 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).

2.5 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.[8] With its features of systems orientation, biological modeling and active environments, FERN extends the VEOS Kernel infrastructure to form the entity-based programming model.

The entity concept is based on distributed object models (Jul *et al*, 1988). We first discuss related work which influenced the development of FERN, then we describe entities in detail.

2.5.1 Distributed Computation

Multi-processor computing is a growing trend (Spector, 1982; Li & Hudak, 1989; Kung *et al*, 1991). VE 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 serial ports. The VEOS architects chose to de-emphasize short-term performance issues

[8] As of mid 1993, higher level functions, such as inference engines and learning nets, have yet to be installed in entities.

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.

2.5.2 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.

A driving performance issue for VE 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 VE 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. EMERALD also 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 virtual environments 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.

2.6 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.

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.

Entities provide a uniform, singular metaphor 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 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.

2.6.1 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 generally 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.

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.

2.6.2 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 2 (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.

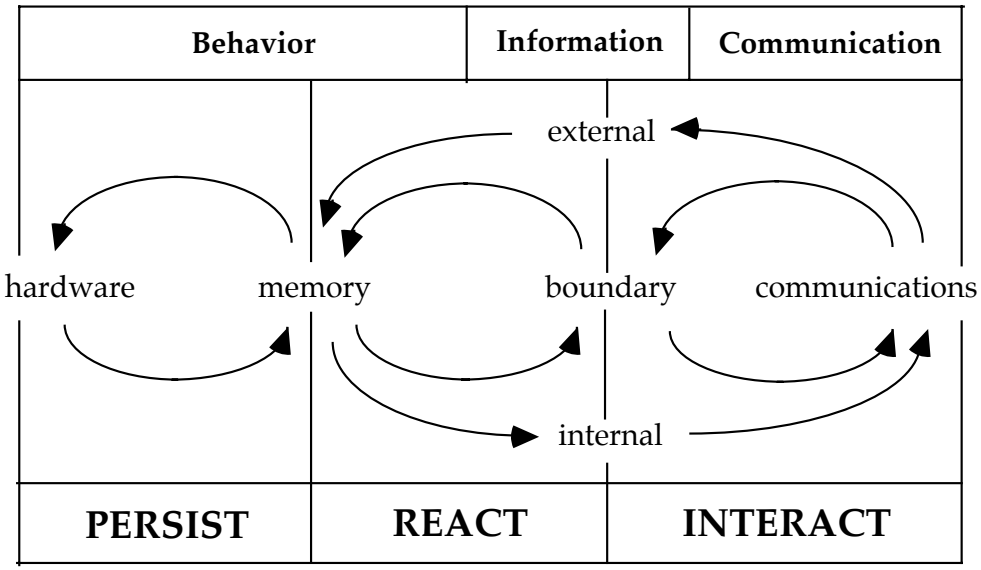

Figure 2: Functionality, Resources and Processes in an Entity

2.6.2.1 Internal Resources

The data used by an entity's processes is stored in five resource areas (Figure 2): 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. Figure 3 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.

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.

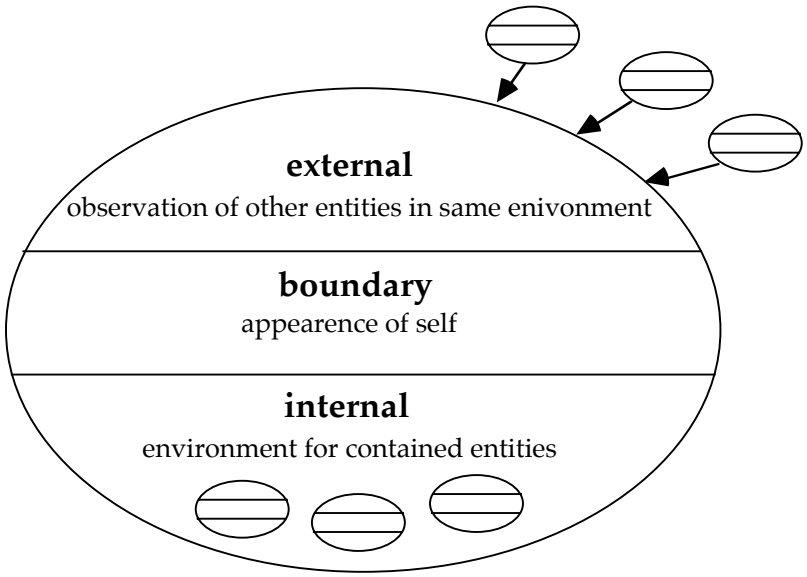

Figure 3: Entities as both Object and Environment

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.

2.6.2.2 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.[9]

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.

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.

2.6.3 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

[9] 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.

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 4 provides a graphical overview of the FERN programming model (Coco 1993).

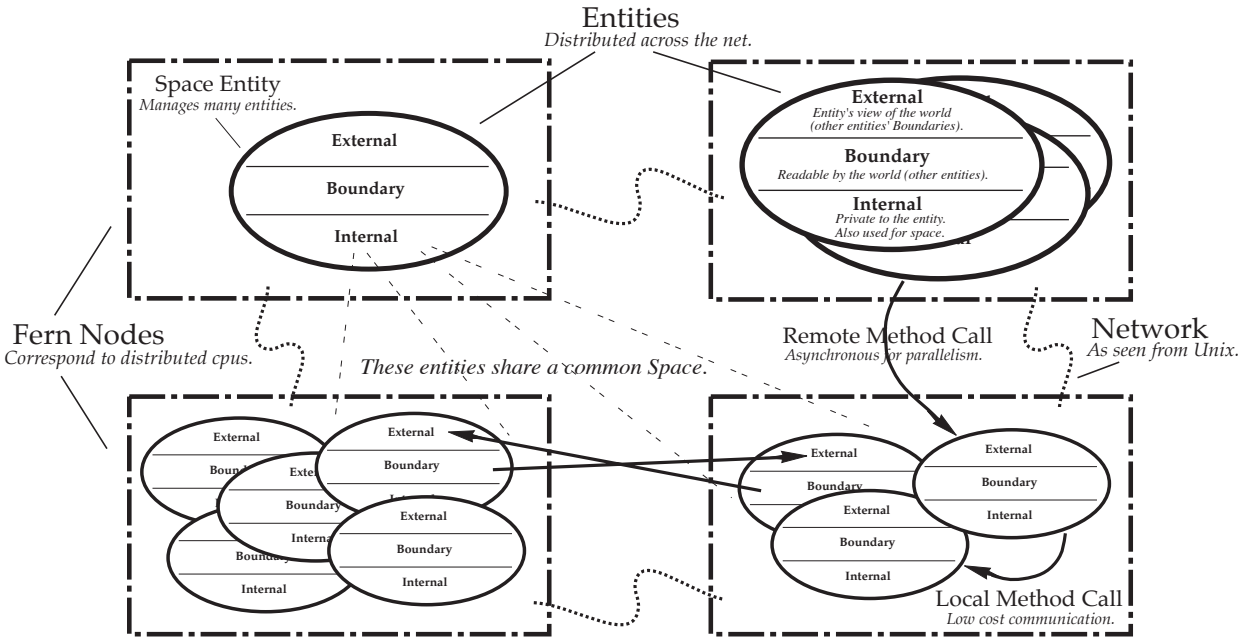

Figure 4: FERN Topology and Entity Structure

3. 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.

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.

4. 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 was the VEOS project, which spread over three and a half years and is now no longer active.

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 attempted 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.

VE design is inherently a multidisciplinary process and requires expertise in many different areas. Successful VE 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.

5. 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.

Zyda, M.J., *et al.* (in press) The software required for the computer generation of a virtual environment. *Presence*, 2(2).